

Matt Pharr, Wenzel Jakob, Greg Humphreys

# PHYSICALLY BASED RENDERING

From Theory to Implementation

**Third** Edition



**MK**  
MORGAN KAUFMANN

# Index of Notation

$p$	Point in 3D space	58
$v$	Direction vector in 3D space	59
$p_x, v_x$	$x$ component of point / vector	59
$\ v\ $	Length of vector $v$	63
$v \cdot w$	Dot product of vectors $v$ and $w$	63
$v \times w$	Cross product of vectors $v$ and $w$	64
$\hat{v}$	Normalized vector $v$	65
$T, M$	Linear transformation ( $4 \times 4$ matrix)	81
$(u, v)$	2D parametric coordinates of point on surface	116
$e(p)$	Directed edge function defined by two points	161
$\epsilon_m$	Machine epsilon	214
$\delta_a$	Absolute error	215
$\delta_r$	Relative error	215
$\gamma_n$	Gamma function for floating-point error bounds	216
$X(\lambda)/Y(\lambda)/Z(\lambda)$	XYZ spectral matching curves	322
$Q$	Joules (energy)	335
$\Phi$	Radiant flux (power)	335
$E$	Irradiance (flux area density)	336
$M$	Radiant exitance (flux area density)	336
$\omega$	Normalized vector on the unit sphere	338
$I$	Radiant intensity (flux density per solid angle)	338
$L$	Radiance (flux density per area per solid angle)	339
$V(\lambda)$	Spectral response curve	341
$Y$	Luminance	342
$\Omega$	Set of directions on the unit sphere	343
$\mathcal{H}^2(n)$	Hemisphere about surface normal $n$	343
$(\theta, \phi)$	Spherical coordinates	344
$f_r(\omega_i, \omega_o)$	BRDF (bidirectional reflectance distribution function)	350
$\mathcal{S}^2$	Sphere of directions	350
$f(\omega_i, \omega_o)$	BSDF (bidirectional scattering distribution function)	350
$S(p_o, \omega_o, p_i, \omega_i)$	BSSRDF (bidirectional scattering surface reflectance distribution function)	351
$F(\omega)$	Frequency-space representation of function $f(x)$	404
$\delta(x)$	Dirac delta distribution	405
$\mathbb{W}_T(x)$	Shah function with period $T$	405
$\otimes$	Convolution operation	406
$\Pi_T(x)$	Box function with period $T$	408
$\Phi_b(n)$	Radical inverse function, base $b$	443
$\rho_{hd}(\omega)$	Hemispherical-directional reflectance	514
$\rho_{hh}$	Hemispherical-hemispherical reflectance	515
$\eta_i, \eta_o$	Index of refraction (incident and outgoing media)	516
$F_r$	Fresnel reflectance	518
$D(\omega_h)$	Microfacet distribution function	537
$G_1(\omega)$	Microfacet masking-shadowing function	541
$G(\omega_o, \omega_i)$	Microfacet geometric attenuation function	543
$\omega_h$	Half-angle vector	545
$\mu_i, \mu_o$	Cosines of incident and outgoing direction vectors	552
$(s, t)$	2D texture coordinates	609
$\sigma_a$	Absorption cross section	673
$L_e(p, \omega)$	Emission in a participating medium	674
$\sigma_s$	Scattering coefficient	676

*Physically Based Rendering is a terrific book. It covers all the marvelous math, fascinating physics, practical software engineering, and clever tricks that are necessary to write a state-of-the-art photorealistic renderer. All of these topics are dealt with in a clear and pedagogical manner without omitting the all-important practical details.*

*pbrt is not just a “toy” implementation of a ray tracer but a general and robust full-scale global illumination renderer. It contains many important optimizations to reduce execution time and memory consumption for complex scenes. Furthermore, pbrt is easy to extend to experiment with other rendering algorithm variations.*

*This book is not only a textbook for students but also a useful reference book for practitioners in the field. The third edition has been extended with new sections on bidirectional path tracing, realistic camera models, and a state-of-the-art explanation of subsurface scattering.*

**Per Christensen**

Senior Software Developer, RenderMan Products, Pixar Animation Studios

*Looking for a job in research or high end rendering? Get your kick-start education and create your own project with this book that comes along with both theory and real examples, meaning real code and real content for your renderer.*

*With their third edition, Matt Pharr, Greg Humphreys, and Wenzel Jakob provide easy access to even the most advanced rendering techniques like multiplexed Metropolis light transport and quasi-Monte Carlo methods. Most importantly, the framework lets you skip the bootstrap pain of getting data into and out of your renderer.*

*The holistic approach of literate programming results in a clear logic of an easy-to-study text. If you are serious about graphics, there is no way around this unique and extremely valuable book that is closest to the state of the art.*

**Alexander Keller**

Director of Research, NVIDIA

This page intentionally left blank

# **Physically Based Rendering**

## **FROM THEORY TO IMPLEMENTATION**

**THIRD EDITION**

**MATT PHARR**

**WENZEL JAKOB**

**GREG HUMPHREYS**



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

Morgan Kaufmann is an imprint of Elsevier



**MORGAN KAUFMANN PUBLISHERS**

Morgan Kaufmann is an imprint of Elsevier  
50 Hampshire Street, 5th Floor, Cambridge, MA 02139, USA

© 2017 Elsevier Inc. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: [www.elsevier.com/permissions](http://www.elsevier.com/permissions).

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

#### Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

#### Library of Congress Cataloging-in-Publication Data

A catalog record for this book is available from the Library of Congress

#### British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library

ISBN: 978-0-12-800645-0

For information on all Morgan Kaufmann publications  
visit our website at <https://www.elsevier.com/>



Working together  
to grow libraries in  
developing countries

[www.elsevier.com](http://www.elsevier.com) • [www.bookaid.org](http://www.bookaid.org)

*Publisher:* Todd Green

*Editorial Project Manager:* Jennifer Pierce

*Production Project Manager:* Mohana Natarajan

*Cover Designer:* Victoria Pearson

Typeset by: Windfall Software and SPi global

**To Deirdre**, who even let me bring the manuscript on our honeymoon.

**M. P.**

**To Olesya**, who thought it was cute that my favorite book is a computer program.

**W. J.**

**To Isabel and Leila**, the two most extraordinary people I've ever met. May your pixels never be little squares.

**G. H.**

## ABOUT THE AUTHORS

**Matt Pharr** is a Software Engineer at Google. He previously co-founded Neoptica, which was acquired by Intel, and co-founded Exluna, which was acquired by NVIDIA. He has a B.S. degree from Yale and a Ph.D. from the Stanford Graphics Lab, where he worked under the supervision of Pat Hanrahan.

**Wenzel Jakob** is an assistant professor in the School of Computer and Communication Sciences at École Polytechnique Fédérale de Lausanne (EPFL). His research interests revolve around material appearance modeling, rendering algorithms, and the high-dimensional geometry of light paths. Wenzel obtained his Ph.D. at Cornell University under the supervision of Steve Marschner, after which he joined ETH Zürich for post-doctoral studies under the supervision of Olga Sorkine Hornung. Wenzel is also the lead developer of the Mitsuba renderer, a research-oriented rendering system.

**Greg Humphreys** is Director of Engineering at FanDuel, having previously worked on the Chrome graphics team at Google and the OptiX GPU ray-tracing engine at NVIDIA. Before that, he was a professor of Computer Science at the University of Virginia, where he conducted research in both high-performance and physically based computer graphics, as well as computer architecture and visualization. Greg has a B.S.E. degree from Princeton and a Ph.D. in Computer Science from Stanford under the supervision of Pat Hanrahan. When he's not tracing rays, Greg can usually be found playing tournament bridge.

# Contents

## PREFACE

xix

<b>CHAPTER 01. INTRODUCTION</b>	<b>1</b>
<b>1.1 Literate Programming</b>	<b>1</b>
1.1.1 Indexing and Cross-Referencing	3
<b>1.2 Photorealistic Rendering and the Ray-Tracing Algorithm</b>	<b>4</b>
1.2.1 Cameras	5
1.2.2 Ray–Object Intersections	7
1.2.3 Light Distribution	8
1.2.4 Visibility	10
1.2.5 Surface Scattering	10
1.2.6 Indirect Light Transport	11
1.2.7 Ray Propagation	13
<b>1.3 pbrt: System Overview</b>	<b>15</b>
1.3.1 Phases of Execution	17
1.3.2 Scene Representation	19
1.3.3 Integrator Interface and SamplerIntegrator	24
1.3.4 The Main Rendering Loop	26
1.3.5 An Integrator for Whitted Ray Tracing	32
<b>1.4 Parallelization of pbrt</b>	<b>38</b>
1.4.1 Data Races and Coordination	39
1.4.2 Conventions in pbrt	42
1.4.3 Thread Safety Expectations in pbrt	43
<b>1.5 How to Proceed through This Book</b>	<b>44</b>
1.5.1 The Exercises	45
<b>1.6 Using and Understanding the Code</b>	<b>45</b>
1.6.1 Pointer or Reference?	45
1.6.2 Abstraction versus Efficiency	46
1.6.3 Code Optimization	46
1.6.4 The Book Web site	47
1.6.5 Extending the System	47
1.6.6 Bugs	47
<b>1.7 A Brief History of Physically Based Rendering</b>	<b>48</b>
1.7.1 Research	48
1.7.2 Production	50
<b>Further Reading</b>	<b>53</b>
<b>Exercise</b>	<b>55</b>

**CHAPTER 02. GEOMETRY AND TRANSFORMATIONS 57**

<b>2.1</b>	<b>Coordinate Systems</b>	<b>57</b>
2.1.1	Coordinate System Handedness	58
<b>2.2</b>	<b>Vectors</b>	<b>59</b>
2.2.1	Dot and Cross Product	63
2.2.2	Normalization	65
2.2.3	Miscellaneous Operations	66
2.2.4	Coordinate System from a Vector	67
<b>2.3</b>	<b>Points</b>	<b>67</b>
<b>2.4</b>	<b>Normals</b>	<b>71</b>
<b>2.5</b>	<b>Rays</b>	<b>72</b>
2.5.1	Ray Differentials	74
<b>2.6</b>	<b>Bounding Boxes</b>	<b>75</b>
<b>2.7</b>	<b>Transformations</b>	<b>81</b>
2.7.1	Homogeneous Coordinates	82
2.7.2	Basic Operations	84
2.7.3	Translations	85
2.7.4	Scaling	87
2.7.5	$x$ , $y$ , and $z$ Axis Rotations	88
2.7.6	Rotation around an Arbitrary Axis	89
2.7.7	The Look-At Transformation	91
<b>2.8</b>	<b>Applying Transformations</b>	<b>93</b>
2.8.1	Points	93
2.8.2	Vectors	93
2.8.3	Normals	93
2.8.4	Rays	95
2.8.5	Bounding Boxes	95
2.8.6	Composition of Transformations	96
2.8.7	Transformations and Coordinate System Handedness	96
<b>* 2.9</b>	<b>Animating Transformations</b>	<b>97</b>
2.9.1	Quaternions	99
2.9.2	Quaternion Interpolation	101
2.9.3	AnimatedTransform Implementation	103
2.9.4	Bounding Moving Bounding Boxes	107
<b>2.10</b>	<b>Interactions</b>	<b>114</b>
2.10.1	Surface Interaction	116
<b>Further Reading</b>		<b>120</b>
<b>Exercises</b>		<b>121</b>

**CHAPTER 03. SHAPES 123**

<b>3.1</b>	<b>Basic Shape Interface</b>	<b>123</b>
3.1.1	Bounding	124
3.1.2	Ray-Bounds Intersections	125

3.1.3	Intersection Tests	129
3.1.4	Surface Area	130
3.1.5	Sidedness	131
<b>3.2</b>	<b>Spheres</b>	<b>131</b>
3.2.1	Bounding	133
3.2.2	Intersection Tests	134
* 3.2.3	Partial Derivatives of Normal Vectors	138
3.2.4	SurfaceInteraction Initialization	140
3.2.5	Surface Area	141
<b>3.3</b>	<b>Cylinders</b>	<b>142</b>
3.3.1	Bounding	143
3.3.2	Intersection Tests	144
3.3.3	Surface Area	146
<b>3.4</b>	<b>Disks</b>	<b>146</b>
3.4.1	Bounding	147
3.4.2	Intersection Tests	148
3.4.3	Surface Area	150
<b>3.5</b>	<b>Other Quadrics</b>	<b>150</b>
3.5.1	Cones	150
3.5.2	Paraboloids	151
3.5.3	Hyperboloids	152
<b>3.6</b>	<b>Triangle Meshes</b>	<b>152</b>
3.6.1	Triangle	155
3.6.2	Triangle Intersection	157
3.6.3	Shading Geometry	166
3.6.4	Surface Area	167
* 3.7	<b>Curves</b>	<b>167</b>
* 3.8	<b>Subdivision Surfaces</b>	<b>181</b>
3.8.1	Mesh Representation	184
3.8.2	Subdivision	192
* 3.9	<b>Managing Rounding Error</b>	<b>206</b>
3.9.1	Floating-Point Arithmetic	208
3.9.2	Conservative Ray-Bounds Intersections	220
3.9.3	Robust Triangle Intersections	221
3.9.4	Bounding Intersection Point Error	222
3.9.5	Robust Spawned Ray Origins	230
3.9.6	Avoiding Intersections Behind Ray Origins	233
3.9.7	Discussion	235
	<b>Further Reading</b>	<b>236</b>
	<b>Exercises</b>	<b>238</b>

## CHAPTER 04. PRIMITIVES AND INTERSECTION ACCELERATION

247

<b>4.1</b>	<b>Primitive Interface and Geometric Primitives</b>	<b>248</b>
4.1.1	Geometric Primitives	250
4.1.2	TransformedPrimitive: Object Instancing and Animated Primitives	251
<b>4.2</b>	<b>Aggregates</b>	<b>254</b>

<b>4.3</b>	<b>Bounding Volume Hierarchies</b>	<b>255</b>
4.3.1	BVH Construction	257
4.3.2	The Surface Area Heuristic	263
4.3.3	Linear Bounding Volume Hierarchies	268
4.3.4	Compact BVH For Traversal	280
4.3.5	Traversal	282
<b>4.4</b>	<b>Kd-Tree Accelerator</b>	<b>284</b>
4.4.1	Tree Representation	286
4.4.2	Tree Construction	288
4.4.3	Traversal	297
	<b>Further Reading</b>	<b>302</b>
	<b>Exercises</b>	<b>308</b>

## CHAPTER 05. COLOR AND RADIOMETRY 313

<b>5.1</b>	<b>Spectral Representation</b>	<b>313</b>
5.1.1	The Spectrum Type	315
5.1.2	CoefficientSpectrum Implementation	315
<b>5.2</b>	<b>The SampledSpectrum Class</b>	<b>318</b>
5.2.1	XYZ Color	322
5.2.2	RGB Color	325
<b>5.3</b>	<b>RGBSpectrum Implementation</b>	<b>331</b>
<b>5.4</b>	<b>Radiometry</b>	<b>334</b>
5.4.1	Basic Quantities	335
5.4.2	Incident and Exitant Radiance Functions	339
5.4.3	Luminance and Photometry	341
<b>5.5</b>	<b>Working with Radiometric Integrals</b>	<b>343</b>
5.5.1	Integrals over Projected Solid Angle	343
5.5.2	Integrals over Spherical Coordinates	344
5.5.3	Integrals over Area	347
<b>5.6</b>	<b>Surface Reflection</b>	<b>348</b>
5.6.1	The BRDF	349
5.6.2	The BSSRDF	351
	<b>Further Reading</b>	<b>352</b>
	<b>Exercises</b>	<b>353</b>

## CHAPTER 06. CAMERA MODELS 355

<b>6.1</b>	<b>Camera Model</b>	<b>356</b>
6.1.1	Camera Coordinate Spaces	358
<b>6.2</b>	<b>Projective Camera Models</b>	<b>358</b>
6.2.1	Orthographic Camera	361
6.2.2	Perspective Camera	365
6.2.3	The Thin Lens Model and Depth of Field	368
<b>6.3</b>	<b>Environment Camera</b>	<b>375</b>

<b>* 6.4</b>	<b>Realistic Cameras</b>	<b>377</b>
6.4.1	Lens System Representation	379
6.4.2	Tracing Rays through Lenses	382
6.4.3	The Thick Lens Approximation	386
6.4.4	Focusing	388
6.4.5	The Exit Pupil	389
6.4.6	Generating Rays	394
6.4.7	The Camera Measurement Equation	395
	<b>Further Reading</b>	<b>397</b>
	<b>Exercises</b>	<b>398</b>

## **CHAPTER 07. SAMPLING AND RECONSTRUCTION**

		<b>401</b>
<b>7.1</b>	<b>Sampling Theory</b>	<b>402</b>
7.1.1	The Frequency Domain and the Fourier Transform	403
7.1.2	Ideal Sampling and Reconstruction	405
7.1.3	Aliasing	409
7.1.4	Antialiasing Techniques	410
7.1.5	Application to Image Synthesis	413
7.1.6	Sources of Aliasing in Rendering	414
7.1.7	Understanding Pixels	415
<b>7.2</b>	<b>Sampling Interface</b>	<b>416</b>
* 7.2.1	Evaluating Sample Patterns: Discrepancy	417
7.2.2	Basic Sampler Interface	421
7.2.3	Sampler Implementation	425
7.2.4	Pixel Sampler	427
7.2.5	Global Sampler	428
<b>7.3</b>	<b>Stratified Sampling</b>	<b>432</b>
<b>* 7.4</b>	<b>The Halton Sampler</b>	<b>441</b>
7.4.1	Hammersley and Halton Sequences	443
7.4.2	Halton Sampler Implementation	450
<b>* 7.5</b>	<b>(0, 2)-Sequence Sampler</b>	<b>454</b>
7.5.1	Sampling with Generator Matrices	455
7.5.2	Sampler Implementation	462
<b>* 7.6</b>	<b>Maximized Minimal Distance Sampler</b>	<b>465</b>
<b>* 7.7</b>	<b>Sobol' Sampler</b>	<b>467</b>
<b>7.8</b>	<b>Image Reconstruction</b>	<b>472</b>
7.8.1	Filter Functions	474
<b>7.9</b>	<b>Film and the Imaging Pipeline</b>	<b>483</b>
7.9.1	The Film Class	484
7.9.2	Supplying Pixel Values to the Film	488
7.9.3	Image Output	494
	<b>Further Reading</b>	<b>496</b>
	<b>Exercises</b>	<b>501</b>

<b>CHAPTER 08. REFLECTION MODELS</b>	<b>507</b>
<b>8.1 Basic Interface</b>	<b>512</b>
8.1.1 Reflectance	514
8.1.2 BxDF Scaling Adapter	515
<b>8.2 Specular Reflection and Transmission</b>	<b>516</b>
8.2.1 Fresnel Reflectance	516
8.2.2 Specular Reflection	523
8.2.3 Specular Transmission	526
8.2.4 Fresnel-Modulated Specular Reflection and Transmission	531
<b>8.3 Lambertian Reflection</b>	<b>532</b>
<b>8.4 Microfacet Models</b>	<b>533</b>
8.4.1 Oren–Nayar Diffuse Reflection	534
8.4.2 Microfacet Distribution Functions	537
8.4.3 Masking and Shadowing	541
8.4.4 The Torrance–Sparrow Model	544
<b>8.5 Fresnel Incidence Effects</b>	<b>549</b>
<b>8.6 Fourier Basis BSDFs</b>	<b>552</b>
8.6.1 Spline Interpolation	560
<b>Further Reading</b>	<b>563</b>
<b>Exercises</b>	<b>567</b>
<b>CHAPTER 09. MATERIALS</b>	<b>571</b>
<b>9.1 BSDFs</b>	<b>571</b>
9.1.1 BSDF Memory Management	576
<b>9.2 Material Interface and Implementations</b>	<b>577</b>
9.2.1 Matte Material	578
9.2.2 Plastic Material	579
9.2.3 Mix Material	581
9.2.4 Fourier Material	583
9.2.5 Additional Materials	584
<b>9.3 Bump Mapping</b>	<b>584</b>
<b>Further Reading</b>	<b>591</b>
<b>Exercises</b>	<b>592</b>
<b>CHAPTER 10. TEXTURE</b>	<b>597</b>
<b>10.1 Sampling and Antialiasing</b>	<b>598</b>
10.1.1 Finding the Texture Sampling Rate	599
10.1.2 Filtering Texture Functions	604
* 10.1.3 Ray Differentials for Specular Reflection and Transmission	605
<b>10.2 Texture Coordinate Generation</b>	<b>608</b>
10.2.1 2D ( $u, v$ ) Mapping	610
10.2.2 Spherical Mapping	611
10.2.3 Cylindrical Mapping	612

10.2.4	Planar Mapping	613
10.2.5	3D Mapping	613
<b>10.3</b>	<b>Texture Interface and Basic Textures</b>	<b>614</b>
10.3.1	Constant Texture	615
10.3.2	Scale Texture	615
10.3.3	Mix Textures	616
10.3.4	Bilinear Interpolation	617
<b>10.4</b>	<b>Image Texture</b>	<b>618</b>
10.4.1	Texture Memory Management	620
10.4.2	ImageTexture Evaluation	622
10.4.3	MIP Maps	623
10.4.4	Isotropic Triangle Filter	632
* 10.4.5	Elliptically Weighted Average	634
<b>10.5</b>	<b>Solid and Procedural Texturing</b>	<b>640</b>
10.5.1	UV Texture	641
10.5.2	Checkerboard	642
10.5.3	Solid Checkerboard	646
<b>10.6</b>	<b>Noise</b>	<b>648</b>
10.6.1	Perlin Noise	649
10.6.2	Random Polka Dots	653
10.6.3	Noise Idioms and Spectral Synthesis	655
10.6.4	Bumpy and Wrinkled Textures	660
10.6.5	Windy Waves	662
10.6.6	Marble	663
	<b>Further Reading</b>	<b>664</b>
	<b>Exercises</b>	<b>667</b>

**\* CHAPTER 11. VOLUME SCATTERING****671**

<b>11.1</b>	<b>Volume Scattering Processes</b>	<b>672</b>
11.1.1	Absorption	673
11.1.2	Emission	674
11.1.3	Out-Scattering and Attenuation	676
11.1.4	In-scattering	678
<b>11.2</b>	<b>Phase Functions</b>	<b>680</b>
<b>11.3</b>	<b>Media</b>	<b>684</b>
11.3.1	Medium Interactions	687
11.3.2	Homogeneous Medium	688
11.3.3	3D Grids	690
<b>11.4</b>	<b>The BSSRDF</b>	<b>692</b>
11.4.1	Separable BSSRDFs	693
11.4.2	Tabulated BSSRDF	696
11.4.3	Subsurface Scattering Materials	700
	<b>Further Reading</b>	<b>702</b>
	<b>Exercises</b>	<b>703</b>

<b>CHAPTER 12. LIGHT SOURCES</b>	<b>707</b>
<b>12.1 Light Emission</b>	<b>708</b>
12.1.1 Blackbody Emitters	709
12.1.2 Standard Illuminants	712
<b>12.2 Light Interface</b>	<b>714</b>
12.2.1 Visibility Testing	717
<b>12.3 Point Lights</b>	<b>719</b>
12.3.1 Spotlights	721
12.3.2 Texture Projection Lights	724
12.3.3 Goniophotometric Diagram Lights	728
<b>12.4 Distant Lights</b>	<b>731</b>
<b>12.5 Area Lights</b>	<b>733</b>
<b>12.6 Infinite Area Lights</b>	<b>737</b>
<b>Further Reading</b>	<b>741</b>
<b>Exercises</b>	<b>744</b>
<b>CHAPTER 13. MONTE CARLO INTEGRATION</b>	<b>747</b>
<b>13.1 Background and Probability Review</b>	<b>748</b>
13.1.1 Continuous Random Variables	749
13.1.2 Expected Values and Variance	750
<b>13.2 The Monte Carlo Estimator</b>	<b>751</b>
<b>13.3 Sampling Random Variables</b>	<b>753</b>
13.3.1 The Inversion Method	753
13.3.2 The Rejection Method	760
<b>* 13.4 Metropolis Sampling</b>	<b>762</b>
13.4.1 Basic Algorithm	763
13.4.2 Choosing Mutation Strategies	764
13.4.3 Start-up Bias	766
13.4.4 1D Setting	766
13.4.5 Estimating Integrals with Metropolis Sampling	771
<b>13.5 Transforming between Distributions</b>	<b>771</b>
13.5.1 Transformation in Multiple Dimensions	772
13.5.2 Polar Coordinates	772
13.5.3 Spherical Coordinates	773
<b>13.6 2D Sampling with Multidimensional Transformations</b>	<b>773</b>
13.6.1 Uniformly Sampling a Hemisphere	774
13.6.2 Sampling a Unit Disk	776
13.6.3 Cosine-Weighted Hemisphere Sampling	779
13.6.4 Sampling a Cone	781
13.6.5 Sampling a Triangle	781
* 13.6.6 Sampling Cameras	783
13.6.7 Piecewise-Constant 2D Distributions	784
<b>13.7 Russian Roulette and Splitting</b>	<b>787</b>
13.7.1 Splitting	788

<b>13.8</b>	<b>Careful Sample Placement</b>	<b>789</b>
13.8.1	Stratified Sampling	789
13.8.2	Quasi Monte Carlo	792
13.8.3	Warping Samples and Distortion	792
<b>13.9</b>	<b>Bias</b>	<b>793</b>
<b>13.10</b>	<b>Importance Sampling</b>	<b>794</b>
13.10.1	Multiple Importance Sampling	797
	<b>Further Reading</b>	<b>799</b>
	<b>Exercises</b>	<b>801</b>

## CHAPTER 14. LIGHT TRANSPORT I: SURFACE REFLECTION

805

<b>14.1</b>	<b>Sampling Reflection Functions</b>	<b>806</b>
14.1.1	Microfacet BxDFs	807
14.1.2	FresnelBlend	814
14.1.3	Specular Reflection and Transmission	815
* 14.1.4	Fourier BSDF	817
14.1.5	Application: Estimating Reflectance	830
14.1.6	Sampling BSDFs	832
<b>14.2</b>	<b>Sampling Light Sources</b>	<b>835</b>
14.2.1	Lights with Singularities	836
14.2.2	Sampling Shapes	836
14.2.3	Area Lights	845
14.2.4	Infinite Area Lights	845
<b>14.3</b>	<b>Direct Lighting</b>	<b>851</b>
14.3.1	Estimating the Direct Lighting Integral	856
<b>14.4</b>	<b>The Light Transport Equation</b>	<b>861</b>
14.4.1	Basic Derivation	862
14.4.2	Analytic Solutions to the LTE	863
14.4.3	The Surface Form of the LTE	865
14.4.4	Integral over Paths	866
14.4.5	Delta Distributions in the Integrand	868
14.4.6	Partitioning the Integrand	869
<b>14.5</b>	<b>Path Tracing</b>	<b>870</b>
14.5.1	Overview	872
14.5.2	Path Sampling	873
14.5.3	Incremental Path Construction	874
14.5.4	Implementation	875
	<b>Further Reading</b>	<b>879</b>
	<b>Exercises</b>	<b>882</b>

## CHAPTER 15. LIGHT TRANSPORT II: VOLUME RENDERING

887

<b>15.1</b>	<b>The Equation of Transfer</b>	<b>888</b>
* 15.1.1	Generalized Path Space	890

<b>15.2</b>	<b>Sampling Volume Scattering</b>	<b>891</b>
15.2.1	Homogeneous Medium	893
15.2.2	Heterogeneous Medium	894
15.2.3	Sampling Phase Functions	898
<b>15.3</b>	<b>Volumetric Light Transport</b>	<b>899</b>
15.3.1	Path Tracing	900
<b>* 15.4</b>	<b>Sampling Subsurface Reflection Functions</b>	<b>903</b>
15.4.1	Sampling the SeparableBSSRDF	905
15.4.2	Sampling the TabulatedBSSRDF	913
15.4.3	Subsurface Scattering in the Path Tracer	915
<b>* 15.5</b>	<b>Subsurface Scattering Using the Diffusion Equation</b>	<b>916</b>
15.5.1	Principle of Similarity	917
15.5.2	Diffusion Theory	918
15.5.3	Monopole Solution	923
15.5.4	Non-classical Diffusion	924
15.5.5	Dipole Solution	925
15.5.6	Beam Solution	928
15.5.7	Single Scattering Term	930
15.5.8	Filling the BSSRDFTable	934
15.5.9	Setting Scattering Properties	938
	<b>Further Reading</b>	<b>939</b>
	<b>Exercises</b>	<b>943</b>

**\* CHAPTER 16. LIGHT TRANSPORT III:  
BIDIRECTIONAL METHODS** **947**

<b>16.1</b>	<b>The Path-Space Measurement Equation</b>	<b>948</b>
16.1.1	Sampling Cameras	949
16.1.2	Sampling Light Rays	955
16.1.3	Non-symmetric Scattering	960
<b>16.2</b>	<b>Stochastic Progressive Photon Mapping</b>	<b>963</b>
16.2.1	Theoretical Basis for Particle Tracing	963
16.2.2	Photon Mapping	966
16.2.3	SPPMIntegrator	972
16.2.4	Accumulating Visible Points	975
16.2.5	Visible Point Grid Construction	979
16.2.6	Accumulating Photon Contributions	983
<b>16.3</b>	<b>Bidirectional Path Tracing</b>	<b>990</b>
16.3.1	Vertex Abstraction Layer	995
16.3.2	Generating the Camera and Light Subpaths	1003
16.3.3	Subpath Connections	1008
16.3.4	Multiple Importance Sampling	1012
16.3.5	Infinite Area Lights and BDPT	1019
<b>16.4</b>	<b>Metropolis Light Transport</b>	<b>1022</b>
16.4.1	Primary Sample Space MLT	1023
16.4.2	Multiplexed MLT	1025
16.4.3	Application to Rendering	1025

16.4.4	Primary Sample Space Sampler	1028
16.4.5	MLT Integrator	1035
<b>Further Reading</b>		<b>1042</b>
<b>Exercises</b>		<b>1046</b>

## CHAPTER 17. RETROSPECTIVE AND THE FUTURE 1051

<b>17.1</b>	<b>Design Retrospective</b>	<b>1051</b>
17.1.1	Triangles Only	1052
17.1.2	Increased Scene Complexity	1053
17.1.3	Production Rendering	1054
17.1.4	Specialized Compilation	1054
<b>17.2</b>	<b>Alternative Hardware Architectures</b>	<b>1055</b>
17.2.1	GPU Ray Tracing	1056
17.2.2	Packet Tracing	1057
17.2.3	Ray-Tracing Hardware	1059
17.2.4	The Future	1059
<b>17.3</b>	<b>Conclusion</b>	<b>1060</b>

## APPENDIXES

<b>A</b>	<b>UTILITIES</b>	<b>1061</b>
<b>B</b>	<b>SCENE DESCRIPTION INTERFACE</b>	<b>1103</b>
<b>C</b>	<b>INDEX OF FRAGMENTS</b>	<b>1135</b>
<b>D</b>	<b>INDEX OF CLASSES AND THEIR MEMBERS</b>	<b>1151</b>
<b>E</b>	<b>INDEX OF MISCELLANEOUS IDENTIFIERS</b>	<b>1161</b>
	<b>REFERENCES</b>	<b>1165</b>
	<b>SUBJECT INDEX</b>	<b>1213</b>
	<b>COLOPHON</b>	<b>1235</b>

This page intentionally left blank

# Preface

*[Just as] other information should be available to those who want to learn and understand, program source code is the only means for programmers to learn the art from their predecessors. It would be unthinkable for playwrights not to allow other playwrights to read their plays [or to allow them] at theater performances where they would be barred even from taking notes. Likewise, any good author is well read, as every child who learns to write will read hundreds of times more than it writes. Programmers, however, are expected to invent the alphabet and learn to write long novels all on their own. Programming cannot grow and learn unless the next generation of programmers has access to the knowledge and information gathered by other programmers before them.* —Erik Naggum

Rendering is a fundamental component of computer graphics. At the highest level of abstraction, rendering is the process of converting a description of a three-dimensional scene into an image. Algorithms for animation, geometric modeling, texturing, and other areas of computer graphics all must pass their results through some sort of rendering process so that they can be made visible in an image. Rendering has become ubiquitous; from movies to games and beyond, it has opened new frontiers for creative expression, entertainment, and visualization.

In the early years of the field, research in rendering focused on solving fundamental problems such as determining which objects are visible from a given viewpoint. As effective solutions to these problems have been found and as richer and more realistic scene descriptions have become available thanks to continued progress in other areas of graphics, modern rendering has grown to include ideas from a broad range of disciplines, including physics and astrophysics, astronomy, biology, psychology and the study of perception, and pure and applied mathematics. The interdisciplinary nature of rendering is one of the reasons that it is such a fascinating area of study.

This book presents a selection of modern rendering algorithms through the documented source code for a complete rendering system. Nearly all of the images in this book, including the one on the front cover, were rendered by this software. All of the algorithms that came together to generate these images are described in these pages. The system, `pbrt`, is written using a programming methodology called *literate programming* that mixes prose describing the system with the source code that implements it. We believe that the literate programming approach is a valuable way to introduce ideas in computer graphics and computer science in general. Often, some of the subtleties of an algorithm can be unclear or hidden until it is implemented, so seeing an actual implementation is a good way to acquire a solid understanding of that algorithm's details. Indeed, we believe that deep understanding of a small number of algorithms in this manner provides a stronger base for further study of computer graphics than does superficial understanding of many.

In addition to clarifying how an algorithm is implemented in practice, presenting these algorithms in the context of a complete and nontrivial software system also allows us to address issues in the design and implementation of medium-sized rendering systems. The design of a rendering system’s basic abstractions and interfaces has substantial implications for both the elegance of the implementation and the ability to extend it later, yet the trade-offs in this design space are rarely discussed.

`pbrt` and the contents of this book focus exclusively on *photorealistic rendering*, which can be defined variously as the task of generating images that are indistinguishable from those that a camera would capture in a photograph or as the task of generating images that evoke the same response from a human observer as looking at the actual scene. There are many reasons to focus on photorealism. Photorealistic images are crucial for the movie special-effects industry because computer-generated imagery must often be mixed seamlessly with footage of the real world. In entertainment applications where all of the imagery is synthetic, photorealism is an effective tool for making the observer forget that he or she is looking at an environment that does not actually exist. Finally, photorealism gives a reasonably well-defined metric for evaluating the quality of the rendering system’s output.

## AUDIENCE

There are three main audiences that this book is intended for. The first is students in graduate or upper-level undergraduate computer graphics classes. This book assumes existing knowledge of computer graphics at the level of an introductory college-level course, although certain key concepts such as basic vector geometry and transformations will be reviewed here. For students who do not have experience with programs that have tens of thousands of lines of source code, the literate programming style gives a gentle introduction to this complexity. We pay special attention to explaining the reasoning behind some of the key interfaces and abstractions in the system in order to give these readers a sense of why the system is structured in the way that it is.

The second audience is advanced graduate students and researchers in computer graphics. For those doing research in rendering, the book provides a broad introduction to the area, and the `pbrt` source code provides a foundation that can be useful to build upon (or at least to use bits of source code from). For those working in other areas, we believe that having a thorough understanding of rendering can be helpful context to carry along.

Our final audience is software developers in industry. Although many of the ideas in this book will likely be familiar to this audience, seeing explanations of the algorithms presented in the literate style may provide new perspectives. `pbrt` includes implementations of a number of advanced and/or difficult-to-implement algorithms and techniques, such as subdivision surfaces, Monte Carlo sampling algorithms, bidirectional path tracing, Metropolis sampling, and subsurface scattering; these should be of particular interest to experienced practitioners in rendering. We hope that delving into one particular organization of a complete and nontrivial rendering system will also be thought provoking to this audience.

## OVERVIEW AND GOALS

`pbrt` is based on the *ray-tracing* algorithm. Ray tracing is an elegant technique that has its origins in lens making; Carl Friedrich Gauß traced rays through lenses by hand in the 19th century. Ray-tracing algorithms on computers follow the path of infinitesimal rays of light through the scene until they intersect a surface. This approach gives a simple method for finding the first visible object as seen from any particular position and direction and is the basis for many rendering algorithms.

`pbrt` was designed and implemented with three main goals in mind: it should be *complete*, it should be *illustrative*, and it should be *physically based*.

Completeness implies that the system should not lack key features found in high-quality commercial rendering systems. In particular, it means that important practical issues, such as antialiasing, robustness, numerical precision, and the ability to efficiently render complex scenes, should all be addressed thoroughly. It is important to consider these issues from the start of the system’s design, since these features can have subtle implications for all components of the system and can be quite difficult to retrofit into the system at a later stage of implementation.

Our second goal means that we tried to choose algorithms, data structures, and rendering techniques with care and with an eye toward readability and clarity. Since their implementations will be examined by more readers than is the case for many other rendering systems, we tried to select the most elegant algorithms that we were aware of and implement them as well as possible. This goal also required that the system be small enough for a single person to understand completely. We have implemented `pbrt` using an extensible architecture, with the core of the system implemented in terms of a set of carefully designed abstract base classes, and as much of the specific functionality as possible in implementations of these base classes. The result is that one doesn’t need to understand all of the specific implementations in order to understand the basic structure of the system. This makes it easier to delve deeply into parts of interest and skip others, without losing sight of how the overall system fits together.

There is a tension between the two goals of being complete and being illustrative. Implementing and describing every possible useful technique would not only make this book unacceptably long, but also would make the system prohibitively complex for most readers. In cases where `pbrt` lacks a particularly useful feature, we have attempted to design the architecture so that the feature could be added without altering the overall system design.

The basic foundations for physically based rendering are the laws of physics and their mathematical expression. `pbrt` was designed to use the correct physical units and concepts for the quantities it computes and the algorithms it implements. When configured to do so, `pbrt` can compute images that are *physically correct*; they accurately reflect the lighting as it would be in a real-world version of the scene. One advantage of the decision to use a physical basis is that it gives a concrete standard of program correctness: for simple scenes, where the expected result can be computed in closed form, if `pbrt` doesn’t compute the same result, we know there must be a bug in the implementation.

Similarly, if different physically based lighting algorithms in pbrt give different results for the same scene, or if pbrt doesn't give the same results as another physically based renderer, there is certainly an error in one of them. Finally, we believe that this physically based approach to rendering is valuable because it is rigorous. When it is not clear how a particular computation should be performed, physics gives an answer that guarantees a consistent result.

Efficiency was given lower priority than these three goals. Since rendering systems often run for many minutes or hours in the course of generating an image, efficiency is clearly important. However, we have mostly confined ourselves to *algorithmic* efficiency rather than low-level code optimization. In some cases, obvious micro-optimizations take a backseat to clear, well-organized code, although we did make some effort to optimize the parts of the system where most of the computation occurs.

In the course of presenting pbrt and discussing its implementation, we hope to convey some hard-learned lessons from years of rendering research and development. There is more to writing a good renderer than stringing together a set of fast algorithms; making the system both flexible and robust is a difficult task. The system's performance must degrade gracefully as more geometry or light sources are added to it or as any other axis of complexity is pushed. Numerical stability must be handled carefully, and algorithms that don't waste floating-point precision are critical.

The rewards for developing a system that addresses all these issues are enormous—it is a great pleasure to write a new renderer or add a new feature to an existing renderer and use it to create an image that couldn't be generated before. Our most fundamental goal in writing this book was to bring this opportunity to a wider audience. Readers are encouraged to use the system to render the example scenes in the pbrt software distribution as they progress through the book. Exercises at the end of each chapter suggest modifications to the system that will help clarify its inner workings and more complex projects to extend the system by adding new features.

The Web site for this book is located at [www.pbrt.org](http://www.pbrt.org). The latest version of the pbrt source code is available from this site, and we will also post errata and bug fixes, additional scenes to render, and supplemental utilities. Any bugs in pbrt or errors in this text that are not listed at the Web site can be reported to the email address [bugs@pbrt.org](mailto:bugs@pbrt.org). We greatly value your feedback!

## CHANGES BETWEEN THE FIRST AND SECOND EDITIONS

Six years passed between the publication of the first edition of this book in 2004 and the second edition in 2010. In that time, thousands of copies of the book were sold, and the pbrt software was downloaded thousands of times from the book's Web site. The pbrt user base gave us a significant amount of feedback and encouragement, and our experience with the system guided many of the decisions we made in making changes between the version of pbrt presented in the first edition and the version in the second edition. In addition to a number of bug fixes, we also made several significant design changes and enhancements:

- Removal of the plugin architecture. The first version of `pbrt` used a run-time plugin architecture to dynamically load code for implementations of objects like shapes, lights, integrators, cameras, and other objects that were used in the scene currently being rendered. This approach allowed users to extend `pbrt` with new object types (e.g., new shape primitives) without recompiling the entire rendering system. This approach initially seemed elegant, but it complicated the task of supporting `pbrt` on multiple platforms and it made debugging more difficult. The only new usage scenario that it truly enabled (binary-only distributions of `pbrt` or binary plugins) was actually contrary to our pedagogical and open-source goals. Therefore, the plugin architecture was dropped in this edition.
- Removal of the image-processing pipeline. The first version of `pbrt` provided a tone-mapping interface that converted high-dynamic-range (HDR) floating-point output images directly into low-dynamic-range TIFFs for display. This functionality made sense in 2004, as support for HDR images was still sparse. In 2010, however, advances in digital photography had made HDR images commonplace. Although the theory and practice of tone mapping are elegant and worth learning, we decided to focus the new book exclusively on the process of image formation and skip the topic of image display. Interested readers should read the book written by Reinhard et al. (2010) for a thorough and modern treatment of the HDR image display process.
- Task parallelism. Multicore architectures became ubiquitous, and we felt that `pbrt` would not remain relevant without the ability to scale to the number of locally available cores. We also hoped that the parallel programming implementation details documented in this book would help graphics programmers understand some of the subtleties and complexities in writing scalable parallel code (e.g., choosing appropriate task granularities), which is still a difficult and too infrequently taught topic.
- Appropriateness for “production” rendering. The first version of `pbrt` was intended exclusively as a pedagogical tool and a stepping-stone for rendering research. Indeed, we made a number of decisions in preparing the first edition that were contrary to use in a production environment, such as limited support for image-based lighting, no support for motion blur, and a photon mapping implementation that wasn’t robust in the presence of complex lighting. With much improved support for these features as well as support for subsurface scattering and Metropolis light transport, we feel that with the second edition, `pbrt` became much more suitable for rendering very high-quality images of complex environments.

## CHANGES BETWEEN THE SECOND AND THIRD EDITIONS

With the passage of another six years, it was time to update and extend the book and the `pbrt` system. We continued to learn from readers’ and users’ experiences to better understand which topics were most useful to cover. Further, rendering research continued apace; many parts of the book were due for an update to reflect current best practices. We made significant improvements on a number of fronts:

- Bidirectional light transport. The third version of `pbrt` now includes a full-featured bidirectional path tracer, including full support for volumetric light transport and multiple importance sampling to weight paths. An all-new Metropolis light transport integrator uses components of the bidirectional path tracer, allowing for a particularly succinct implementation of that algorithm. The foundations of these algorithms were established approximately fifteen years ago; it's overdue to have solid support for them in `pbrt`.
- Subsurface scattering. The appearance of many objects—notably, skin and translucent objects—is a result of subsurface light transport. Our implementation of subsurface scattering in the second edition reflected the state of the art in the early 2000s; we have thoroughly updated both our BSSRDF models and our subsurface light transport algorithms to reflect the progress made in ten subsequent years of research. We now use a considerably more accurate diffusion solution together with a ray-tracing-based sampling technique, removing the need for the costly preprocessing step used in the second edition.
- Numerically robust intersections. The effects of floating-point round-off error in geometric ray intersection calculations have been a long-standing challenge in ray tracing: they can cause small errors to be present throughout the image. We have focused on this issue and derived conservative (but tight) bounds of this error, which makes our implementation more robust to this issue than previous rendering systems.
- Participating media representation. We have significantly improved the way that scattering media are described and represented in the system; this allows for more accurate results with nested scattering media. A new sampling technique enables unbiased rendering of heterogeneous media in a way that cleanly integrates with all of the other parts of the system.
- Measured materials. This edition includes a new technique to represent and evaluate measured materials using a sparse frequency-space basis. This approach is convenient because it allows for exact importance sampling, which was not possible with the representation used in the previous edition.
- Photon mapping. A significant step forward for photon mapping algorithms has been the development of variants that don't require storing all of the photons in memory. We have replaced `pbrt`'s photon mapping algorithm with an implementation based on stochastic progressive photon mapping, which efficiently renders many difficult light transport effects.
- Sample generation algorithms. The distribution of sample values used for numerical integration in rendering algorithms can have a surprisingly large effect on the quality of the final results. We have thoroughly updated our treatment of this topic, covering new approaches and efficient implementation techniques in more depth than before.

Many other parts of the system have been improved and updated to reflect progress in the field: microfacet reflection models are treated in more depth, with much better sampling techniques; a new “curve” shape has been added for modeling hair and other fine geometry; and a new camera model that simulates realistic lens systems is now available. Throughout the book, we have made numerous smaller changes to more clearly explain and illustrate the key concepts in physically based rendering systems like `pbrt`.

## ACKNOWLEDGMENTS

Pat Hanrahan has contributed to this book in more ways than we could hope to acknowledge; we owe a profound debt to him. He tirelessly argued for clean interfaces and finding the right abstractions to use throughout the system, and his understanding of and approach to rendering deeply influenced its design. His willingness to use `pbrt` and this manuscript in his rendering course at Stanford was enormously helpful, particularly in the early years of its life when it was still in very rough form; his feedback throughout this process has been crucial for bringing the text to its current state. Finally, the group of people that Pat helped assemble at the Stanford Graphics Lab, and the open environment that he fostered, made for an exciting, stimulating, and fertile environment. Matt and Greg both feel extremely privileged to have been there.

We owe a debt of gratitude to the many students who used early drafts of this book in courses at Stanford and the University of Virginia between 1999 and 2004. These students provided an enormous amount of feedback about the book and `pbrt`. The teaching assistants for these courses deserve special mention: Tim Purcell, Mike Cammarano, Ian Buck, and Ren Ng at Stanford, and Nolan Goodnight at Virginia. A number of students in those classes gave particularly valuable feedback and sent bug reports and bug fixes; we would especially like to thank Evan Parker and Phil Beatty. A draft of the manuscript of this book was used in classes taught by Bill Mark and Don Fussell at the University of Texas, Austin, and Raghu Machiraju at Ohio State University; their feedback was invaluable, and we are grateful for their adventurousness in incorporating this system into their courses, even while it was still being edited and revised.

Matt Pharr would like to acknowledge colleagues and co-workers in rendering-related endeavors who have been a great source of education and who have substantially influenced his approach to writing renderers and his understanding of the field. Particular thanks go to Craig Kolb, who provided a cornerstone of Matt’s early computer graphics education through the freely available source code to the `rayshade` ray-tracing system, and Eric Veach, who has also been generous with his time and expertise. Thanks also to Doug Shult and Stan Eisenstat for formative lessons in mathematics and computer science during high school and college, respectively, and most important to Matt’s parents, for the education they’ve provided and continued encouragement along the way. Finally, thanks also to Nick Triantos, Jayant Kolhe, and NVIDIA for their understanding and support through the final stages of the preparation of the first edition of the book.

Greg Humphreys is very grateful to all the professors and TAs who tolerated him when he was an undergraduate at Princeton. Many people encouraged his interest in graphics, specifically Michael Cohen, David Dobkin, Adam Finkelstein, Michael Cox, Gordon Stoll, Patrick Min, and Dan Wallach. Doug Clark, Steve Lyon, and Andy Wolfe also supervised various independent research boondoggles without even laughing once. Once, in a group meeting about a year-long robotics project, Steve Lyon became exasperated and yelled, “Stop telling me why it can’t be done, and figure out how to do it!”—an impromptu lesson that will never be forgotten. Eric Ristad fired Greg as a summer research assistant after his freshman year (before the summer even began), pawning him off on an unsuspecting Pat Hanrahan and beginning an advising relationship that would span 10 years and both coasts. Finally, Dave Hanson taught Greg that literate programming was

a great way to work and that computer programming can be a beautiful and subtle art form.

Wenzel Jakob was excited when the first edition of *pbrt* arrived in his mail during his undergraduate studies in 2004. Needless to say, this had a lasting effect on his career—thus Wenzel would like to begin by thanking his co-authors for inviting him to become a part of third edition of this book. Wenzel is extremely indebted to Steve Marschner, who was his PhD advisor during a fulfilling five years at Cornell University. Steve brought him into the world of research and remains a continuous source of inspiration. Wenzel is also thankful for the guidance and stimulating research environment created by the other members of the graphics group, including Kavita Bala, Doug James, and Bruce Walter. Wenzel spent a wonderful postdoc with Olga Sorkine Hornung who introduced him to geometry processing. Olga’s support for Wenzel’s involvement in this book is deeply appreciated.

For the first edition, we are also grateful to Don Mitchell, for his help with understanding some of the details of sampling and reconstruction; Thomas Kollig and Alexander Keller, for explaining the finer points of low-discrepancy sampling; and Christer Ericson, who had a number of suggestions for improving our kd-tree implementation. For the second edition, we’re thankful to Christophe Hery and Eugene d’Eon for helping us with the nuances of subsurface scattering.

For the third edition, we’d especially like to thank Leo Grünschloß for reviewing our sampling chapter; Alexander Keller for suggestions about topics for that chapter; Eric Heitz for extensive help with microfacets (and reviewing our text on that topic); Thiago Ize for thoroughly reviewing the text on floating-point error; Tom van Bussel for reporting a number of errors in our BSSRDF code; Ralf Habel for reviewing our BSSRDF text; and Toshiya Hachisuka and Anton Kaplanyan for extensive review and comments about our light transport chapters. Discussions with Eric Veach about floating-point round-off error and ray tracing were extremely helpful to our development of our approach to that topic. We’d also like to thank Per Christensen, Doug Epps, Luca Fascione, Marcos Fajardo, Christophe Hery, John “Spike” Hughes, Andrew Kensler, Alan King, Chris Kulla, Morgan McGuire, Andy Selle, and Ingo Wald for helpful discussions, suggestions, and pointers to research.

We would also like to thank the book’s reviewers, all of whom had insightful and constructive feedback about the manuscript at various stages of its progress. We’d particularly like to thank the reviewers who provided feedback on both the first and second editions of the book: Ian Ashdown, Per Christensen, Doug Epps, Dan Goldman, Eric Haines, Erik Reinhard, Pete Shirley, Peter-Pike Sloan, Greg Ward, and a host of anonymous reviewers. For the second edition, Janne Kontkanen, Nelson Max, Bill Mark, and Eric Tabellion also contributed numerous helpful suggestions.

Many people have contributed to not only *pbrt* but to our own better understanding of rendering through bug reports, patches, and suggestions about better implementation approaches. A few have made particularly substantial contributions over the years—we would especially like to thank Solomon Boulos, Stephen Chenney, John Danks, Kevin Egan, Volodymyr Kachurovskyi, and Ke Xu.

In addition, we would like to thank Rachit Agrawal, Frederick Akalin, Mark Bolstad, Thomas de Bodt, Brian Budge, Mark Colbert, Yunjian Ding, Tao Du, Shaohua Fan, Etienne Ferrier, Nigel Fisher, Jeppe Revall Frisvad, Robert G. Graf, Asbjørn Heid, Keith Jeffery, Greg Johnson, Aaron Karp, Donald Knuth, Martin Kraus, Murat Kurt, Larry Lai, Craig McNaughton, Swaminathan Narayanan, Anders Nilsson, Jens Olsson, Vincent Pegoraro, Srinath Ravichandiran, Sébastien Speierer, Nils Thuerey, Xiong Wei, Wei-Wei Xu, Arek Zimny, and Matthias Zwicker for their suggestions and bug reports. Finally, we would like to thank the *LuxRender* developers and the *LuxRender* community, particularly Terrence Vergauwen, Jean-Philippe Grimaldi, and Asbjørn Heid; it has been a delight to see the rendering system they have built from pbrt's foundation, and we have learned from reading their source code and implementations of new rendering algorithms.

Special thanks to Martin Preston and Steph Bruning from Framestore for their help with our being able to use a frame from *Gravity* (image courtesy of Warner Bros. and Framestore), and to Joe Letteri, Dave Gouge, and Luca Fascione from Weta Digital for their help with the frame from *The Hobbit: The Battle of the Five Armies* (© 2014 Warner Bros. Entertainment Inc. and Metro-Goldwyn-Mayer Pictures Inc. (US, Canada & New Line Foreign Territories), © 2014 Metro-Goldwyn-Mayer Pictures Inc. and Warner Bros. Entertainment Inc. (all other territories). All Rights Reserved).

## PRODUCTION

For the production of the first edition, we would also like to thank Tim Cox (senior editor), for his willingness to take on this slightly unorthodox project and for both his direction and patience throughout the process. We are very grateful to Elisabeth Beller (project manager), who has gone well beyond the call of duty for this book; her ability to keep this complex project in control and on schedule has been remarkable, and we particularly thank her for the measurable impact she has had on the quality of the final result. Thanks also to Rick Camp (editorial assistant) for his many contributions along the way. Paul Anagnostopoulos and Jacqui Scarlott at Windfall Software did the book's composition; their ability to take the authors' homebrew literate programming file format and turn it into high-quality final output while also juggling the multiple unusual types of indexing we asked for is greatly appreciated. Thanks also to Ken DellaPenta (copyeditor) and Jennifer McClain (proofreader) as well as to Max Spector at Chen Design (text and cover designer), and Steve Rath (indexer).

For the second edition, we'd like to thank Greg Chalson who talked us into expanding and updating the book; Greg also ensured that Paul Anagnostopoulos at Windfall Software would again do the book's composition. We'd like to thank Paul again for his efforts in working with this book's production complexity. Finally, we'd also like to thank Todd Green, Paul Gottehrer, and Heather Scherer at Elsevier.

For the third edition, we'd like to thank Todd Green from Elsevier, who oversaw this go-round, and Amy Invernizzi, who kept the train on the rails throughout the process. We were delighted to have Paul Anagnostopoulos at Windfall Software part of this process for a third time; his efforts have been critical to the book's high production value, which is so important to us.

## SCENES AND MODELS

Many people and organizations have generously supplied us with scenes and models for use in this book and the *pbrt* distribution. Their generosity has been invaluable in helping us create interesting example images throughout the text.

The bunny, Buddha, and dragon models are courtesy of the Stanford Computer Graphics Laboratory’s scanning repository. The “killeroo” model is included with permission of Phil Dench and Martin Rezard (3D scan and digital representations by headus, design and clay sculpt by Rezard). The dragon model scan used in Chapters 8 and 9 is courtesy of Christian Schüller, and thanks to Yasutoshi Mori for the sports car used in Chapters 7 and 12. The glass used to illustrate caustics in Figures 16.9 and 16.11 is thanks to Simon Wendsche, and the physically accurate smoke data sets were created by Duc Nguyen and Ron Fedkiw.

The head model used to illustrate subsurface scattering was made available by Infinite Realities, Inc. under a Creative Commons Attribution 3.0 license. Thanks to “Wig42” for the breakfast table scene used in Figure 16.8 and “guismo” for the coffee splash scene used in Figure 15.5; both were posted to *blendswap.com* also under a Creative Commons Attribution 3.0 license.

Nolan Goodnight created environment maps with a realistic skylight model, and Paul Debevec provided numerous high dynamic-range environment maps. Thanks also to Bernhard Vogl (*dativ.at/lightprobes/*) for environment maps that we used in numerous figures. Marc Ellens provided spectral data for a variety of light sources, and the spectral RGB measurement data for a variety of displays is courtesy of Tom Lianza at X-Rite.

We are most particularly grateful to Guillermo M. Leal Llaguno of Evolución Visual, *www.evvisual.com*, who modeled and rendered the San Miguel scene that was featured on the cover of the second edition and is still used in numerous figures in the book. We would also especially like to thank Marko Dabrovic (*www.3lhd.com*) and Mihovil Odak at RNA Studios (*www.rna.hr*), who supplied a bounty of excellent models and scenes, including the Sponza atrium, the Sibenik cathedral, and the Audi TT car model. Many thanks are also due to Florent Boyer (*www.florentboyer.com*), who provided the contemporary house scene used in some of the images in Chapter 16.

## ABOUT THE COVER

The “Countryside” scene on the cover of the book was created by Jan-Walter Schliep, Burak Kahraman, and Timm Dapper of Laubwerk (*www.laubwerk.com*). The scene features 23,241 individual plants, with a total of 3.1 billion triangles. (Thanks to object instancing, only 24 million triangles need to be stored in memory.) The *pbrt* files that describe the scene geometry require 1.1 GB of on-disk storage. There are a total of 192 texture maps, representing 528 MB of texture data. The scene is one of the example scenes that are available from the *pbrt* Web site.

## ADDITIONAL READING

Donald Knuth's article *Literate Programming* (Knuth 1984) describes the main ideas behind literate programming as well as his web programming environment. The seminal TeX typesetting system was written with web and has been published as a series of books (Knuth 1986; Knuth 1993a). More recently, Knuth has published a collection of graph algorithms in literate format in *The Stanford GraphBase* (Knuth 1993b). These programs are enjoyable to read and are excellent presentations of their respective algorithms. The Web site [www.literateprogramming.com](http://www.literateprogramming.com) has pointers to many articles about literate programming, literate programs to download, and a variety of literate programming systems; many refinements have been made since Knuth's original development of the idea.

The only other literate programs we know of that have been published as books are the implementation of the 1cc compiler, which was written by Christopher Fraser and David Hanson and published as *A Retargetable C Compiler: Design and Implementation* (Fraser and Hanson 1995), and Martin Ruckert's book on the mp3 audio format, *Understanding MP3* (Ruckert 2005).



# CHAPTER ONE

# 01 INTRODUCTION

Rendering is the process of producing an image from the description of a 3D scene. Obviously, this is a very broad task, and there are many ways to approach it. *Physically based* techniques attempt to simulate reality; that is, they use principles of physics to model the interaction of light and matter. While a physically based approach may seem to be the most obvious way to approach rendering, it has only been widely adopted in practice over the past 10 or so years. Section 1.7 at the end of this chapter gives a brief history of physically based rendering and its recent adoption for offline rendering for movies and for interactive rendering for games.

This book describes pbrt, a physically based rendering system based on the ray-tracing algorithm. Most computer graphics books present algorithms and theory, sometimes combined with snippets of code. In contrast, this book couples the theory with a complete implementation of a fully functional rendering system. The source code to the system (as well as example scenes and a collection of data for rendering) can be found on the pbrt Web site, [pbrt.org](http://pbrt.org).

## 1.1 LITERATE PROGRAMMING

While writing the TeX typesetting system, Donald Knuth developed a new programming methodology based on the simple but revolutionary idea that *programs should be written more for people's consumption than for computers' consumption*. He named this methodology *literate programming*. This book (including the chapter you're reading now) is a long literate program. This means that in the course of reading this book, you will read the *full* implementation of the pbrt rendering system, not just a high-level description of it.

Literate programs are written in a metalanguage that mixes a document formatting language (e.g., TeX or HTML) and a programming language (e.g., C++). Two separate systems process the program: a “weaver” that transforms the literate program into a

document suitable for typesetting and a “tangler” that produces source code suitable for compilation. Our literate programming system is homegrown, but it was heavily influenced by Norman Ramsey’s noweb system.

The literate programming metalanguage provides two important features. The first is the ability to mix prose with source code. This feature makes the description of the program just as important as its actual source code, encouraging careful design and documentation. Second, the language provides mechanisms for presenting the program code to the reader in an order that is entirely different from the compiler input. Thus, the program can be described in a logical manner. Each named block of code is called a *fragment*, and each fragment can refer to other fragments by name.

As a simple example, consider a function `InitGlobals()` that is responsible for initializing all of a program’s global variables:<sup>1</sup>

```
void InitGlobals() {
    nMarbles = 25.7;
    shoeSize = 13;
    dielectric = true;
}
```

Despite its brevity, this function is hard to understand without any context. Why, for example, can the variable `nMarbles` take on floating-point values? Just looking at the code, one would need to search through the entire program to see where each variable is declared and how it is used in order to understand its purpose and the meanings of its legal values. Although this structuring of the system is fine for a compiler, a human reader would much rather see the initialization code for each variable presented separately, near the code that actually declares and uses the variable.

In a literate program, one can instead write `InitGlobals()` like this:

```
(Function Definitions) ≡
void InitGlobals() {
    (Initialize Global Variables 2)
}
```

This defines a fragment, called *(Function Definitions)*, that contains the definition of the `InitGlobals()` function. The `InitGlobals()` function itself refers to another fragment, *(Initialize Global Variables)*. Because the initialization fragment has not yet been defined, we don’t know anything about this function except that it will probably contain assignments to global variables. This is just the right level of abstraction for now, since no variables have been declared yet. When we introduce the global variable `shoeSize` somewhere later in the program, we can then write

```
(Initialize Global Variables) ≡
shoeSize = 13;
```

2

---

<sup>1</sup> The example code in this section is merely illustrative and is not part of pbrt itself.

Here we have started to define the contents of *(Initialize Global Variables)*. When the literate program is tangled into source code for compilation, the literate programming system will substitute the code `shoeSize = 13;` inside the definition of the `InitGlobals()` function. Later in the text, we may define another global variable, `dielectric`, and we can append its initialization to the fragment:

```
(Initialize Global Variables) +≡  
    dielectric = true;
```

2

The `+≡` symbol after the fragment name shows that we have added to a previously defined fragment. When tangled, the result of these three fragments is the code

```
void InitGlobals() {  
    shoeSize = 13;  
    dielectric = true;  
}
```

In this way, we can decompose complex functions into logically distinct parts, making them much easier to understand. For example, we can write a complicated function as a series of fragments:

```
(Function Definitions) +≡  
void complexFunc(int x, int y, double *values) {  
    (Check validity of arguments)  
    if (x < y) {  
        (Swap parameter values)  
    }  
    (Do precomputation before loop)  
    (Loop through and update values array)  
}
```

Again, the contents of each fragment are expanded inline in `complexFunc()` for compilation. In the document, we can introduce each fragment and its implementation in turn. This decomposition lets us present code a few lines at a time, making it easier to understand. Another advantage of this style of programming is that by separating the function into logical fragments, each with a single and well-delineated purpose, each one can then be written, verified, or read independently. In general, we will try to make each fragment less than 10 lines long.

In some sense, the literate programming system is just an enhanced macro substitution package tuned to the task of rearranging program source code. This may seem like a trivial change, but in fact literate programming is quite different from other ways of structuring software systems.

### 1.1.1 INDEXING AND CROSS-REFERENCING

The following features are designed to make the text easier to navigate. Indices in the page margins give page numbers where the functions, variables, and methods used on that page are defined. Indices at the end of the book collect all of these identifiers so that it's possible to find definitions by name. Appendix C, "Index of Fragments," lists the pages where each fragment is defined and the pages where it is used. Within the text, a defined

fragment name is followed by a list of page numbers on which that fragment is used. For example, a hypothetical fragment definition such as

<i>(A fascinating fragment)</i> ≡	184, 690
nMarbles += .001;	

indicates that this fragment is used on pages 184 and 690. Occasionally we elide fragments from the printed book that are either boilerplate code or substantially the same as other fragments; when these fragments are used, no page numbers will be listed.

When a fragment is used inside another fragment, the page number on which it is first defined appears after the fragment name. For example,

<i>(Do something interesting)</i> + ≡	500
InitializeSomethingInteresting();	
<i>(Do something else interesting 486)</i>	
CleanUp();	

indicates that the *(Do something else interesting)* fragment is defined on page 486. If the definition of the fragment is not included in the book, no page number will be listed.

## 1.2 PHOTOREALISTIC RENDERING AND THE RAY-TRACING ALGORITHM

The goal of photorealistic rendering is to create an image of a 3D scene that is indistinguishable from a photograph of the same scene. Before we describe the rendering process, it is important to understand that in this context the word *indistinguishable* is imprecise because it involves a human observer, and different observers may perceive the same image differently. Although we will cover a few perceptual issues in this book, accounting for the precise characteristics of a given observer is a very difficult and largely unsolved problem. For the most part, we will be satisfied with an accurate simulation of the physics of light and its interaction with matter, relying on our understanding of display technology to present the best possible image to the viewer.

Almost all photorealistic rendering systems are based on the ray-tracing algorithm. Ray tracing is actually a very simple algorithm; it is based on following the path of a ray of light through a scene as it interacts with and bounces off objects in an environment. Although there are many ways to write a ray tracer, all such systems simulate at least the following objects and phenomena:

- *Cameras*: A camera model determines how and from where the scene is being viewed, including how an image of the scene is recorded on a sensor. Many rendering systems generate viewing rays starting at the camera that are then traced into the scene.
- *Ray-object intersections*: We must be able to tell precisely where a given ray intersects a given geometric object. In addition, we need to determine certain properties of the object at the intersection point, such as a surface normal or its material. Most ray tracers also have some facility for testing the intersection of a ray with multiple objects, typically returning the closest intersection along the ray.

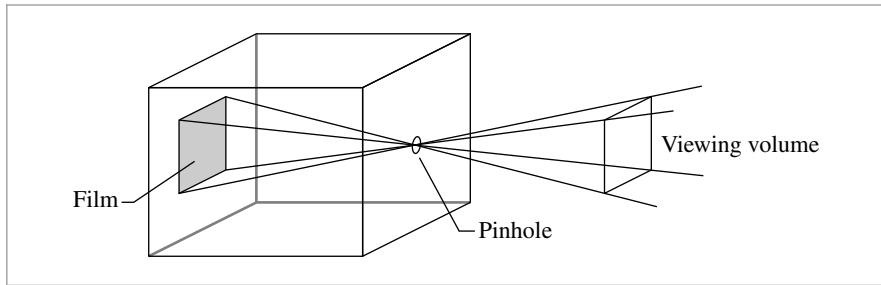
- *Light sources*: Without lighting, there would be little point in rendering a scene. A ray tracer must model the distribution of light throughout the scene, including not only the locations of the lights themselves but also the way in which they distribute their energy throughout space.
- *Visibility*: In order to know whether a given light deposits energy at a point on a surface, we must know whether there is an uninterrupted path from the point to the light source. Fortunately, this question is easy to answer in a ray tracer, since we can just construct the ray from the surface to the light, find the closest ray-object intersection, and compare the intersection distance to the light distance.
- *Surface scattering*: Each object must provide a description of its appearance, including information about how light interacts with the object’s surface, as well as the nature of the reradiated (or *scattered*) light. Models for surface scattering are typically parameterized so that they can simulate a variety of appearances.
- *Indirect light transport*: Because light can arrive at a surface after bouncing off or passing through other surfaces, it is usually necessary to trace additional rays originating at the surface to fully capture this effect.
- *Ray propagation*: We need to know what happens to the light traveling along a ray as it passes through space. If we are rendering a scene in a vacuum, light energy remains constant along a ray. Although true vacuums are unusual on Earth, they are a reasonable approximation for many environments. More sophisticated models are available for tracing rays through fog, smoke, the Earth’s atmosphere, and so on.

We will briefly discuss each of these simulation tasks in this section. In the next section, we will show pbrt’s high-level interface to the underlying simulation components and follow the progress of a single ray through the main rendering loop. We will also present the implementation of a surface scattering model based on Turner Whitted’s original ray-tracing algorithm.

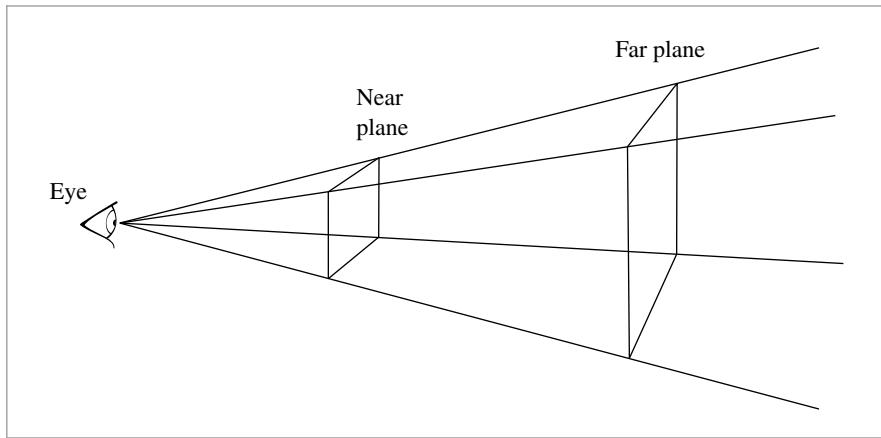
### 1.2.1 CAMERAS

Nearly everyone has used a camera and is familiar with its basic functionality: you indicate your desire to record an image of the world (usually by pressing a button or tapping a screen), and the image is recorded onto a piece of film or an electronic sensor. One of the simplest devices for taking photographs is called the *pinhole camera*. Pinhole cameras consist of a light-tight box with a tiny hole at one end (Figure 1.1). When the hole is uncovered, light enters this hole and falls on a piece of photographic paper that is affixed to the other end of the box. Despite its simplicity, this kind of camera is still used today, frequently for artistic purposes. Very long exposure times are necessary to get enough light on the film to form an image.

Although most cameras are substantially more complex than the pinhole camera, it is a convenient starting point for simulation. The most important function of the camera is to define the portion of the scene that will be recorded onto the film. In Figure 1.1, we can see how connecting the pinhole to the edges of the film creates a double pyramid that extends into the scene. Objects that are not inside this pyramid cannot be imaged onto the film. Because actual cameras image a more complex shape than a pyramid, we will refer to the region of space that can potentially be imaged onto the film as the *viewing volume*.



**Figure 1.1:** A Pinhole Camera.



**Figure 1.2:** When we simulate a pinhole camera, we place the film in front of the hole at the near plane, and the hole is renamed the *eye*.

Another way to think about the pinhole camera is to place the film plane in *front* of the pinhole but at the same distance (Figure 1.2). Note that connecting the hole to the film defines exactly the same viewing volume as before. Of course, this is not a practical way to build a real camera, but for simulation purposes it is a convenient abstraction. When the film (or image) plane is in front of the pinhole, the pinhole is frequently referred to as the *eye*.

Now we come to the crucial issue in rendering: at each point in the image, what color value does the camera record? If we recall the original pinhole camera, it is clear that only light rays that travel along the vector between the pinhole and a point on the film can contribute to that film location. In our simulated camera with the film plane in front of the eye, we are interested in the amount of light traveling from the image point to the eye.

Therefore, an important task of the camera simulator is to take a point on the image and generate *rays* along which incident light will contribute to that image location. Because

a ray consists of an origin point and a direction vector, this task is particularly simple for the pinhole camera model of Figure 1.2: it uses the pinhole for the origin and the vector from the pinhole to the near plane as the ray’s direction. For more complex camera models involving multiple lenses, the calculation of the ray that corresponds to a given point on the image may be more involved. (Section 6.4 describes the implementation of such a model.)

With the process of converting image locations to rays completely encapsulated in the camera module, the rest of the rendering system can focus on evaluating the lighting along those rays, and a variety of camera models can be supported. pbrt’s camera abstraction is described in detail in Chapter 6.

### 1.2.2 RAY-OBJECT INTERSECTIONS

Each time the camera generates a ray, the first task of the renderer is to determine which object, if any, that ray intersects first and where the intersection occurs. This intersection point is the visible point along the ray, and we will want to simulate the interaction of light with the object at this point. To find the intersection, we must test the ray for intersection against all objects in the scene and select the one that the ray intersects first. Given a ray  $r$ , we first start by writing it in *parametric form*:

$$r(t) = o + t\mathbf{d},$$

where  $o$  is the ray’s origin,  $\mathbf{d}$  is its direction vector, and  $t$  is a parameter whose legal range is  $(0, \infty)$ . We can obtain a point along the ray by specifying its parametric  $t$  value and evaluating the above equation.

It is often easy to find the intersection between the ray  $r$  and a surface defined by an implicit function  $F(x, y, z) = 0$ . We first substitute the ray equation into the implicit equation, producing a new function whose only parameter is  $t$ . We then solve this function for  $t$  and substitute the smallest positive root into the ray equation to find the desired point. For example, the implicit equation of a sphere centered at the origin with radius  $r$  is

$$x^2 + y^2 + z^2 - r^2 = 0.$$

Substituting the ray equation, we have

$$(o_x + t\mathbf{d}_x)^2 + (o_y + t\mathbf{d}_y)^2 + (o_z + t\mathbf{d}_z)^2 - r^2 = 0.$$

All of the values besides  $t$  are known, giving us an easily solved quadratic equation in  $t$ . If there are no real roots, the ray misses the sphere; if there are roots, the smallest positive one gives the intersection point.

The intersection point alone is not enough information for the rest of the ray tracer; it needs to know certain properties of the surface at the point. First, a representation of the material at the point must be determined and passed along to later stages of the ray-tracing algorithm. Second, additional geometric information about the intersection point will also be required in order to shade the point. For example, the surface normal  $\mathbf{n}$  is always required. Although many ray tracers operate with only  $\mathbf{n}$ , more sophisticated rendering systems like pbrt require even more information, such as various partial

derivatives of position and surface normal with respect to the local parameterization of the surface.

Of course, most scenes are made up of multiple objects. The brute-force approach would be to test the ray against each object in turn, choosing the minimum positive  $t$  value of all intersections to find the closest intersection. This approach, while correct, is very slow, even for scenes of modest complexity. A better approach is to incorporate an *acceleration structure* that quickly rejects whole groups of objects during the ray intersection process. This ability to quickly cull irrelevant geometry means that ray tracing frequently runs in  $O(I \log N)$  time, where  $I$  is the number of pixels in the image and  $N$  is the number of objects in the scene.<sup>2</sup> (Building the acceleration structure itself is necessarily at least  $O(N)$  time, however.)

pbrt's geometric interface and implementations of it for a variety of shapes is described in Chapter 3, and the acceleration interface and implementations are shown in Chapter 4.

### 1.2.3 LIGHT DISTRIBUTION

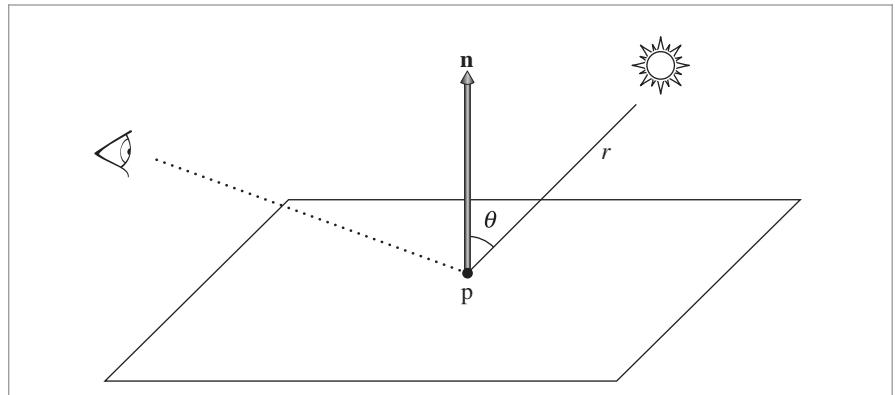
The ray–object intersection stage gives us a point to be shaded and some information about the local geometry at that point. Recall that our eventual goal is to find the amount of light leaving this point in the direction of the camera. In order to do this, we need to know how much light is *arriving* at this point. This involves both the *geometric* and *radiometric* distribution of light in the scene. For very simple light sources (e.g., point lights), the geometric distribution of lighting is a simple matter of knowing the position of the lights. However, point lights do not exist in the real world, and so physically based lighting is often based on *area* light sources. This means that the light source is associated with a geometric object that emits illumination from its surface. However, we will use point lights in this section to illustrate the components of light distribution; rigorous discussion of light measurement and distribution is the topic of Chapters 5 and 12.

We frequently would like to know the amount of light power being deposited on the differential area surrounding the intersection point (Figure 1.3). We will assume that the point light source has some power  $\Phi$  associated with it and that it radiates light equally in all directions. This means that the power per area on a unit sphere surrounding the light is  $\Phi/(4\pi)$ . (These measurements will be explained and formalized in Section 5.4.)

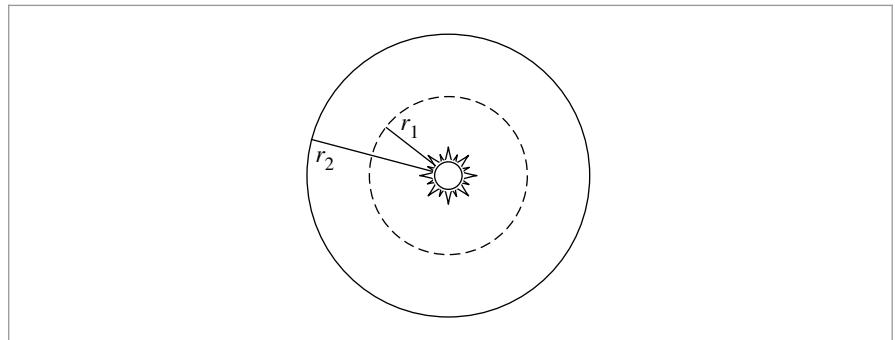
If we consider two such spheres (Figure 1.4), it is clear that the power per area at a point on the larger sphere must be less than the power at a point on the smaller sphere because the same total power is distributed over a larger area. Specifically, the power per area arriving at a point on a sphere of radius  $r$  is proportional to  $1/r^2$ . Furthermore, it can be shown that if the tiny surface patch  $dA$  is tilted by an angle  $\theta$  away from the vector from the surface point to the light, the amount of power deposited on  $dA$  is proportional

---

<sup>2</sup> Although ray tracing's logarithmic complexity is often heralded as one of its key strengths, this complexity is typically only true on average. A number of ray-tracing algorithms that have guaranteed logarithmic running time have been published in the computational geometry literature, but these algorithms only work for certain types of scenes and have very expensive preprocessing and storage requirements. Szirmay-Kalos and Márton provide pointers to the relevant literature (Szirmay-Kalos and Márton 1998). One consolation is that scenes representing realistic environments generally don't exhibit this worst-case behavior. In practice, the ray intersection algorithms presented in this book are sublinear, but without expensive preprocessing and huge memory usage it is always possible to construct worst-case scenes where ray tracing runs in  $O(IN)$  time.



**Figure 1.3:** Geometric construction for determining the power per area arriving at a point due to a point light source. The distance from the point to the light source is denoted by  $r$ .



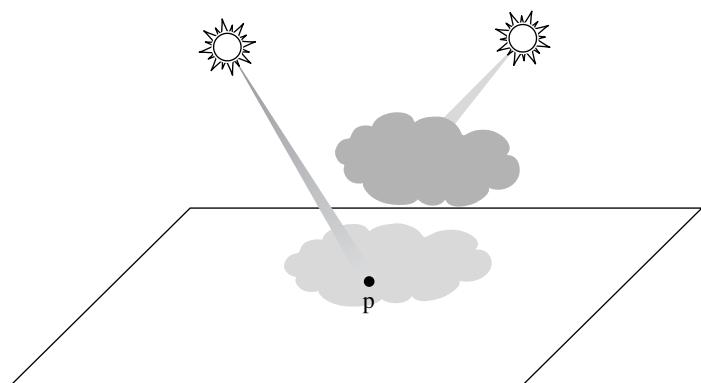
**Figure 1.4:** Since the point light radiates light equally in all directions, the same total power is deposited on all spheres centered at the light.

to  $\cos \theta$ . Putting this all together, the differential power per area  $dE$  (the *differential irradiance*) is

$$dE = \frac{\Phi \cos \theta}{4\pi r^2}.$$

Readers already familiar with basic lighting in computer graphics will notice two familiar laws encoded in this equation: the cosine falloff of light for tilted surfaces mentioned above, and the one-over- $r$ -squared falloff of light with distance.

Scenes with multiple lights are easily handled because illumination is *linear*: the contribution of each light can be computed separately and summed to obtain the overall contribution.



**Figure 1.5:** A light source only deposits energy on a surface if the source is not obscured as seen from the receiving point. The light source on the left illuminates the point  $p$ , but the light source on the right does not.

## 1.2.4 VISIBILITY

The lighting distribution described in the previous section ignores one very important component: *shadows*. Each light contributes illumination to the point being shaded only if the path from the point to the light’s position is unobstructed (Figure 1.5).

Fortunately, in a ray tracer it is easy to determine if the light is visible from the point being shaded. We simply construct a new ray whose origin is at the surface point and whose direction points toward the light. These special rays are called *shadow rays*. If we trace this ray through the environment, we can check to see whether any intersections are found between the ray’s origin and the light source by comparing the parametric  $t$  value of any intersections found to the parametric  $t$  value along the ray of the light source position. If there is no blocking object between the light and the surface, the light’s contribution is included.

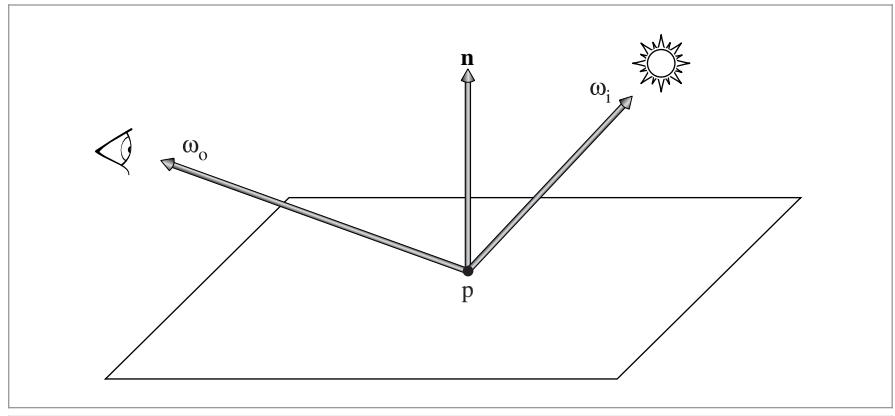
## 1.2.5 SURFACE SCATTERING

We now are able to compute two pieces of information that are vital for proper shading of a point: its location and the incident lighting.<sup>3</sup> Now we need to determine how the incident lighting is *scattered* at the surface. Specifically, we are interested in the amount of light energy scattered back along the ray that we originally traced to find the intersection point, since that ray leads to the camera (Figure 1.6).

Each object in the scene provides a *material*, which is a description of its appearance properties at each point on the surface. This description is given by the *bidirectional reflectance distribution function* (BRDF). This function tells us how much energy is reflected

---

<sup>3</sup> Readers already familiar with rendering might object that the discussion in this section considers only direct lighting. Rest assured that pbrt does support global illumination.



**Figure 1.6: The Geometry of Surface Scattering.** Incident light arriving along direction  $\omega_i$  interacts with the surface at point  $p$  and is scattered back toward the camera along direction  $\omega_o$ . The amount of light scattered toward the camera is given by the product of the incident light energy and the BRDF.

from an incoming direction  $\omega_i$  to an outgoing direction  $\omega_o$ . We will write the BRDF at  $p$  as  $f_r(p, \omega_o, \omega_i)$ . Now, computing the amount of light  $L$  scattered back toward the camera is straightforward:

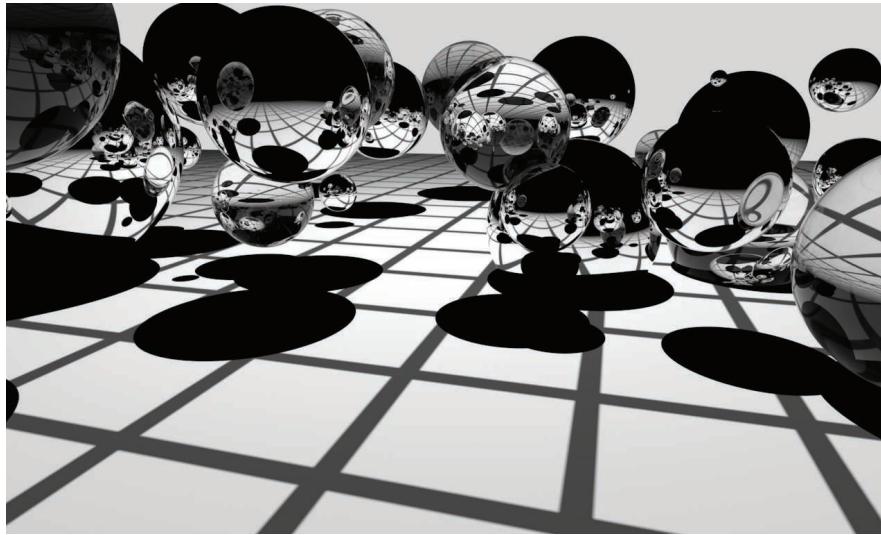
```
for each light:
    if light is not blocked:
        incident_light = light.L(point)
        amount_reflected =
            surface.BRDF(hit_point, camera_vector, light_vector)
        L += amount_reflected * incident_light
```

Here we are using the symbol  $L$  to represent the light; this represents a slightly different unit for light measurement than  $dE$ , which was used before.  $L$  represents *radiance*, a unit for measuring light that we will see much of in the following.

It is easy to generalize the notion of a BRDF to transmitted light (obtaining a BTDF) or to general scattering of light arriving from either side of the surface. A function that describes general scattering is called a *bidirectional scattering distribution function* (BSDF). pbrt supports a variety of BSDF models; they are described in Chapter 8. More complex yet is the *bidirectional subsurface scattering reflectance distribution function* (BSSRDF), which models light that exits a surface at a different point than it enters. The BSSRDF is described in Sections 5.6.2, 11.4, and 15.5.

### 1.2.6 INDIRECT LIGHT TRANSPORT

Turner Whitted's original paper on ray tracing (1980) emphasized its *recursive* nature, which was the key that made it possible to include indirect specular reflection and transmission in rendered images. For example, if a ray from the camera hits a shiny object like a mirror, we can reflect the ray about the surface normal at the intersection point and recursively invoke the ray-tracing routine to find the light arriving at the point on the



**Figure 1.7: A Prototypical Example of Early Ray Tracing.** Note the use of mirrored and glass objects, which emphasizes the algorithm’s ability to handle these kinds of surfaces.

mirror, adding its contribution to the original camera ray. This same technique can be used to trace transmitted rays that intersect transparent objects. For a long time, most early ray-tracing examples showcased mirrors and glass balls (Figure 1.7) because these types of effects were difficult to capture with other rendering techniques.

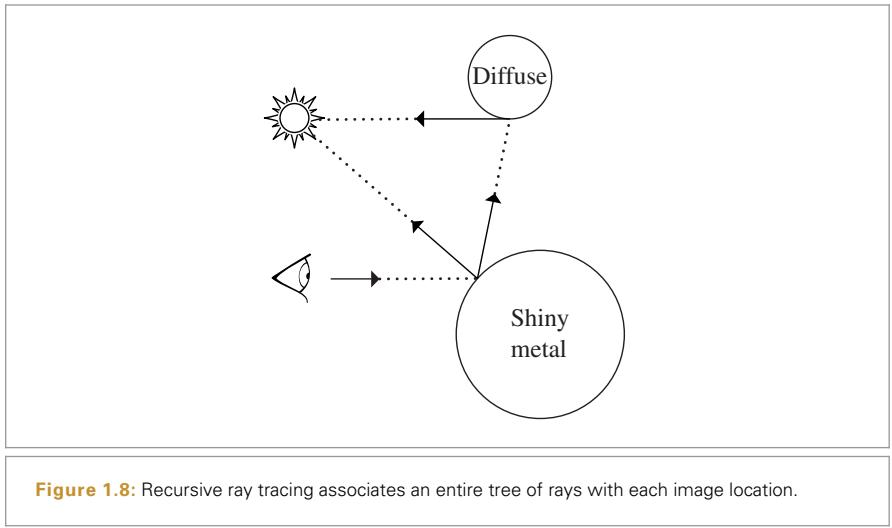
In general, the amount of light that reaches the camera from a point on an object is given by the sum of light emitted by the object (if it is itself a light source) and the amount of reflected light. This idea is formalized by the *light transport equation* (also often known as the *rendering equation*), which says that the outgoing radiance  $L_o(p, \omega_o)$  from a point  $p$  in direction  $\omega_o$  is the emitted radiance at that point in that direction,  $L_e(p, \omega_o)$ , plus the incident radiance from all directions on the sphere  $S^2$  around  $p$  scaled by the BSDF  $f(p, \omega_o, \omega_i)$  and a cosine term:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{S^2} f(p, \omega_o, \omega_i) L_i(p, \omega_i) |\cos \theta_i| d\omega_i. \quad (1.1)$$

We will show a more complete derivation of this equation in Sections 5.6.1 and 14.4. Solving this integral analytically is not possible except for the simplest of scenes, so we must either make simplifying assumptions or use numerical integration techniques.

Whitted’s algorithm simplifies this integral by ignoring incoming light from most directions and only evaluating  $L_i(p, \omega_i)$  for directions to light sources and for the directions of perfect reflection and refraction. In other words, it turns the integral into a sum over a small number of directions.

Whitted’s method can be extended to capture more effects than just perfect mirrors and glass. For example, by tracing many recursive rays near the mirror-reflection direction and averaging their contributions, we obtain an approximation of glossy reflection. In



fact, we can *always* recursively trace a ray whenever we hit an object. For example, we can randomly choose a reflection direction  $\omega_i$  and weight the contribution of this newly spawned ray by evaluating the BRDF  $f_r(p, \omega_o, \omega_i)$ . This simple but powerful idea can lead to very realistic images because it captures all of the interreflection of light between objects. Of course, we need to know when to terminate the recursion, and choosing directions completely at random may make the rendering algorithm slow to converge to a reasonable result. These problems can be addressed, however; these issues are the topics of Chapters 13 through 16.

When we trace rays recursively in this manner, we are really associating a *tree* of rays with each image location (Figure 1.8), with the ray from the camera at the root of this tree. Each ray in this tree can have a *weight* associated with it; this allows us to model, for example, shiny surfaces that do not reflect 100% of the incoming light.

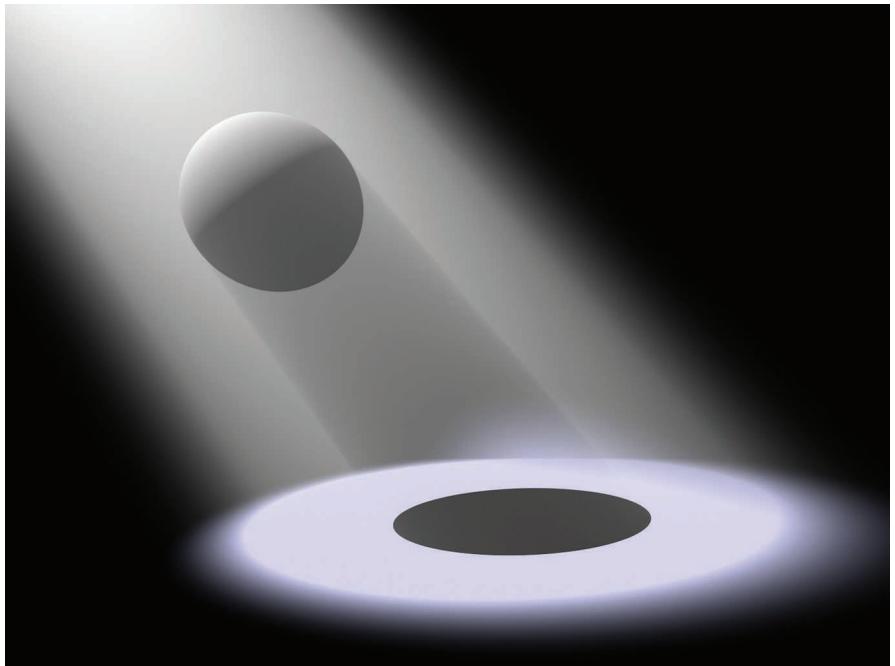
### 1.2.7 RAY PROPAGATION

The discussion so far has assumed that rays are traveling through a vacuum. For example, when describing the distribution of light from a point source, we assumed that the light's power was distributed equally on the surface of a sphere centered at the light without decreasing along the way. The presence of *participating media* such as smoke, fog, or dust can invalidate this assumption. These effects are important to simulate: even if we are not making a rendering of a smoke-filled room, almost all outdoor scenes are affected substantially by participating media. For example, Earth's atmosphere causes objects that are farther away to appear less saturated (Figure 1.9).

There are two ways in which a participating medium can affect the light propagating along a ray. First, the medium can *extinguish* (or *attenuate*) light, either by absorbing it or by scattering it in a different direction. We can capture this effect by computing the *transmittance*  $T$  between the ray origin and the intersection point. The transmittance



**Figure 1.9: Earth’s Atmosphere Decreases Saturation with Distance.** The scene on the top is rendered without simulating this phenomenon, while the scene on the bottom includes an atmospheric model. This sort of atmospheric attenuation is an important depth cue when viewing real scenes and adds a sense of scale to the rendering on the bottom.



**Figure 1.10: A Spotlight Shining on a Sphere through Fog.** Notice that the shape of the spotlight’s lighting distribution and the sphere’s shadow are clearly visible due to the additional scattering in the participating medium.

tells us how much of the light scattered at the intersection point makes it back to the ray origin.

A participating medium can also add to the light along a ray. This can happen either if the medium emits light (as with a flame) or if the medium scatters light from other directions back along the ray (Figure 1.10). We can find this quantity by numerically evaluating the *volume light transport equation*, in the same way we evaluated the light transport equation to find the amount of light reflected from a surface. We will leave the description of participating media and volume rendering until Chapters 11 and 15. For now, it will suffice to say that we can compute the effect of participating media and incorporate its effect into the amount of light carried by the ray.

## 1.3 pbrt: SYSTEM OVERVIEW

pbrt is structured using standard object-oriented techniques: abstract base classes are defined for important entities (e.g., a `Shape` abstract base class defines the interface that all geometric shapes must implement, the `Light` abstract base class acts similarly for lights, etc.). The majority of the system is implemented purely in terms of the interfaces provided by these abstract base classes; for example, the code that checks for occluding objects between a light source and a point being shaded calls the `Shape` intersection

**Table 1.1: Main Interface Types.** Most of pbrt is implemented in terms of 10 key abstract base classes, listed here. Implementations of each of these can easily be added to the system to extend its functionality.

Base class	Directory	Section
Shape	shapes/	3.1
Aggregate	accelerators/	4.2
Camera	cameras/	6.1
Sampler	samplers/	7.2
Filter	filters/	7.8
Material	materials/	9.2
Texture	textures/	10.3
Medium	media/	11.3
Light	lights/	12.2
Integrator	integrators/	1.3.3

methods and doesn't need to consider the particular types of shapes that are present in the scene. This approach makes it easy to extend the system, as adding a new shape only requires implementing a class that implements the Shape interface and linking it into the system.

pbrt is written using a total of 10 key abstract base classes, summarized in Table 1.1. Adding a new implementation of one of these types to the system is straightforward; the implementation must inherit from the appropriate base class, be compiled and linked into the executable, and the object creation routines in Appendix B must be modified to create instances of the object as needed as the scene description file is parsed. Section B.4 discusses extending the system in this manner in more detail.

The pbrt source code distribution is available from [pbrt.org](http://pbrt.org). (A large collection of example scenes is also available as a separate download.) All of the code for the pbrt core is in the `src/core` directory, and the `main()` function is contained in the short file `src/main/pbrt.cpp`. Various implementations of instances of the abstract base classes are in separate directories: `src/shapes` has implementations of the Shape base class, `src/materials` has implementations of Material, and so forth.

Throughout this section are a number of images rendered with extended versions of pbrt. Of them, Figures 1.11 through 1.14 are notable: not only are they visually impressive but also each of them was created by a student in a rendering course where the final class project was to extend pbrt with new functionality in order to render an interesting image. These images are among the best from those courses. Figures 1.15 and 1.16 were rendered with *LuxRender*, a GPL-licensed rendering system originally based on the pbrt source code from the first edition of the book. (See [www.luxrender.net](http://www.luxrender.net) for more information about *LuxRender*.)

Aggregate 255

Camera 356

Filter 474

Integrator 25

Light 714

main() 21

Material 577

Medium 684

Sampler 421

Shape 123

Texture 614



**Figure 1.11:** Guillaume Poncin and Pramod Sharma extended pbrt in numerous ways, implementing a number of complex rendering algorithms, to make this prize-winning image for Stanford’s cs348b rendering competition. The trees are modeled procedurally with L-systems, a glow image processing filter increases the apparent realism of the lights on the tree, snow was modeled procedurally with metaballs, and a subsurface scattering algorithm gave the snow its realistic appearance by accounting for the effect of light that travels beneath the snow for some distance before leaving it.

### 1.3.1 PHASES OF EXECUTION

pbrt can be conceptually divided into two phases of execution. First, it parses the scene description file provided by the user. The scene description is a text file that specifies the geometric shapes that make up the scene, their material properties, the lights that illuminate them, where the virtual camera is positioned in the scene, and parameters to all of the individual algorithms used throughout the system. Each statement in the input file has a direct mapping to one of the routines in Appendix B; these routines comprise the procedural interface for describing a scene. The scene file format is documented on the pbrt Web site, [pbrt.org](http://pbrt.org).

The end results of the parsing phase are an instance of the Scene class and an instance of the Integrator class. The Scene contains a representation of the contents of the scene (geometric objects, lights, etc.), and the Integrator implements an algorithm to render it. The integrator is so-named because its main task is to evaluate the integral from Equation (1.1).



**Figure 1.12:** Abe Davis, David Jacobs, and Jongmin Baek rendered this amazing image of an ice cave to take the grand prize in the 2009 Stanford CS348b rendering competition. They first implemented a simulation of the physical process of glaciation, the process where snow falls, melts, and refreezes over the course of many years, forming stratified layers of ice. They then simulated erosion of the ice due to melted water runoff before generating a geometric model of the ice. Scattering of light inside the volume was simulated with volumetric photon mapping; the blue color of the ice is entirely due to modeling the wavelength-dependent absorption of light in the ice volume.

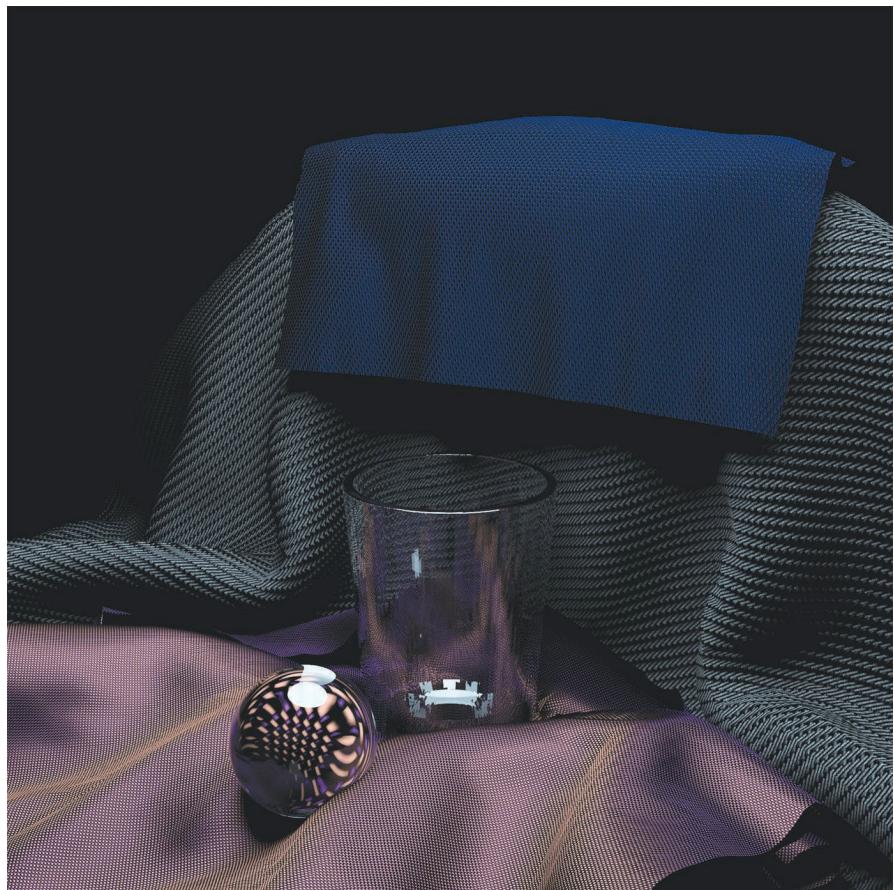
Once the scene has been specified, the second phase of execution begins, and the main rendering loop executes. This phase is where pbrt usually spends the majority of its running time, and most of this book describes code that executes during this phase. The rendering loop is performed by executing an implementation of the `Integrator::Render()` method, which is the focus of Section 1.3.4.

This chapter will describe a particular `Integrator` subclass named `SamplerIntegrator`, whose `Render()` method determines the light arriving at a virtual film plane for a large number of rays that model the process of image formation. After the contributions of all of these film samples have been computed, the final image is written to a file. The scene

`Integrator` 25

`Integrator::Render()` 25

`SamplerIntegrator` 25

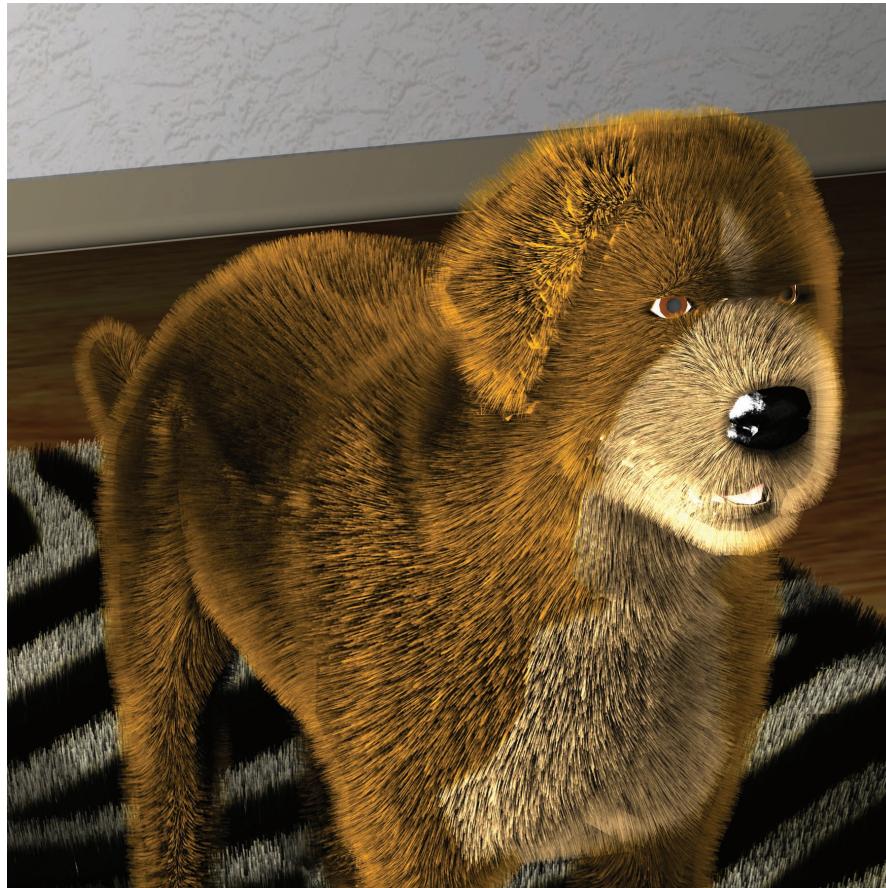


**Figure 1.13:** Lingfeng Yang implemented a bidirectional texture function to simulate the appearance of cloth, adding an analytic self-shadowing model, to render this image that took first prize in the 2009 Stanford CS348b rendering competition.

description data in memory are deallocated, and the system then resumes processing statements from the scene description file until no more remain, allowing the user to specify another scene to be rendered, if desired.

### 1.3.2 SCENE REPRESENTATION

pbrt's `main()` function can be found in the file `main/pbrt.cpp`. This function is quite simple; it first loops over the provided command-line arguments in `argv`, initializing values in the `Options` structure and storing the filenames provided in the arguments. Running pbrt with `--help` as a command-line argument prints all of the options that can be specified on the command line. The fragment that parses the command-line argu-



**Figure 1.14:** Jared Jacobs and Michael Turitzin added an implementation of Kajiya and Kay’s texel-based fur rendering algorithm (Kajiya and Kay 1989) to pbrt and rendered this image, where both the fur on the dog and the shag carpet are rendered with the texel fur algorithm.

ments, *(Process command-line arguments)*, is straightforward and therefore not included in the book here.

The options structure is then passed to the `pbrtInit()` function, which does systemwide initialization. The `main()` function then parses the given scene description(s), leading to the creation of a `Scene` and an `Integrator`. After all rendering is done, `pbrtCleanup()` does final cleanup before the system exits.

The `pbrtInit()` and `pbrtCleanup()` functions appear in a *mini-index* in the page margin, along with the number of the page where they are actually defined. The mini-indices have pointers to the definitions of almost all of the functions, classes, methods, and member variables used or referred to on each page.

`Integrator` 25

`pbrtCleanup()` 1109

`pbrtInit()` 1109

`Scene` 23



**Figure 1.15:** This contemporary indoor scene was modeled and rendered by Florent Boyer ([www.florentboyer.com](http://www.florentboyer.com)). The image was rendered using *LuxRender*, a GPL-licensed physically-based rendering system originally based on pbrt’s source code. Modeling and texturing were done using Blender.

```

⟨Main program⟩ ≡
int main(int argc, char *argv[]) {
    Options options;
    std::vector<std::string> filenames;
    ⟨Process command-line arguments⟩
    pbrtInit(options);
    ⟨Process scene description 21⟩
    pbrtCleanup();
    return 0;
}

```

If pbrt is run with no input filenames provided, then the scene description is read from standard input. Otherwise it loops through the provided filenames, processing each file in turn.

```

⟨Process scene description⟩ ≡
if (filenames.size() == 0) {
    ⟨Parse scene from standard input 22⟩
} else {
    ⟨Parse scene from input files 22⟩
}

Options 1109
ParseFile() 21
pbrtCleanup() 1109
pbrtInit() 1109

```

The `ParseFile()` function parses a scene description file, either from standard input or from a file on disk; it returns `false` if it was unable to open the file. The mechanics of parsing scene description files will not be described in this book; the parser implementation can be found in the lex and yacc files `core/pbrtlex.ll` and `core/pbrtparse.y`,



**Figure 1.16:** Martin Lubich modeled this scene of the Austrian Imperial Crown and rendered it using *LuxRender*, an open source fork of the *pbrt* codebase. The scene was modeled in Blender and consists of approximately 1.8 million vertices. It is illuminated by six area light sources with emission spectra based on measured data from a real-world light source and was rendered with 1280 samples per pixel in 73 hours of computation on a quad-core CPU. See Martin's Web site, [www.loramell.net](http://www.loramell.net), for more information, including downloadable Blender scene files.

respectively. Readers who want to understand the parsing subsystem but are not familiar with these tools may wish to consult Levine, Mason, and Brown (1992).

We use the common UNIX idiom that a file named “-” represents standard input:

```
(Parse scene from standard input) ≡  
ParseFile("-");
```

21

If a particular input file can't be opened, the `Error()` routine reports this information to the user. `Error()` uses the same format string semantics as `printf()`.

```
(Parse scene from input files) ≡  
for (const std::string &f : filenames)  
    if (!ParseFile(f))  
        Error("Couldn't open scene file \"%s\"", f.c_str());
```

21

```
Error() 1068  
ParseFile() 21  
Scene 23
```

As the scene file is parsed, objects are created that represent the lights and geometric primitives in the scene. These are all stored in the `Scene` object, which is created by the

`RenderOptions::MakeScene()` method in Section B.3.7 in Appendix B. The Scene class is declared in `core/scene.h` and defined in `core/scene.cpp`.

```
(Scene Declarations) ≡
class Scene {
public:
    (Scene Public Methods 23)
    (Scene Public Data 23)
private:
    (Scene Private Data 23)
};

(Scene Public Methods) ≡ 23
Scene(std::shared_ptr<Primitive> aggregate,
      const std::vector<std::shared_ptr<Light>> &lights)
: lights(lights), aggregate(aggregate) {
    (Scene Constructor Implementation 24)
}
```

Each light source in the scene is represented by a `Light` object, which specifies the shape of a light and the distribution of energy that it emits. The `Scene` stores all of the lights using a vector of `shared_ptr` instances from the C++ standard library. `pbzt` uses shared pointers to track how many times objects are referenced by other instances. When the last instance holding a reference (the `Scene` in this case) is destroyed, the reference count reaches zero and the `Light` can be safely freed, which happens automatically at that point.

While some renderers support separate light lists per geometric object, allowing a light to illuminate only some of the objects in the scene, this idea does not map well to the physically based rendering approach taken in `pbzt`, so `pbzt` only supports a single global per-scene list. Many parts of the system need access to the lights, so the `Scene` makes them available as a public member variable.

```
(Scene Public Data) ≡ 23
std::vector<std::shared_ptr<Light>> lights;
```

Each geometric object in the scene is represented by a `Primitive`, which combines two objects: a `Shape` that specifies its geometry, and a `Material` that describes its appearance (e.g., the object's color, whether it has a dull or glossy finish). All of the geometric primitives are collected into a single aggregate `Primitive` in the `Scene` member variable `Scene::aggregate`. This aggregate is a special kind of primitive that itself holds references to many other primitives. Because it implements the `Primitive` interface it appears no different from a single primitive to the rest of the system. The aggregate implementation stores all the scene's primitives in an acceleration data structure that reduces the number of unnecessary ray intersection tests with primitives that are far away from a given ray.

```
(Scene Private Data) ≡ 23
std::shared_ptr<Primitive> aggregate;
```

The constructor caches the bounding box of the scene geometry in the `worldBound` member variable.

Light 714  
 Material 577  
 Primitive 248  
`RenderOptions::MakeScene()` 1130  
 Scene 23  
`Scene::aggregate` 23  
`Scene::lights` 23  
 Shape 123

*(Scene Constructor Implementation)*  $\equiv$  23  
`worldBound = aggregate->WorldBound();`

*(Scene Private Data)*  $+ \equiv$  23  
`Bounds3f worldBound;`

The bound is made available via the `WorldBound()` method.

*(Scene Public Methods)*  $+ \equiv$  23  
`const Bounds3f &WorldBound() const { return worldBound; }`

Some Light implementations find it useful to do some additional initialization after the scene has been defined but before rendering begins. The Scene constructor calls their `Preprocess()` methods to allow them to do so.

*(Scene Constructor Implementation)*  $+ \equiv$  23  
`for (const auto &light : lights)  
 light->Preprocess(*this);`

The Scene class provides two methods related to ray-primitive intersection. Its `Intersect()` method traces the given ray into the scene and returns a Boolean value indicating whether the ray intersected any of the primitives. If so, it fills in the provided `SurfaceInteraction` structure with information about the closest intersection point along the ray. The `SurfaceInteraction` structure is defined in Section 4.1.

*(Scene Method Definitions)*  $\equiv$   
`bool Scene::Intersect(const Ray &ray, SurfaceInteraction *isect) const {  
 return aggregate->Intersect(ray, isect);  
}`

A closely related method is `Scene::IntersectP()`, which checks for the existence of intersections along the ray but does not return any information about those intersections. Because this routine doesn't need to search for the closest intersection or compute any additional information about intersections, it is generally more efficient than `Scene::Intersect()`. This routine is used for shadow rays.

*(Scene Method Definitions)*  $+ \equiv$   
`bool Scene::IntersectP(const Ray &ray) const {  
 return aggregate->IntersectP(ray);  
}`

### 1.3.3 INTEGRATOR INTERFACE AND SamplerIntegrator

Rendering an image of the scene is handled by an instance of a class that implements the Integrator interface. Integrator is an abstract base class that defines the `Render()` method that must be provided by all integrators. In this section, we will define one Integrator implementation, the SamplerIntegrator. The basic integrator interfaces are defined in `core/integrator.h`, and some utility functions used by integrators are in `core/integrator.cpp`. The implementations of the various integrators are in the `integrators` directory.

Bounds3f 76  
 Integrator 25  
 Light::Preprocess() 717  
 Primitive::Intersect() 249  
 Primitive::IntersectP() 249  
 Primitive::WorldBound() 249  
 Ray 73  
 SamplerIntegrator 25  
 Scene 23  
 Scene::aggregate 23  
 Scene::Intersect() 24  
 Scene::IntersectP() 24  
 Scene::lights 23  
 Scene::worldBound 24  
 SurfaceInteraction 116

```
(Integrator Declarations) ≡
    class Integrator {
    public:
        (Integrator Interface 25)
    };
```

The method that Integrators must provide is `Render()`; it is passed a reference to the Scene to use to compute an image of the scene or more generally, a set of measurements of the scene lighting. This interface is intentionally kept very general to permit a wide range of implementations—for example, one could implement an Integrator that takes measurements only at a sparse set of positions distributed through the scene rather than generating a regular 2D image.

```
(Integrator Interface) ≡
    virtual void Render(const Scene &scene) = 0;
```

25

In this chapter, we'll focus on `SamplerIntegrator`, which is an `Integrator` subclass, and the `WhittedIntegrator`, which implements the `SamplerIntegrator` interface. (Implementations of other `SamplerIntegrators` will be introduced in Chapters 14 and 15; the integrators in Chapter 16 inherit directly from `Integrator`.) The name of the `Sampler Integrator` derives from the fact that its rendering process is driven by a stream of *samples* from a Sampler; each such sample identifies a point on the image at which the integrator should compute the arriving light to form the image.

```
(SamplerIntegrator Declarations) ≡
    class SamplerIntegrator : public Integrator {
    public:
        (SamplerIntegrator Public Methods 26)
    protected:
        (SamplerIntegrator Protected Data 26)
    private:
        (SamplerIntegrator Private Data 25)
    };
```

The `SamplerIntegrator` stores a pointer to a `Sampler`. The role of the sampler is subtle, but its implementation can substantially affect the quality of the images that the system generates. First, the sampler is responsible for choosing the points on the image plane from which rays are traced. Second, it is responsible for supplying the sample positions used by integrators for estimating the value of the light transport integral, Equation (1.1). For example, some integrators need to choose random points on light sources to compute illumination from area lights. Generating a good distribution of these samples is an important part of the rendering process that can substantially affect overall efficiency; this topic is the main focus of Chapter 7.

```
(SamplerIntegrator Private Data) ≡
    std::shared_ptr<Sampler> sampler;
```

25

`Camera` 356  
`Film` 484  
`Integrator` 25  
`Sampler` 421  
`SamplerIntegrator` 25  
`Scene` 23  
`WhittedIntegrator` 32

The `Camera` object controls the viewing and lens parameters such as position, orientation, focus, and field of view. A `Film` member variable inside the `Camera` class handles image storage. The `Camera` classes are described in Chapter 6, and `Film` is described in

Section 7.9. The `Film` is responsible for writing the final image to a file and possibly displaying it on the screen as it is being computed.

*(SamplerIntegrator Protected Data)* ≡  
`std::shared_ptr<const Camera> camera;`

25

The `SamplerIntegrator` constructor stores pointers to these objects in member variables. The `SamplerIntegrator` is created in the `RenderOptions::MakeIntegrator()` method, which is in turn called by `pbrtWorldEnd()`, which is called by the input file parser when it is done parsing a scene description from an input file and is ready to render the scene.

*(SamplerIntegrator Public Methods)* ≡  
`SamplerIntegrator(std::shared_ptr<const Camera> camera,  
 std::shared_ptr<Sampler> sampler)  
: camera(camera), sampler(sampler) {}`

25

`SamplerIntegrator` implementations may optionally implement the `Preprocess()` method. It is called after the `Scene` has been fully initialized and gives the integrator a chance to do scene-dependent computation, such as allocating additional data structures that are dependent on the number of lights in the scene, or precomputing a rough representation of the distribution of radiance in the scene. Implementations that don't need to do anything along these lines can leave this method unimplemented.

*(SamplerIntegrator Public Methods)* +≡  
`virtual void Preprocess(const Scene &scene, Sampler &sampler) {}`

25

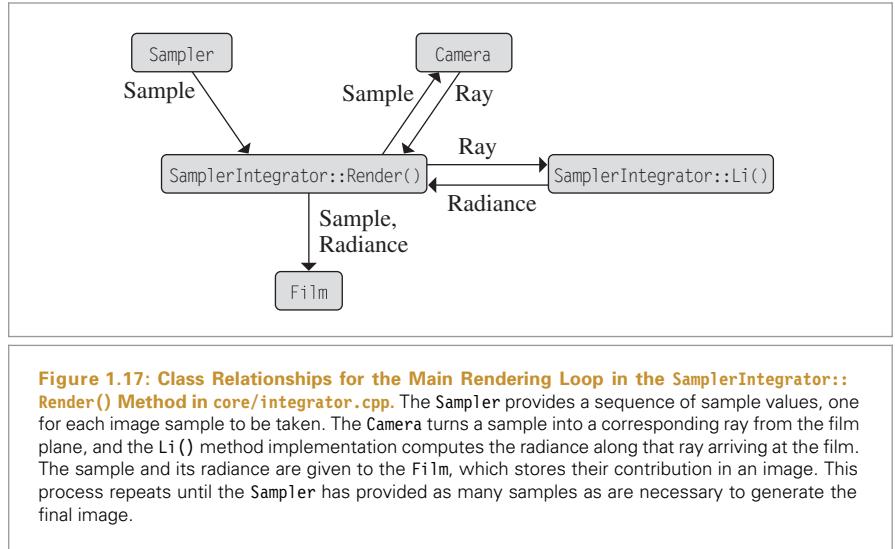
### 1.3.4 THE MAIN RENDERING LOOP

After the `Scene` and the `Integrator` have been allocated and initialized, the `Integrator::Render()` method is invoked, starting the second phase of `pbrt`'s execution: the main rendering loop. In the `SamplerIntegrator`'s implementation of this method, at each of a series of positions on the image plane, the method uses the `Camera` and the `Sampler` to generate a ray into the scene and then uses the `Li()` method to determine the amount of light arriving at the image plane along that ray. This value is passed to the `Film`, which records the light's contribution. Figure 1.17 summarizes the main classes used in this method and the flow of data among them.

*(SamplerIntegrator Method Definitions)* ≡  
`void SamplerIntegrator::Render(const Scene &scene) {  
 Preprocess(scene, *sampler);  
(Render image tiles in parallel 27)  
(Save final image after rendering 32)  
}`

So that rendering can proceed in parallel on systems with multiple processing cores, the image is decomposed into small tiles of pixels. Each tile can be processed independently and in parallel. The `ParallelFor()` function, which is described in more detail in Section A.6, implements a parallel for loop, where multiple iterations may run in parallel. A C++ lambda expression provides the loop body. Here, a variant of `ParallelFor()` that loops over a 2D domain is used to iterate over the image tiles.

`Camera` 356  
`Film` 484  
`Integrator` 25  
`Integrator::Render()` 25  
`pbrtWorldEnd()` 1129  
`RenderOptions::  
 MakeIntegrator()`  
 1130  
`Sampler` 421  
`SamplerIntegrator` 25  
`SamplerIntegrator::camera` 26  
`SamplerIntegrator:::  
 Preprocess()`  
 26  
`SamplerIntegrator::sampler` 25  
`Scene` 23



```

⟨Render image tiles in parallel⟩ ≡
⟨Compute number of tiles, nTiles, to use for parallel rendering 28⟩
ParallelFor2D(
    [&](Point2i tile) {
        ⟨Render section of image corresponding to tile 28⟩
    }, nTiles);

```

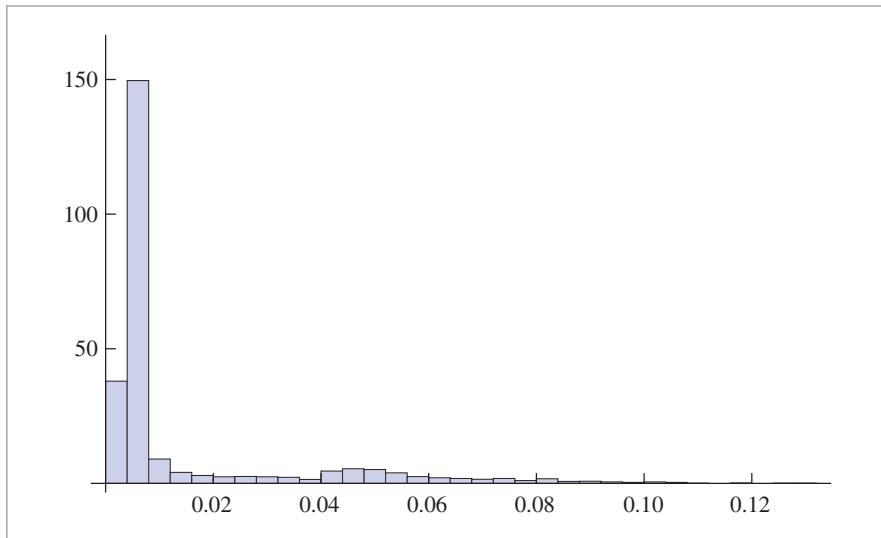
26

There are two factors to trade off in deciding how large to make the image tiles: load-balancing and per-tile overhead. On one hand, we'd like to have significantly more tiles than there are processors in the system: consider a four-core computer with only four tiles. In general, it's likely that some of the tiles will take less processing time than others; the ones that are responsible for parts of the image where the scene is relatively simple will usually take less processing time than parts of the image where the scene is relatively complex. Therefore, if the number of tiles was equal to the number of processors, some processors would finish before others and sit idle while waiting for the processor that had the longest running tile. Figure 1.18 illustrates this issue; it shows the distribution of execution time for the tiles used to render the shiny sphere scene in Figure 1.7. The longest running one took 151 times longer than the shortest one.

Camera 356  
 Film 484  
 Film::GetSampleBounds() 487  
 ParallelFor2D() 1093  
 Point2i 68  
 Sampler 421  
 SamplerIntegrator::Render() 26

On the other hand, having tiles that are too small is also inefficient. There is a small fixed overhead for a processing core to determine which loop iteration it should run next; the more tiles there are, the more times this overhead must be paid.

For simplicity, pbrt always uses  $16 \times 16$  tiles; this granularity works well for almost all images, except for very low-resolution ones. We implicitly assume that the small image case isn't particularly important to render at maximum efficiency. The Film's GetSampleBounds() method returns the extent of pixels over which samples must be generated for the image being rendered. The addition of tileSize - 1 in the computation of nTiles results in a number of tiles that is rounded to the next higher integer when the



**Figure 1.18: Histogram of Time Spent Rendering Each Tile for the Scene in Figure 1.7.** The horizontal axis measures time in seconds. Note the wide variation in execution time, illustrating that different parts of the image required substantially different amounts of computation.

sample bounds along an axis are not exactly divisible by 16. This means that the lambda function invoked by `ParallelFor()` must be able to deal with partial tiles containing some unused pixels.

*(Compute number of tiles, nTiles, to use for parallel rendering) ≡* 27  
`Bounds2i sampleBounds = camera->film->GetSampleBounds();`  
`Vector2i sampleExtent = sampleBounds.Diagonal();`  
`const int tileSize = 16;`  
`Point2i nTiles((sampleExtent.x + tileSize - 1) / tileSize,`  
`(sampleExtent.y + tileSize - 1) / tileSize);`

When the parallel for loop implementation that is defined in Appendix A.6.4 decides to run a loop iteration on a particular processor, the lambda will be called with the tile's coordinates. It starts by doing a little bit of setup work, determining which part of the film plane it is responsible for and allocating space for some temporary data before using the Sampler to generate image samples, the Camera to determine corresponding rays leaving the film plane, and the `Li()` method to compute radiance along those rays arriving at the film.

*(Render section of image corresponding to tile) ≡* 27  
`(Allocate MemoryArena for tile 29)`  
`(Get sampler instance for tile 29)`  
`(Compute sample bounds for tile 29)`  
`(Get FilmTile for tile 30)`  
`(Loop over pixels in tile to render them 30)`  
`(Merge image tile into Film 32)`

`Bounds2::Diagonal() 80`  
`Bounds2i 76`  
`Camera 356`  
`Camera::film 356`  
`Film::GetSampleBounds() 487`  
`ParallelFor() 1088`  
`Point2i 68`  
`Sampler 421`  
`SamplerIntegrator::camera 26`  
`Vector2i 60`

Implementations of the `Li()` method will generally need to temporarily allocate small amounts of memory for each radiance computation. The large number of resulting allocations can easily overwhelm the system’s regular memory allocation routines (e.g., `malloc()` or `new`), which must maintain and synchronize elaborate internal data structures to track sets of free memory regions among processors. A naive implementation could potentially spend a fairly large fraction of its computation time in the memory allocator.

To address this issue, we will pass an instance of the `MemoryArena` class to the `Li()` method. `MemoryArena` instances manage pools of memory to enable higher performance allocation than what is possible with the standard library routines.

The arena’s memory pool is always released in its entirety, which removes the need for complex internal data structures. Instances of this class can only be used by a single thread—concurrent access without additional synchronization is not permitted. We create a unique `MemoryArena` for each loop iteration that can be used directly, which also ensures that the arena is only accessed by a single thread.

```
(Allocate MemoryArena for tile) ≡
    MemoryArena arena;
```

28

Most `Sampler` implementations find it useful to maintain some state, such as the coordinates of the current pixel being sampled. This means that multiple processing threads cannot use a single `Sampler` concurrently. Therefore, `Samplers` provide a `Clone()` method to create a new instance of a given `Sampler`; it takes a seed that is used by some implementations to seed a pseudo-random number generator so that the same sequence of pseudo-random numbers isn’t generated in every tile. (Note that not all `Samplers` use pseudo-random numbers; those that don’t just ignore the seed.)

```
(Get sampler instance for tile) ≡
    int seed = tile.y * nTiles.x + tile.x;
    std::unique_ptr<Sampler> tileSampler = sampler->Clone(seed);
```

28

Next, the extent of pixels to be sampled in this loop iteration is computed based on the tile indices. Two issues must be accounted for in this computation: first, the overall pixel bounds to be sampled may not be equal to the full image resolution. For example, the user may have specified a “crop window” of just a subset of pixels to sample. Second, if the image resolution isn’t an exact multiple of 16, then the tiles on the right and bottom images won’t be a full  $16 \times 16$ .

```
(Compute sample bounds for tile) ≡
    int x0 = sampleBounds.pMin.x + tile.x * tileSize;
    int x1 = std::min(x0 + tileSize, sampleBounds.pMax.x);
    int y0 = sampleBounds.pMin.y + tile.y * tileSize;
    int y1 = std::min(y0 + tileSize, sampleBounds.pMax.y);
    Bounds2i tileBounds(Point2i(x0, y0), Point2i(x1, y1));
```

28

Finally, a `FilmTile` is acquired from the `Film`. This class provides a small buffer of memory to store pixel values for the current tile. Its storage is private to the loop iteration, so

Bounds2i 76  
 Film 484  
 FilmTile 489  
 MemoryArena 1074  
 Point2i 68  
 Sampler 421  
`Sampler::Clone()` 424

pixel values can be updated without worrying about other threads concurrently modifying the same pixels. The tile is merged into the film's storage once the work for rendering it is done; serializing concurrent updates to the image is handled then.

```
(Get FilmTile for tile) ≡ 28
    std::unique_ptr<FilmTile> filmTile =
        camera->film->GetFilmTile(tileBounds);
```

Rendering can now proceed. The implementation loops over all of the pixels in the tile using a range-based for loop that automatically uses iterators provided by the Bounds2 class. The cloned Sampler is notified that it should start generating samples for the current pixel, and samples are processed in turn until StartNextSample() returns false. (As we'll see in Chapter 7, taking multiple samples per pixel can greatly improve final image quality.)

```
(Loop over pixels in tile to render them) ≡ 28
    for (Point2i pixel : tileBounds) {
        tileSampler->StartPixel(pixel);
        do {
            <Initialize CameraSample for current sample 30>
            <Generate camera ray for current sample 31>
            <Evaluate radiance along camera ray 31>
            <Add camera ray's contribution to image 32>
            <Free MemoryArena memory from computing image sample value 32>
        } while (tileSampler->StartNextSample());
    }
```

The CameraSample structure records the position on the film for which the camera should generate the corresponding ray. It also stores time and lens position sample values, which are used when rendering scenes with moving objects and for camera models that simulate non-pinhole apertures, respectively.

```
(Initialize CameraSample for current sample) ≡ 30
    CameraSample cameraSample = tileSampler->GetCameraSample(pixel);
```

The Camera interface provides two methods to generate rays: Camera::GenerateRay(), which returns the ray for a given image sample position, and Camera::GenerateRayDifferential(), which returns a *ray differential*, which incorporates information about the rays that the Camera would generate for samples that are one pixel away on the image plane in both the *x* and *y* directions. Ray differentials are used to get better results from some of the texture functions defined in Chapter 10, making it possible to compute how quickly a texture varies with respect to the pixel spacing, a key component of texture antialiasing.

After the ray differential has been returned, the ScaleDifferentials() method scales the differential rays to account for the actual spacing between samples on the film plane for the case where multiple samples are taken per pixel.

The camera also returns a floating-point weight associated with the ray. For simple camera models, each ray is weighted equally, but camera models that more accurately model the process of image formation by lens systems may generate some rays that

Bounds2 76  
 Camera 356  
 Camera::film 356  
 Camera::GenerateRay() 357  
 Camera::  
     GenerateRayDifferential()  
     357  
 CameraSample 357  
 Film::GetFilmTile() 488  
 FilmTile 489  
 MemoryArena 1074  
 Point2i 68  
 Sampler::GetCameraSample()  
     423  
 Sampler::StartNextSample()  
     424  
 Sampler::StartPixel() 422  
 SamplerIntegrator::camera 26

contribute more than others. Such a camera model might simulate the effect of less light arriving at the edges of the film plane than at the center, an effect called *vignetting*. The returned weight will be used later to scale the ray's contribution to the image.

```
(Generate camera ray for current sample) ≡ 30
    RayDifferential ray;
    Float rayWeight = camera->GenerateRayDifferential(cameraSample, &ray);
    ray.ScaleDifferentials(1 / std::sqrt(tileSampler->samplesPerPixel));
```

Note the capitalized floating-point type `Float`: depending on the compilation flags of `pbrt`, this is an alias for either `float` or `double`. More detail on this design choice is provided in Section A.1.

Given a ray, the next task is to determine the radiance arriving at the image plane along that ray. The `Li()` method takes care of this task.

```
(Evaluate radiance along camera ray) ≡ 30
    Spectrum L(0.f);
    if (rayWeight > 0)
        L = Li(ray, scene, *tileSampler, arena);
(Issue warning if unexpected radiance value is returned)
```

`Li()` is a pure virtual method that returns the incident radiance at the origin of a given ray; each subclass of `SamplerIntegrator` must provide an implementation of this method. The parameters to `Li()` are the following:

- `ray`: the ray along which the incident radiance should be evaluated.
- `scene`: the `Scene` being rendered. The implementation will query the scene for information about the lights and geometry, and so on.
- `sampler`: a sample generator used to solve the light transport equation via Monte Carlo integration.
- `arena`: a `MemoryArena` for efficient temporary memory allocation by the integrator. The integrator should assume that any memory it allocates with the arena will be freed shortly after the `Li()` method returns and thus should not use the arena to allocate any memory that must persist for longer than is needed for the current ray.
- `depth`: the number of ray bounces from the camera that have occurred up until the current call to `Li()`.

```
Camera::: 357
    GenerateRayDifferential()
    357
Float 1062
MemoryArena 1074
RayDifferential 75
RayDifferential::: 75
    ScaleDifferentials()
    75
Sampler 421
Sampler:::samplesPerPixel 422
SamplerIntegrator 25
SamplerIntegrator:::camera 26
SamplerIntegrator:::Li() 31
Scene 23
Spectrum 315
```

The method returns a `Spectrum` that represents the incident radiance at the origin of the ray:

```
(SamplerIntegrator Public Methods) +≡ 25
    virtual Spectrum Li(const RayDifferential &ray, const Scene &scene,
                        Sampler &sampler, MemoryArena &arena, int depth = 0) const = 0;
```

A common side effect of bugs in the rendering process is that impossible radiance values are computed. For example, division by zero results in radiance values equal either to the IEEE floating-point infinity or “not a number” value. The renderer looks for this possibility, as well as for spectra with negative contributions, and prints an error message when it encounters them. Here we won't include the fragment that does this, *(Issue warning if*

*unexpected radiance value is returned*). See the implementation in `core/integrator.cpp` if you're interested in its details.

After the radiance arriving at the ray's origin is known, the image can be updated: the `FilmTile::AddSample()` method updates the pixels in the tile's image given the results from a sample. The details of how sample values are recorded in the film are explained in Sections 7.8 and 7.9.

*(Add camera ray's contribution to image) ≡* 30  
`filmTile->AddSample(cameraSample.pFilm, L, rayWeight);`

After processing a sample, all of the allocated memory in the `MemoryArena` is freed together when `MemoryArena::Reset()` is called. (See Section 9.1.1 for an explanation of how the `MemoryArena` is used to allocate memory to represent BSDFs at intersection points.)

*(Free MemoryArena memory from computing image sample value) ≡* 30  
`arena.Reset();`

Once radiance values for all of the samples in a tile have been computed, the `FilmTile` is handed off to the `Film`'s `MergeFilmTile()` method, which handles adding the tile's pixel contributions to the final image. Note that the `std::move()` function is used to transfer ownership of the `unique_ptr` to `MergeFilmTile()`.

*(Merge image tile into Film) ≡* 28  
`camera->film->MergeFilmTile(std::move(filmTile));`

After all of the loop iterations have finished, the `SamplerIntegrator`'s `Render()` method calls the `Film`'s `WriteImage()` method to write the image out to a file.

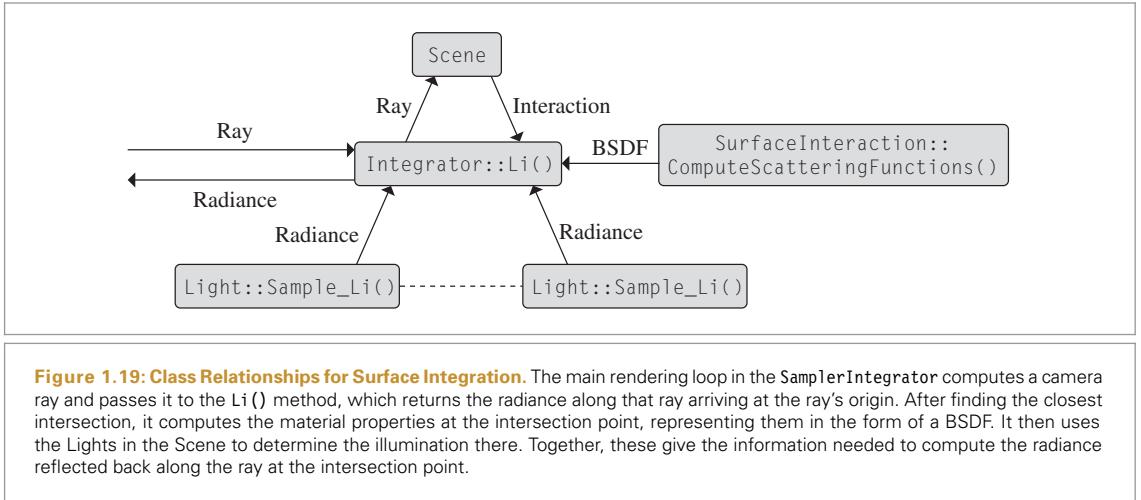
*(Save final image after rendering) ≡* 26  
`camera->film->WriteImage();`

### 1.3.5 AN INTEGRATOR FOR WHITTED RAY TRACING

Chapters 14 and 15 include the implementations of many different integrators, based on a variety of algorithms with differing levels of accuracy. Here we will present an integrator based on Whitted's ray-tracing algorithm. This integrator accurately computes reflected and transmitted light from specular surfaces like glass, mirrors, and water, although it doesn't account for other types of indirect lighting effects like light bouncing off a wall and illuminating a room. The `WhittedIntegrator` class can be found in the `integrators/whitted.h` and `integrators/whitted.cpp` files in the `pbrt` distribution.

*(WhittedIntegrator Declarations) ≡*  
`class WhittedIntegrator : public SamplerIntegrator {`  
`public:`  
 `{WhittedIntegrator Public Methods 33}`  
`private:`  
 `{WhittedIntegrator Private Data 33}`  
`};`

`Camera::film` 356  
`CameraSample::pFilm` 357  
`Film` 484  
`Film::MergeFilmTile()` 493  
`Film::WriteImage()` 494  
`FilmTile::AddSample()` 490  
`MemoryArena` 1074  
`MemoryArena::Reset()` 1076  
`SamplerIntegrator` 25  
`SamplerIntegrator::camera` 26  
`WhittedIntegrator` 32  
`WriteImage()` 1068



*(WhittedIntegrator Public Methods) ≡*

```
WhittedIntegrator(int maxDepth, std::shared_ptr<const Camera> camera,
                  std::shared_ptr<Sampler> sampler)
    : SamplerIntegrator(camera, sampler), maxDepth(maxDepth) { }
```

32

The Whitted integrator works by recursively evaluating radiance along reflected and refracted ray directions. It stops the recursion at a predetermined maximum depth, `WhittedIntegrator::maxDepth`. By default, the maximum recursion depth is five. Without this termination criterion, the recursion might never terminate (imagine, e.g., a hall-of-mirrors scene). This member variable is initialized in the `WhittedIntegrator` constructor (not included here), based on parameters set in the scene description file.

*(WhittedIntegrator Private Data) ≡*

```
const int maxDepth;
```

32

As a `SamplerIntegrator` implementation, the `WhittedIntegrator` must provide an implementation of the `Li()` method, which returns the radiance arriving at the origin of the given ray. Figure 1.19 summarizes the data flow among the main classes used during integration at surfaces.

*(WhittedIntegrator Method Definitions) ≡*

```
Spectrum WhittedIntegrator::Li(const RayDifferential &ray,
                                const Scene &scene, Sampler &sampler, MemoryArena &arena,
                                int depth) const {
    Spectrum L(0.0);
    (Find closest ray intersection or return background radiance 34)
    (Compute emitted and reflected light at ray intersection point 34)
    return L;
}
```

Camera 356  
MemoryArena 1074  
RayDifferential 75  
Sampler 421  
SamplerIntegrator 25  
Scene 23  
Scene::Intersect() 24  
Spectrum 315  
WhittedIntegrator 32  
WhittedIntegrator::maxDepth 33

The first step is to find the first intersection of the ray with the shapes in the scene. The `Scene::Intersect()` method takes a ray and returns a Boolean value indicating whether

it intersected a shape. For rays where an intersection was found, it initializes the provided `SurfaceInteraction` with geometric information about the intersection.

If no intersection was found, radiance may be carried along the ray due to light sources that don't have associated geometry. One example of such a light is the `InfiniteAreaLight`, which can represent illumination from the sky. The `Light::Le()` method allows such lights to return their radiance along a given ray.

*(Find closest ray intersection or return background radiance) ≡ 33*

```
SurfaceInteraction isect;
if (!scene.Intersect(ray, &isect)) {
    for (const auto &light : scene.lights)
        L += light->Le(ray);
    return L;
}
```

Otherwise a valid intersection has been found. The integrator must determine how light is scattered by the surface of the shape at the intersection point, determine how much illumination is arriving from light sources at the point, and apply an approximation to Equation (1.1) to compute how much light is leaving the surface in the viewing direction. Because this integrator ignores the effect of participating media like smoke or fog, the radiance leaving the intersection point is the same as the radiance arriving at the ray's origin.

*(Compute emitted and reflected light at ray intersection point) ≡ 33*

```
(Initialize common variables for Whitted integrator 34)
(Compute scattering functions for surface interaction 35)
(Compute emitted light if ray hit an area light source 35)
>Add contribution of each light source 36
if (depth + 1 < maxDepth) {
    (Trace rays for specular reflection and refraction 37)
}
```

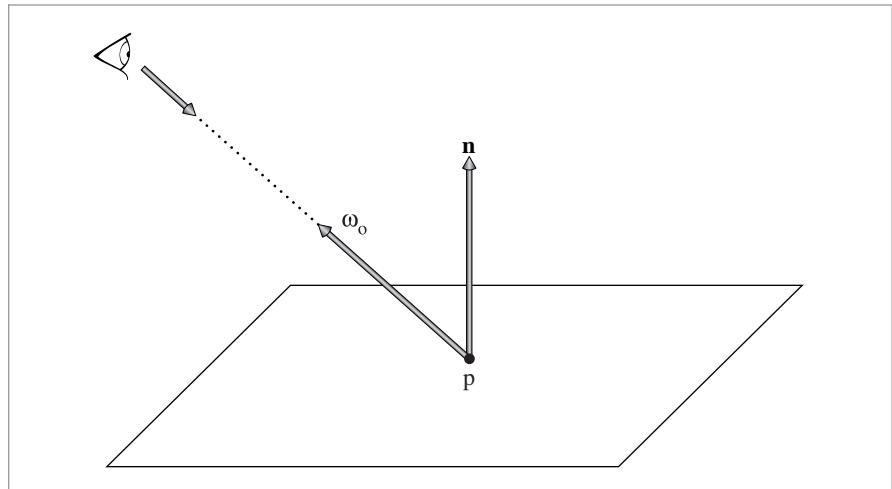
Figure 1.20 shows a few quantities that will be used frequently in the fragments to come.  $n$  is the surface normal at the intersection point and the normalized direction from the hit point back to the ray origin is stored in  $w_o$ ; Cameras are responsible for normalizing the direction component of generated rays, so there's no need to renormalize it here. Normalized directions are denoted by the  $\omega$  symbol in this book, and in pbrt's code we will use  $w_o$  to represent  $\omega_o$ , the outgoing direction of scattered light.

*(Initialize common variables for Whitted integrator) ≡ 34*

```
Normal3f n = isect.shading.n;
Vector3f wo = isect.wo;
```

If an intersection was found, it's necessary to determine how the surface's material scatters light. The `ComputeScatteringFunctions()` method handles this task, evaluating texture functions to determine surface properties and then initializing a representation of the BSDF (and possibly BSSRDF) at the point. This method generally needs to allocate memory for the objects that constitute this representation; because this memory only

Camera 356  
`InfiniteAreaLight` 737  
`Interaction::wo` 115  
`Light::Le()` 741  
`Normal3f` 71  
`Scene::Intersect()` 24  
`Scene::lights` 23  
`SurfaceInteraction` 116  
`SurfaceInteraction::shading` 118  
`SurfaceInteraction::shading::n` 118  
`Vector3f` 60  
`WhittedIntegrator::maxDepth` 33



**Figure 1.20: Geometric Setting for the Whitted Integrator.**  $p$  is the ray intersection point and  $n$  is its surface normal. The direction in which we'd like to compute reflected radiance is  $\omega_o$ ; it is the vector pointing in the opposite direction of the incident ray.

needs to be available for the current ray, the `MemoryArena` is provided for it to use for its allocations.

```
(Compute scattering functions for surface interaction) ≡  
    isect.ComputeScatteringFunctions(ray, arena);
```

34

In case the ray happened to hit geometry that is emissive (such as an area light source), the integrator accounts for any emitted radiance by calling the `SurfaceInteraction::Le()` method. This gives the first term of the light transport equation, Equation (1.1) on page 12. If the object is not emissive, this method returns a black spectrum.

```
(Compute emitted light if ray hit an area light source) ≡  
    L += isect.Le(wo);
```

34

For each light, the integrator calls the `Light::Sample_Li()` method to compute the radiance from that light falling on the surface at the point being shaded. This method also returns the direction vector from the point being shaded to the light source, which is stored in the variable  $w_i$  (denoting an incident direction  $\omega_i$ ).<sup>4</sup>

The spectrum returned by this method does not account for the possibility that some other shape may block light from the light and prevent it from reaching the point being shaded. Instead, it returns a `VisibilityTester` object that can be used to determine if any primitives block the surface point from the light source. This test is done by tracing a shadow ray between the point being shaded and the light to verify that the path is

---

**Integrator** 25  
**Light::Sample\_Li()** 716  
**MemoryArena** 1074  
**SurfaceInteraction::ComputeScatteringFunctions()**  
 578  
**SurfaceInteraction::Le()** 734  
**VisibilityTester** 717

<sup>4</sup> When considering light scattering at a surface location, pbrt uses the convention that  $\omega_i$  always refers to the direction from which the quantity of interest (radiance in this case) arrives, rather than the direction from which the Integrator reached the surface.

clear. pbrt's code is organized in this way so that it can avoid tracing the shadow ray unless necessary: this way it can first make sure that the light falling on the surface *would* be scattered in the direction  $\omega_o$  if the light isn't blocked. For example, if the surface is not transmissive, then light arriving at the back side of the surface doesn't contribute to reflection.

The `Sample_Li()` method also returns the probability density for the light to have sampled the direction  $\omega_i$  in the `pdf` variable. This value is used for Monte Carlo integration with complex area light sources where light is arriving at the point from many directions even though just one direction is sampled here; for simple lights like point lights, the value of `pdf` is one. The details of how this probability density is computed and used in rendering are the topics of Chapters 13 and 14; in the end, the light's contribution must be divided by `pdf`, so this is done by the implementation here.

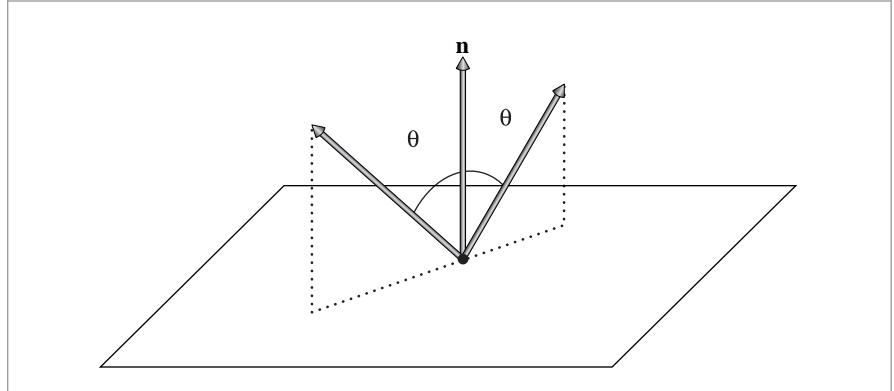
If the arriving radiance is nonzero and the BSDF indicates that some of the incident light from the direction  $\omega_i$  is in fact scattered to the outgoing direction  $\omega_o$ , then the integrator multiplies the radiance value  $L_i$  by the value of the BSDF  $f$  and the cosine term. The cosine term is computed using the `AbsDot()` function, which returns the absolute value of the dot product between two vectors. If the vectors are normalized, as both  $\omega_i$  and  $n$  are here, this is equal to the absolute value of the cosine of the angle between them (Section 2.2.1).

This product represents the light's contribution to the light transport equation integral, Equation (1.1), and it is added to the total reflected radiance  $L$ . After all lights have been considered, the integrator has computed the total contribution of *direct lighting*—light that arrives at the surface directly from emissive objects (as opposed to light that has reflected off other objects in the scene before arriving at the point).

```
(Add contribution of each light source) ≡ 34
for (const auto &light : scene.lights) {
    Vector3f wi;
    Float pdf;
    VisibilityTester visibility;
    Spectrum Li = light->Sample_Li(isect, sampler.Get2D(), &wi,
                                    &pdf, &visibility);
    if (Li.IsBlack() || pdf == 0) continue;
    Spectrum f = isect.bsdf->f(wo, wi);
    if (!f.IsBlack() && visibility.Unoccluded(scene))
        L += f * Li * AbsDot(wi, n) / pdf;
}
```

`AbsDot()` 64  
`BSDF::f()` 575  
`Float` 1062  
`Light::Sample_Li()` 716  
`Sampler::Get2D()` 422  
`SamplerIntegrator` 25  
`Scene::lights` 23  
`Spectrum` 315  
`Spectrum::IsBlack()` 317  
`SurfaceInteraction::bsdf` 250  
`Vector3f` 60  
`VisibilityTester` 717  
`VisibilityTester::Unoccluded()` 718

This integrator also handles light scattered by perfectly specular surfaces like mirrors or glass. It is fairly simple to use properties of mirrors to find the reflected directions (Figure 1.21) and to use Snell's law to find the transmitted directions (Section 8.2). The integrator can then recursively follow the appropriate ray in the new direction and add its contribution to the reflected radiance at the point originally seen from the camera. The computation of the effect of specular reflection and transmission is handled in separate utility methods so these functions can easily be reused by other `SamplerIntegrator` implementations.



**Figure 1.21:** Reflected rays due to perfect specular reflection make the same angle with the surface normal as the incident ray.

*(Trace rays for specular reflection and refraction)  $\equiv$*

```
L += SpecularReflect(ray, isect, scene, sampler, arena, depth);
L += SpecularTransmit(ray, isect, scene, sampler, arena, depth);
```

34

*(SamplerIntegrator Method Definitions)  $+ \equiv$*

```
Spectrum SamplerIntegrator::SpecularReflect(const RayDifferential &ray,
                                              const SurfaceInteraction &isect, const Scene &scene,
                                              Sampler &sampler, MemoryArena &arena, int depth) const {
    <Compute specular reflection direction wi and BSDF value 38>
    <Return contribution of specular reflection 38>
}
```

**BSDF** 572  
**BSDF::Sample\_f()** 832  
**MemoryArena** 1074  
**RayDifferential** 75  
**Sampler** 421  
**SamplerIntegrator::Li()** 31  
**SamplerIntegrator::SpecularReflect()**  
 37  
**SamplerIntegrator::SpecularTransmit()**  
 38  
**Scene** 23  
**Spectrum** 315  
**SurfaceInteraction** 116  
**WhittedIntegrator::Li()** 33

In the `SpecularReflect()` and `SpecularTransmit()` methods, the `BSDF::Sample_f()` method returns an incident ray direction for a given outgoing direction and a given mode of light scattering. This method is one of the foundations of the Monte Carlo light transport algorithms that will be the subject of the last few chapters of this book. Here, we will use it to find only outgoing directions corresponding to perfect specular reflection or refraction, using flags to indicate to `BSDF::Sample_f()` that other types of reflection should be ignored. Although `BSDF::Sample_f()` can sample random directions leaving the surface for probabilistic integration algorithms, the randomness is constrained to be consistent with the BSDF's scattering properties. In the case of perfect specular reflection or refraction, only one direction is possible, so there is no randomness at all.

The calls to `BSDF::Sample_f()` in these functions initialize  $w_i$  with the chosen direction and return the BSDF's value for the directions  $(\omega_o, \omega_i)$ . If the value of the BSDF is nonzero, the integrator uses the `SamplerIntegrator::Li()` method to get the incoming radiance along  $\omega_i$ , which in this case will in turn resolve to the `WhittedIntegrator::Li()` method.

```
(Compute specular reflection direction wi and BSDF value) ≡ 37
Vector3f wo = isect.wo, wi;
Float pdf;
BxDFType type = BxDFType(BSDF_REFLECTION | BSDF_SPECULAR);
Spectrum f = isect.bsdf->Sample_f(wo, &wi, sampler.Get2D(), &pdf, type);
```

In order to use ray differentials to antialias textures that are seen in reflections or refractions, it is necessary to know how reflection and transmission affect the screen-space footprint of rays. The fragments that compute the ray differentials for these rays are defined later, in Section 10.1.3. Given the fully initialized ray differential, a recursive call to `Li()` provides incident radiance, which is scaled by the value of the BSDF, the cosine term, and divided by the PDF, as per Equation (1.1).

```
(Return contribution of specular reflection) ≡ 37
const Normal3f &ns = isect.shading.n;
if (pdf > 0 && !f.IsBlack() && AbsDot(wi, ns) != 0) {
    (Compute ray differential rd for specular reflection 607)
    return f * Li(rd, scene, sampler, arena, depth + 1) * AbsDot(wi, ns) /
        pdf;
}
else
    return Spectrum(0.f);
```

The `SpecularTransmit()` method is essentially the same as `SpecularReflect()` but just requests the `BSDF_TRANSMISSION` specular component of the BSDF, if any, rather than the `BSDF_REFLECTION` component used by `SpecularReflect()`. We therefore won't include its implementation in the text of the book here.

## 1.4 PARALLELIZATION OF pbrt

It's now nearly impossible to buy a new laptop or desktop computer with only one processing core. (The same even holds for mobile phones, for that matter.) The computer systems of today and of the future will increasingly feature multiple processing cores, both on CPUs and on highly parallel throughput processors like GPUs. At the same time, the computational capabilities of their cores are only improving slowly; as such, significant increases in performance over time are now only available to programs that can run in parallel, with many separate threads of execution performing computation simultaneously on multiple cores.

Writing a parallel program is more difficult than writing a serial program. When two computations that the programmer believes are independent are executing simultaneously but then interact unexpectedly, the program may crash or generate unexpected results. However, such a bug may not manifest itself again if the program is run again, perhaps due to those particular computations not running simultaneously during the

AbsDot() 64  
 BSDF::Sample\_f() 832  
 BSDF\_REFLECTION 513  
 BSDF\_SPECULAR 513  
 BSDF\_TRANSMISSION 513  
 BxDFType 513  
 Float 1062  
 Interaction::wo 115  
 Normal3f 71  
 Sampler::Get2D() 422  
 SamplerIntegrator::Li() 31  
 Spectrum 315  
 Spectrum::IsBlack() 317  
 SurfaceInteraction::shading 118  
 SurfaceInteraction::shading::n 118  
 Vector3f 60

next run. Fortunately, increasingly good tools to help developers find these sorts of interactions are increasingly available.<sup>5</sup>

For a parallel program to scale well to many processors, it needs to be able to provide a substantial amount of independent computation: any computation dependent on results of prior computation can't be run concurrently with the computation it depends on. Fortunately, most rendering algorithms based on ray tracing have abundant parallelism; for the `SamplerIntegrator`, each image sample is independent of all of the other ones, and many millions of samples may be used for high-quality images.

One of the biggest challenges with parallel ray tracing is the impact of non-parallel phases of computation. For example, it's not as easy to effectively parallelize the construction of many acceleration structures while the scene is being constructed than it is to parallelize rendering. While this may seem like a minor issue, *Amdahl's law*, which describes the speedup of a workload that has both serial and parallel phases, points to the challenge. Given  $n$  cores performing computation and a workload where the fraction  $s$  of its overall computation is inherently serial, then the maximum speedup possible is

$$\frac{1}{s + \frac{1}{n}(1-s)}.$$

Thus, even with an infinite number of cores, the maximum speedup is  $1/s$ . If, for example, a seemingly innocuous 5% of the run time is spent in a serial phase of parsing the scene file and building acceleration structures, the maximum speedup possible is  $1/0.05 = 20\times$ , no matter how quickly the parallel phase executes.

### 1.4.1 DATA RACES AND COORDINATION

In pbrt, we assume that the computation is running on processors that provide *coherent shared memory*. The main idea of coherent shared memory is that all threads can read and write to a common set of memory locations and that changes to memory made by one thread will eventually be seen by other threads. These properties greatly simplify the implementation of the system as there's no need to explicitly communicate data between cores. (Coherent shared memory is generally available on today's CPUs and is likely to continue to be on future CPUs. On the other hand, if one is running a computation across multiple computers in a cluster, coherent shared memory is generally not available.)

Although coherent shared memory relieves the need for separate threads to explicitly communicate data with each other, they still need to *coordinate* their access to shared data; a danger of coherent shared memory is *data races*. If two threads modify the same memory location without coordination between the two of them, the program will almost certainly compute incorrect results or even crash. Consider the example of two processors simultaneously running the following innocuous-looking code, where `globalCounter` starts with a value of two:

---

SamplerIntegrator 25

<sup>5</sup> We found the open-source tool helgrind, part of the valgrind suite of tools, instrumental for helping to find bugs in pbrt's parallel code as we were developing it. "Thread sanitizer" is also well regarded.

```
extern int globalCounter;
if (--globalCounter == 0)
    printf("done!\n");
```

Because the two threads don't coordinate their reading and writing of `globalCounter`, it is possible that "done" will be printed zero, one, or even two times! The assembly instructions generated by the compiler are likely to correspond to steps something like the following:

```
extern int globalCounter;
int temp = globalCounter;
temp = temp - 1;
globalCounter = temp;
if (globalCounter == 0)
    printf("done!\n");
```

Now, consider different ways this code could be executed on two processors. For example, the second processor could start executing slightly after the first one, but the first one could go idle for a few cycles after executing the first few instructions:

Thread A	Thread B
<code>int temp = globalCounter;</code>	
<code>temp = temp - 1;</code>	<code>(idle)</code>
<code>globalCounter = temp;</code>	
<code>// (idle)</code>	<code>int temp = globalCounter;</code>
	<code>temp = temp - 1;</code>
	<code>globalCounter = temp;</code>
<code>if (globalCounter == 0)</code>	<code>if (globalCounter == 0)</code>
<code>printf("done!\n");</code>	<code>printf("done!\n");</code>

(Many unpredictable events can cause these sorts of execution bubbles, ranging from the OS interrupting the thread to cache misses.) In this ordering, both threads read the value of zero from `globalCounter`, and both execute the `printf()` call. In this case, the error is not fatal, but if instead the system was freeing resources in the `if` block, then it would attempt to free the same resources twice, which would very likely cause a crash. Consider now this potential execution order:

Thread A	Thread B
<code>int temp = globalCounter;</code>	<code>int temp = globalCounter;</code>
<code>temp = temp - 1;</code>	<code>temp = temp - 1;</code>
<code>globalCounter = temp;</code>	<code>// (idle)</code>
<code>// (idle)</code>	<code>globalCounter = temp;</code>
<code>if (globalCounter == 0)</code>	<code>if (globalCounter == 0)</code>
<code>printf("done!\n");</code>	<code>printf("done!\n");</code>

In this case, `globalCounter` ends up with a value of one, and neither thread executes the `if` block. These examples illustrate the principle that when multiple threads of execution are accessing shared modified data, they must somehow synchronize their access.

Two main mechanisms are available today for doing this type of synchronization: mutual exclusion and atomic operations. Mutual exclusion is implemented with `std::mutex`

objects in pbrt. A `std::mutex` can be used to protect access to some resource, ensuring that only one thread can access it at a time. Consider the following updated version of the previous computation; here a `std::lock_guard` object acquires a lock on the mutex and releases it when it goes out of scope at the final brace.

```
extern int globalCounter;
extern std::mutex globalCounterMutex;
{ std::lock_guard<std::mutex> lock(globalCounterMutex);
    int temp = globalCounter;
    temp = temp - 1;
    globalCounter = temp;
    if (globalCounter == 0)
        printf("done!\n");
}
```

If two threads are executing this code and try to acquire the mutex at the same time, then the mutex will allow only one of them to proceed, stalling the other one in the `std::lock_guard` constructor. Only when the first thread has finished the computation and its `std::lock_guard` goes out of scope, releasing the lock on the mutex, is the second thread able to acquire the mutex itself and continue the computation.

Thread A	Thread B
{ std::lock_guard<std::mutex> lock(	{ std::lock_guard<std::mutex> lock(
globalCounterMutex);	globalCounterMutex);
int temp = globalCounter;	// (stalled by mutex)
:	
:	
}	// (mutex released)
	// (mutex acquired)
	int temp = globalCounter;
	:
	:
	} // (mutex released)

With correct mutual exclusion here, the `printf()` will only be executed once, no matter what the ordering of execution between the two threads is.

*Atomic memory operations* (or *atomics*) are the other option for correctly performing this type of memory update with multiple threads. Atomics are machine instructions that guarantee that their respective memory updates will be performed in a single transaction. (*Atomic* in this case refers to the notion that the memory updates are indivisible.) The implementations of atomic operations in pbrt are from the C++11 standard library and are further discussed in Appendix A.6.2. Using atomics, the computation above could be written to use the `std::atomic<int>` type, which has overloaded add, subtract, increment, and decrement operations, as below:

```
extern std::atomic<int> globalCounter;
if (--globalCounter == 0)
    printf("done!\n");
```

The `std::atomic --` operator subtracts 1 from the given variable, `globalCounter`, and returns the previous value of the variable. Using an atomic operation ensures that if two threads simultaneously try to update the variable then not only will the final value of the variable be the expected value but each thread will be returned the value of the variable after its update alone. In this example, then, `globalCounter` will end up with a value of zero, as expected, with one thread guaranteed to have the value one returned from the atomic subtraction and the other thread guaranteed to have zero returned.

An additional option, *transactional memory*, is just starting to become available in CPUs as of this writing. With transactional memory, a set of memory writes are bundled as a transaction; if no other threads access those memory locations while the transaction is executing, then all of the writes are committed in a single atomic operation. Otherwise, it is rolled back and none of the writes reach memory, and thus the computation has had no effect; the transaction must then be tried again. Transactional memory helps bridge the fine-grained operation of atomics and the higher overhead of mutexes. However, because it isn't yet widely available, transactional memory isn't currently used in pbrt.

Section A.6 in Appendix A has more information about parallel programming, with additional details on performance issues and pitfalls, as well as the various routines used in the parallel implementation of pbrt.

### 1.4.2 CONVENTIONS IN pbrt

In pbrt (as is the case for most ray tracers) the vast majority of data at render time is read only (e.g., the scene description and texture maps). Almost all of the parsing of the scene file and creation of the scene representation in memory is done with a single thread of execution, so there are no synchronization issues during that phase of execution.<sup>6</sup> During rendering, concurrent read access to all of the read-only data by multiple threads works with no problems; we only need to be concerned with situations where data in memory is being modified.

When adding new code to pbrt, it's important to make sure to not inadvertently add code that modifies shared data without proper synchronization. This is usually straightforward; for example, when adding a new Shape implementation, the Shape will normally only perform read accesses to its member variables after it has been created. Sometimes, however, shared data may be inadvertently introduced. Consider the following code idiom, often seen in single-threaded code:

```
static bool firstCall = true;
if (firstCall) {
    : additional initialization
    firstCall = false;
}
```

BVHAccel 256

Shape 123

---

<sup>6</sup> The two exceptions are some image resampling performed on image texture maps, and construction of one variant of the BVHAccel, though both of these are highly localized.

This code is unsafe with multiple threads of execution, as multiple threads may see the value of `firstCall` as `true` and all execute the initialization code. Writing this safely requires either atomic operations or mutexes. (This particular idiom can also be implemented safely using the `std::call_once()` function.)

### 1.4.3 THREAD SAFETY EXPECTATIONS IN pbrt

Many class methods in pbrt are required to be safe for multiple concurrent threads of execution. Particular instances of these methods must either be safe naturally due to not updating shared global data or due to using mutexes or atomic operations to safely perform any updates that are needed.

As a general rule, the low-level classes and structures in the system are not thread-safe. For example, the `Point3f` class, which stores three `float` values to represent a point in 3D space, is not safe for multiple threads to call methods that modify it at the same time. (Multiple threads can use `Point3fs` as read-only data simultaneously, of course.) The run-time overhead to make `Point3f` thread-safe would have a substantial effect on performance with little benefit in return.

The same is true for classes like `Vector3f`, `Normal3f`, `Spectrum`, `Transform`, `Quaternion`, and `SurfaceInteraction`. These classes are usually either created at scene construction time and then used as read-only data or allocated on the stack during rendering and used only by a single thread.

The utility classes `MemoryArena` (used for high-performance temporary memory allocation) and `RNG` (pseudo-random number generation) are also not safe for use by multiple threads; these classes store state that is modified when their methods are called, and the overhead from protecting modification to their state with mutual exclusion would be excessive relative to the amount of computation they perform. Consequently, in code like the `SamplerIntegrator::Render()` method above, the implementation allocates per-thread instances of these classes on the stack.

With two exceptions, implementations of the higher level abstract base classes listed in Table 1.1 are all expected to be safe for multiple threads to use simultaneously. With a little care, it is usually straightforward to implement specific instances of these base classes so they don't modify any shared state in their methods.

The first exceptions are the `SamplerIntegrator` and `Light Preprocess()` methods. These are called by the system during scene construction, and implementations of them generally modify shared state in their implementations—for example, by building data structures that represent the distribution of illumination in the scene. Therefore, it's helpful to allow the implementer to assume that only a single thread will call into these methods. (This is a separate issue from the consideration that implementations of these methods that are computationally intensive may use `ParallelFor()` to parallelize their computation.)

The second exception is the `Sampler`; its methods are also not expected to be thread safe. This is another instance where this requirement would impose an excessive performance and scalability impact; many threads simultaneously trying to get samples from a single

<code>Light</code>	714
<code>MemoryArena</code>	1074
<code>Normal3f</code>	71
<code>ParallelFor()</code>	1088
<code>Point3f</code>	68
<code>Quaternion</code>	99
<code>RNG</code>	1065
<code>Sampler</code>	421
<code>SamplerIntegrator</code>	25
<code>SamplerIntegrator::Render()</code>	26
<code>Spectrum</code>	315
<code>SurfaceInteraction</code>	116
<code>Transform</code>	83
<code>Vector3f</code>	60

`Sampler` would limit the system's overall performance. Therefore, as described in Section 1.3.4, a unique `Sampler` is created for each image tile using `Sampler::Clone()`; this sampler can then be used for just the one tile, without any mutual exclusion overhead.

All stand-alone functions in `pbrt` are thread-safe (as long as multiple threads don't pass pointers to the same data to them).

## 1.5 HOW TO PROCEED THROUGH THIS BOOK

We wrote this book assuming it will be read in roughly front-to-back order. Generally, we tried to minimize the number of forward references to ideas and interfaces that haven't yet been introduced, but we do assume that the reader is acquainted with the previous content at any particular point in the text. However, some sections go into depth about advanced topics that some readers may wish to skip over (particularly on first reading); each advanced section is identified by an asterisk in its title.

Because of the modular nature of the system, the main requirement is that the reader be familiar with the low-level classes like `Point3f`, `Ray`, and `Spectrum`; the interfaces defined by the abstract base classes listed in Table 1.1; and the rendering loop in `Sampler Integrator::Render()`. Given that knowledge, for example, the reader who doesn't care about precisely how a camera model based on a perspective projection matrix maps `CameraSamples` to rays can skip over the implementation of that camera and can just remember that the `Camera::GenerateRayDifferential()` method somehow turns a `CameraSample` into a `RayDifferential`.

The rest of this book is divided into four main parts of a few chapters each. First, Chapters 2 through 4 define the main geometric functionality in the system. Chapter 2 has the low-level classes like `Point3f`, `Ray`, and `Bounds3f`. Chapter 3 defines the `Shape` interface, gives implementations of a number of shapes, and shows how to perform ray-shape intersection tests. Chapter 4 has the implementations of the acceleration structures for speeding up ray tracing by skipping tests with primitives that a ray can be shown to definitely not intersect.

The second part covers the image formation process. First, Chapter 5 introduces the physical units used to measure light and the `Spectrum` class that represents wavelength-varying distributions (i.e., color). Chapter 6 defines the `Camera` interface and has a few different camera implementations. The `Sampler` classes that place samples on the image plane are the topic of Chapter 7, and the overall process of turning radiance values on the film into images suitable for display is explained in Section 7.9.

The third part of the book is about light and how it scatters from surfaces and participating media. Chapter 8 includes a set of building-block classes that define a variety of types of reflection from surfaces. Materials, described in Chapter 9, use these reflection functions to implement a number of different surface materials, such as plastic, glass, and metal. Chapter 10 introduces texture, which describes variation in material properties (color, roughness, etc.) over surfaces, and Chapter 11 has the abstractions that describe how light is scattered and absorbed in participating media. Finally, Chapter 12 has the interface for light sources and a number of light source implementations.

<code>Bounds3f</code>	76
<code>Camera</code>	356
<code>Camera::GenerateRayDifferential()</code>	357
<code>CameraSample</code>	357
<code>Point3f</code>	68
<code>Ray</code>	73
<code>RayDifferential</code>	75
<code>Sampler</code>	421
<code>Sampler::Clone()</code>	424
<code>SamplerIntegrator::Render()</code>	26
<code>Shape</code>	123
<code>Spectrum</code>	315

The last part brings all of the ideas from the rest of the book together to implement a number of interesting light transport algorithms. Chapter 13 introduces the theory of Monte Carlo integration, a statistical technique for estimating the value of complex integrals, and describes low-level routines for applying Monte Carlo to illumination and light scattering. The integrators in Chapters 14, 15, and 16 use Monte Carlo integration to compute more accurate approximations of the light transport equation than the `WhittedIntegrator`, using techniques like path tracing, bidirectional path tracing, Metropolis light transport, and photon mapping.

Chapter 17, the last chapter of the book, provides a brief retrospective and discussion of system design decisions along with a number of suggestions for more far-reaching projects than those in the exercises. Appendices describe utility functions and details of how the scene description is created as the input file is parsed.

### 1.5.1 THE EXERCISES

At the end of each chapter you will find exercises related to the material covered in that chapter. Each exercise is marked as one of three levels of difficulty:

- ➊ An exercise that should take only an hour or two
- ➋ A reading and/or implementation task that would be suitable for a course assignment and should take between 10 and 20 hours of work
- ➌ A suggested final project for a course that will likely take 40 hours or more to complete

## 1.6 USING AND UNDERSTANDING THE CODE

We wrote `pbrt` in C++ but focused on readability for non-C++ experts by limiting usage of esoteric features of the language. Staying close to the core language features also helps with the system’s portability. In particular, we avoid multiple inheritance, run-time exception handling, and excessive use of C++11 and C++14 features. We also use only a small subset of C++’s extensive standard library.

We will occasionally omit short sections of `pbrt`’s source code from the book. For example, when there are a number of cases to be handled, all with nearly identical code, we will present one case and note that the code for the remaining cases has been omitted from the text. Of course, all the omitted code can be found in the `pbrt` source code distribution.

### 1.6.1 POINTER OR REFERENCE?

C++ provides two different mechanisms for passing the address of a data structure to a function or method: pointers and references. If a function argument is not intended as an output variable, either can be used to save the expense of passing the entire structure on the stack. By convention, `pbrt` uses pointers when the argument will be completely changed by the function or method, references when some of its internal state will be changed but it won’t be fully re-initialized, and `const` references when it won’t be changed at all. One important exception to this rule is that we will always use a pointer when we

want to be able to pass `nullptr` to indicate that a parameter is not available or should not be used.

### 1.6.2 ABSTRACTION VERSUS EFFICIENCY

One of the primary tensions when designing interfaces for software systems is making a reasonable trade-off between abstraction and efficiency. For example, many programmers religiously make all data in all classes `private` and provide methods to obtain or modify the values of the data items. For simple classes (e.g., `Vector3f`), we believe that approach needlessly hides a basic property of the implementation—that the class holds three floating-point coordinates—that we can reasonably expect to never change. Of course, using no information hiding and exposing all details of all classes’ internals leads to a code maintenance nightmare. Yet, we believe that there is nothing wrong with judiciously exposing basic design decisions throughout the system. For example, the fact that a `Ray` is represented with a point, a vector, and values that give its extent, time, and recursion depth is a decision that doesn’t need to be hidden behind a layer of abstraction. Code elsewhere is shorter and easier to understand when details like these are exposed.

An important thing to keep in mind when writing a software system and making these sorts of trade-offs is the expected final size of the system. The core of `pbrt` (excluding the implementations of specific shapes, lights, and so forth), where all of the basic interfaces, abstractions, and policy decisions are defined, is under 20,000 lines of code. Adding additional functionality to the system will generally only increase the amount of code in the implementations of the various abstract base classes. `pbrt` is never going to grow to be a million lines of code; this fact can and should be reflected in the amount of information hiding used in the system. It would be a waste of programmer time (and likely a source of run-time inefficiency) to design the interfaces to accommodate a system of that level of complexity.

### 1.6.3 CODE OPTIMIZATION

We tried to make `pbrt` efficient through the use of well-chosen algorithms rather than through local micro-optimizations, so that the system can be more easily understood. However, we applied some local optimizations to the parts of `pbrt` that account for the most execution time, as long as doing so didn’t make the code too hard to understand. There are two main local optimization principles used throughout the code:

- On current CPU architectures, the slowest mathematical operations are divides, square roots, and trigonometric functions. Addition, subtraction, and multiplication are generally 10 to 50 times faster than those operations. Reducing the number of slow mathematical operations can help performance substantially; for example, instead of repeatedly dividing by some value  $v$ , we will often precompute the reciprocal  $1/v$  and multiply by that instead.
- The speed of CPUs continues to grow more quickly than the speed at which data can be loaded from main memory into the CPU. This means that waiting for values to be fetched from memory can be a major performance limitation. Organizing algorithms and data structures in ways that give good performance from memory caches can speed up program execution much more than reducing the total number of instructions executed. Section A.4 in Appendix A discusses general principles

Ray 73

`Vector3f` 60

for memory-efficient programming; these ideas are mostly applied in the ray intersection acceleration structures of Chapter 4 and the image map representation in Section 10.4.3, although they influence many of the design decisions throughout the system.

#### 1.6.4 THE BOOK WEB SITE

We created a companion Web site for this book, located at [pbrt.org](http://pbrt.org). The Web site includes the system’s source code, documentation, images rendered with pbrt, example scenes, errata, and links to a bug reporting system. We encourage you to visit the Web site and subscribe to the pbrt mailing list.

#### 1.6.5 EXTENDING THE SYSTEM

One of our goals in writing this book and building the pbrt system was to make it easier for developers and researchers to experiment with new (or old!) ideas in rendering. One of the great joys in computer graphics is writing new software that makes a new image; even small changes to the system can be fun to experiment with. The exercises throughout the book suggest many changes to make to the system, ranging from small tweaks to major open-ended research projects. Section B.4 in Appendix B has more information about the mechanics of adding new implementations of the abstract base classes listed in Table 1.1.

#### 1.6.6 BUGS

Although we made every effort to make pbrt as correct as possible through extensive testing, it is inevitable that some bugs are still present.

If you believe you have found a bug in the system, please do the following:

1. Reproduce the bug with an unmodified copy of the latest version of pbrt.
2. Check the online discussion forum and the bug-tracking system at [pbrt.org](http://pbrt.org). Your issue may be a known bug, or it may be a commonly misunderstood feature.
3. Try to find the simplest possible test case that demonstrates the bug. Many bugs can be demonstrated by scene description files that are just a few lines long, and debugging is much easier with a simple scene than a complex one.
4. Submit a detailed bug report using our online bug-tracking system. Make sure that you include the scene file that demonstrates the bug and a detailed description of why you think pbrt is not behaving correctly with the scene. If you can provide a patch to the code that fixes the bug, all the better!

We will periodically release updated versions of pbrt with bug fixes and minor enhancements. (Be aware that we often let bug reports accumulate for a few months before going through them; don’t take this as an indication that we don’t value them!) However, we will not make major changes to the pbrt source code so that it doesn’t diverge from the system described here in the book.

## 1.7 A BRIEF HISTORY OF PHYSICALLY BASED RENDERING

Through the early years of computer graphics in the 1970s, the most important problems to solve were fundamental issues like visibility algorithms and geometric representations. When a megabyte of RAM was a rare and expensive luxury and when a computer capable of a million floating-point operations per second cost hundreds of thousands of dollars, the complexity of what was possible in computer graphics was correspondingly limited, and any attempt to accurately simulate physics for rendering was infeasible.

As computers have become more capable and less expensive, it became possible to consider more computationally demanding approaches to rendering, which in turn has made physically based approaches viable. This progression is neatly explained by *Blinn's law*: "as technology advances, rendering time remains constant."

Jim Blinn's simple statement captures an important constraint: given a certain number of images that must be rendered (be it a handful for a research paper or over a hundred thousand for a feature film), it's only possible to take so much processing time for each one. One has a certain amount of computation available and one has some amount of time available before rendering must be finished, so the maximum computation per image is necessarily limited.

Blinn's law also expresses the observation that there remains a gap between the images people would like to be able to render and the images that they can render: as computers have become faster, content creators have continued to be use increased computational capability to render more complex scenes with more sophisticated rendering algorithms, rather than rendering the same scenes as before, just more quickly. Rendering continues to consume all of the computational capabilities made available to it.

### 1.7.1 RESEARCH

Physically based approaches to rendering started to be seriously considered by graphics researchers in the 1980s. Whitted's paper (1980) introduced the idea of using ray tracing for global lighting effects, opening the door to accurately simulating the distribution of light in scenes. The rendered images his approach produced were markedly different from any that had been seen before, which spurred excitement about this approach.

Another notable early advancement in physically based rendering was Cook and Torrance's reflection model (1981, 1982), which introduced microfacet reflection models to graphics. Among other contributions, they showed that accurately modeling microfacet reflection made it possible to render metal surfaces accurately; metal was not well rendered by earlier approaches.

Shortly afterward, Goral et al. (1984) made connections between the thermal transfer literature and rendering, showing how to incorporate global diffuse lighting effects using a physically based approximation of light transport. This method was based on finite-element methods, where areas of surfaces in the scene exchanged energy with each other. This approach came to be referred to as "radiosity," after a related physical unit. Following work by Cohen and Greenberg (1985) and Nishita and Nakamae (1985) introduced important improvements. Once again, a physically based approach led to images with

lighting effects that hadn't previously been seen in rendered images, which led to many researchers pursuing improvements in this area.

While the radiosity approach was strongly based on physical units and conservation of energy, in time it became clear that it didn't lead to viable rendering algorithms: the asymptotic computational complexity was a difficult-to-manage  $O(n^2)$ , and it was necessary to be able to re-tessellate geometric models along shadow boundaries for good results; researchers had difficulty developing robust and efficient tessellation algorithms for this purpose and radiosity's adoption in practice was limited.

During the radiosity years, a small group of researchers pursued physically based approaches to rendering that were based on ray tracing and Monte Carlo integration. At the time, many looked at their work with skepticism; objectionable noise in images due to variance from Monte Carlo integration seemed unavoidable, while radiosity-based methods quickly gave visually pleasing results, at least on relatively simple scenes.

In 1984, Cook, Porter, and Carpenter introduced distributed ray tracing, which generalized Whitted's algorithm to compute motion blur and defocus blur from cameras, blurry reflection from glossy surfaces, and illumination from area light sources (Cook et al. 1984), showing that ray tracing was capable of generating a host of important lighting effects.

Shortly afterward, Kajiya (1986) introduced path tracing; he set out a rigorous formulation of the rendering problem (the light transport integral equation) and showed how to apply Monte Carlo integration to solve it. This work required immense amounts of computation: to render a  $256 \times 256$  pixel image of two spheres with path tracing required 7 hours of computation on an IBM 4341 computer, which cost roughly \$280,000 when it was first released (Farmer 1981). With von Herzen, Kajiya also introduced the volume-rendering equation to graphics (Kajiya and von Herzen 1984); this equation rigorously describes the scattering of light in participating media.

Both Cook et al.'s and Kajiya's work once again led to images unlike any that had been seen before, demonstrating the value of physically based methods. In subsequent years, important work on Monte Carlo for realistic image synthesis was described in papers by Arvo and Kirk (1990) and Kirk and Arvo (1991). Shirley's Ph.D. dissertation (1990) and follow-on work by Shirley et al. (1996) were important contributions to Monte Carlo-based efforts. Hall's book, *Illumination and Color in Computer Generated Imagery*, (1989) is one of the first books to present rendering in a physically based framework, and Andrew Glassner's *Principles of Digital Image Synthesis* rigorously laid out foundations of the field (1995). Ward's *Radiance* rendering system was an early open source physically based rendering system, focused on lighting design (Ward 1994), and Slusallek's *Vision* renderer was deisgned to bridge the gap between physically based approaches and the then widely used RenderMan interface, which wasn't physically based (Slusallek 1996).

Following Torrance and Cook's work, much of the research in the Program of Computer Graphics at Cornell University investigated physically based approaches. The motivations for this work were summarized by Greenberg et al. (1997), who made a strong argument for a physically accurate rendering based on measurements of the material properties of real-world objects and on deep understanding of the human visual system.

A crucial step forward for physically based rendering was Veach’s work, described in detail in his dissertation (Veach 1997). Veach advanced key theoretical foundations of Monte Carlo rendering while also developing new algorithms like multiple importance sampling, bidirectional path tracing, and Metropolis light transport that greatly improved its efficiency. Using Blinn’s law as a guide, we believe that these significant improvements in efficiency were critical to practical adoption of these approaches.

Around this time, as computers became faster and more parallel, a number of researchers started pursuing real-time ray tracing; Wald, Slusallek, and Benthin wrote an influential paper that described a highly optimized ray tracer that was much more efficient than previous ray tracers (Wald et al. 2001b). Many subsequent papers introduced increasingly more efficient ray-tracing algorithms. Though most of this work wasn’t physically based, the results led to great progress in ray-tracing acceleration structures and performance of the geometric components of ray tracing. Because physically based rendering generally makes substantial use of ray tracing, this work has in turn had the same helpful effect as faster computers have, making it possible to render more complex scenes with physical approaches.

At this point, we’ll end our summary of the key steps in the research progress of physically based rendering; much more has been done. The “Further Reading” sections in all of the subsequent chapters of this book cover this work in detail.

### 1.7.2 PRODUCTION

With more capable computers in the 1980s, computer graphics could start to be used for animation and film production. Early examples include Jim Blinn’s rendering of the Voyager 2 Flyby of Saturn in 1981 and visual effects in the movies *Star Trek II: The Wrath of Khan* (1982), *Tron* (1982), and *The Last Starfighter* (1984).

In early production use of computer-generated imagery, rasterization-based rendering (notably, the Reyes algorithm (Cook et al. 1987)) was the only viable option. One reason was that not enough computation was available for complex reflection models or for the global lighting effects that physically based ray tracing could provide. More significantly, rasterization had the important advantage that it didn’t require that the entire scene representation fit into main memory.

When RAM was much less plentiful, almost any interesting scene was too large to fit into main memory. Rasterization-based algorithms made it possible to render scenes while having only a small subset of the full scene representation in memory at any time. Global lighting effects are difficult to achieve if the whole scene can’t fit into main memory; for many years, with limited computer systems, content creators effectively decided that geometric and texture complexity was more important to visual realism than lighting complexity (and in turn physical accuracy).

Many practitioners at this time also believed that physically based approaches were undesirable for production: one of the great things about computer graphics is that one can cheat reality with impunity to achieve a desired artistic effect. For example, lighting designers on regular movies often struggle to place light sources so that they aren’t visible to the camera or spend a lot of effort placing a light to illuminate an actor without shin-

ing too much light on the background. Computer graphics offers the opportunity to, for example, implement a light source model that shines twice as much light on a character as on a background object, in a fairly straightforward manner. For many years, this capability seemed much more useful than physical accuracy.

Visual effects practitioners who had the specific need to match rendered imagery to filmed real-world environments pioneered capturing real-world lighting and shading effects and were early adopters of physically based approaches in the late 1990s and early 2000s. (See Snow (2010) for a history of ILM's early work in this area, for example.)

During this time, the Blue Sky studio adopted a physically based pipeline early in their history (Ohmer 1997). The photorealism of an advertisement they made for a Braun shaver in 1992 caught the attention of many, and their short film, *Bunny*, shown in 1998, was an early example of Monte Carlo global illumination used in production. Its visual look was substantially different from those of films and shorts rendered with Reyes and was widely noted. Subsequent feature films from Blue Sky also followed this approach. Unfortunately, Blue Sky never published significant technical details of their approach, limiting their wider influence.

During the early 2000s, the *mental ray* ray-tracing system was used by a number of studios, mostly for visual effects. It was a very efficient ray tracer with sophisticated global illumination algorithm implementations. The main focus of its developers was computer-aided design and product design applications, so it lacked features like the ability to handle extremely complex scenes and the enormous numbers of texture maps that film production demanded.

After *Bunny*, another watershed moment came in 2001, when Marcos Fajardo came to the SIGGRAPH with an early version of his *Arnold* renderer. He showed images in the Monte Carlo image synthesis course that not only had complex geometry, textures, and global illumination but also were rendered in tens of minutes. While these scenes weren't of the complexity of those used in film production at the time, his results showed many of the creative opportunities from global illumination in complex scenes.

Fajardo brought *Arnold* to Sony Pictures Imageworks, where work started to transform it to a production-capable physically based rendering system. Work on efficient motion blur, programmable shading, support for massively complex scenes and deferred loading of scene geometry, and support for texture caching, where only a small subset of the texture in the scene is kept in memory, were all important areas to be addressed. *Arnold* was first used on the movie *Monster House* and is now generally available as a product.

In the mid-2000s, Pixar's RenderMan renderer started to support hybrid rasterization and ray-tracing algorithms and included a number of innovative algorithms for computing global illumination solutions in complex scenes. RenderMan was recently rewritten to be a physically based ray tracer, following the general system architecture of pbrt (Christensen 2015).

One of the main reasons that physically based Monte Carlo approaches to rendering have been successful in production is that they end up improving the productivity of artists. Some of the important factors have been:



**Figure 1.22:** *Gravity* (2013) featured spectacular computer-generated imagery of a realistic space environment with volumetric scattering and large numbers of anisotropic metal surfaces. The image was generated using Arnold, a physically based rendering system that accounts for global illumination. Image courtesy of Warner Bros. and Framestore.

- The algorithms involved have essentially just a single quality knob: how many samples to take per pixel; this is extremely helpful for artists. Ray-tracing algorithms are also suited to both progressive refinement and quickly computing rough previews by taking just a few samples per pixel; rasterization-based renderers don't have equivalent capabilities.
- Adopting physically based reflection models has made it easier to design surface materials. Earlier, when reflection models that didn't necessarily conserve energy were used, an object might be placed in a single lighting environment while its surface reflection parameters were adjusted. The object might look great in that environment, but it would often appear completely wrong when moved to another lighting environment because surfaces were actually reflecting too little or too much energy: surface properties had been set to unreasonable values.
- The quality of shadows computed with ray tracing is much better than it is with rasterization. Eliminating the need to tweak shadow map resolutions, biases, and other parameters has eliminated an unpleasant task of lighting artists. Further, physically based methods bring with them bounce lighting and other soft-lighting effects from the method itself, rather than as an artistically tuned manual process.

As of this writing, physically based rendering is used widely for producing computer-generated imagery for movies; Figures 1.22 and 1.23 show images from two recent movies that used physically based approaches.



**Figure 1.23:** This image from *The Hobbit: The Desolation of Smaug* (2013) was also rendered using a physically based rendering system; the characters feature heterogeneous subsurface scattering and vast amounts of geometric detail. Image by Weta Digital, courtesy of Warner Bros. and Metro-Goldwyn-Mayer.

## FURTHER READING

In a seminal early paper, Arthur Appel (1968) first described the basic idea of ray tracing to solve the hidden surface problem and to compute shadows in polygonal scenes. Goldstein and Nagel (1971) later showed how ray tracing could be used to render scenes with quadric surfaces. Kay and Greenberg (1979) described a ray-tracing approach to rendering transparency, and Whitted's seminal *CACM* article described the general recursive ray-tracing algorithm that is implemented in this chapter, accurately simulating reflection and refraction from specular surfaces and shadows from point light sources (Whitted 1980). Heckbert (1987) was the first to explore realistic rendering of dessert.

Notable early books on physically based rendering and image synthesis include Cohen and Wallace's *Radiosity and Realistic Image Synthesis* (1993), Sillion and Puech's *Radiosity and Global Illumination* (1994), and Ashdown's *Radiosity: A Programmer's Perspective* (1994), all of which primarily describe the finite-element radiosity method.

In a paper on ray-tracing system design, Kirk and Arvo (1988) suggested many principles that have now become classic in renderer design. Their renderer was implemented as a core kernel that encapsulated the basic rendering algorithms and interacted with primitives and shading routines via a carefully constructed object-oriented interface. This approach made it easy to extend the system with new primitives and acceleration methods. pbrt's design is based on these ideas.

Another good reference on ray-tracer design is *Introduction to Ray Tracing* (Glassner 1989a), which describes the state of the art in ray tracing at that time and has a chapter by Heckbert that sketches the design of a basic ray tracer. More recently, Shirley and Morley's *Realistic Ray Tracing* (2003) gives an easy-to-understand introduction to ray

tracing and includes the complete source code to a basic ray tracer. Suffern’s book (2007) also provides a gentle introduction to ray tracing.

Researchers at Cornell University have developed a rendering testbed over many years; its design and overall structure were described by Trumbore, Lytle, and Greenberg (1993). Its predecessor was described by Hall and Greenberg (1983). This system is a loosely coupled set of modules and libraries, each designed to handle a single task (ray–object intersection acceleration, image storage, etc.) and written in a way that makes it easy to combine appropriate modules to investigate and develop new rendering algorithms. This testbed has been quite successful, serving as the foundation for much of the rendering research done at Cornell.

*Radiance* was the first widely available open source renderer based fundamentally on physical quantities. It was designed to perform accurate lighting simulation for architectural design. Ward described its design and history in a paper and a book (Ward 1994; Larson and Shakespeare 1998). *Radiance* is designed in the UNIX style, as a set of interacting programs, each handling a different part of the rendering process. This general type of rendering architecture was first described by Duff (1985).

Glassner’s (1993) *Spectrum* rendering architecture also focuses on physically based rendering, approached through a signal-processing-based formulation of the problem. It is an extensible system built with a plug-in architecture; pbrt’s approach of using parameter/value lists for initializing implementations of the main abstract interfaces is similar to *Spectrum*’s. One notable feature of *Spectrum* is that all parameters that describe the scene can be functions of time.

Slusallek and Seidel (1995, 1996; Slusallek 1996) described the *Vision* rendering system, which is also physically based and designed to support a wide variety of light transport algorithms. In particular, it had the ambitious goal of supporting both Monte Carlo and finite-element-based light transport algorithms.

Many papers have been written that describe the design and implementation of other rendering systems, including renderers for entertainment and artistic applications. The Reyes architecture, which forms the basis for Pixar’s RenderMan renderer, was first described by Cook et al. (1987), and a number of improvements to the original algorithm have been summarized by Apodaca and Gritz (2000). Gritz and Hahn (1996) described the *BMRT* ray tracer. The renderer in the Maya modeling and animation system was described by Sung et al. (1998), and some of the internal structure of the *mental ray* renderer is described in Driemeyer and Herken’s book on its API (Driemeyer and Herken 2002). The design of the high-performance *Manta* interactive ray tracer is described by Bigler et al. (2006).

The source code to pbrt is licensed under the BSD License; this has made it possible for other developers to use pbrt code as a basis for their efforts. *LuxRender*, available from [www.luxrender.net](http://www.luxrender.net), is a physically based renderer built using pbrt as a starting point; it offers a number of additional features and has a rich set of scene export plugins for modeling systems.

*Ray Tracing News*, an electronic newsletter compiled by Eric Haines dates to 1987 and is occasionally still published. It’s a very good resource for general ray-tracing informa-

tion and has particularly useful discussions about intersection acceleration approaches, implementation issues, and tricks of the trade. More recently, the forums at [ompf2.com](http://ompf2.com) have been frequented by many experienced ray-tracer developers.

The object-oriented approach used to structure pbrt makes the system easy to understand but is not the only way to structure rendering systems. An important counterpoint to the object-oriented approach is *data-oriented design* (DoD), a way of programming that has notably been advocated by a number of game developers (for whom performance is critical). The key tenet behind DoD is that many principles of traditional object-oriented design are incompatible with high-performance software systems as they lead to cache-inefficient layout of data in memory. Instead, its proponents argue for driving system design first from considerations of the layout of data in memory and how those data are transformed by the program. See, for example, Mike Acton's keynote at the C++ Conference (Acton 2014).

## EXERCISE

- ① 1.1 A good way to gain an understanding of pbrt is to follow the process of computing the radiance value for a single ray in a debugger. Build a version of pbrt with debugging symbols and set up your debugger to run pbrt with the `killeroo-simple.pbrt` scene from the `scenes` directory. Set breakpoints in the `SamplerIntegrator::Render()` method and trace through the process of how a ray is generated, how its radiance value is computed, and how its contribution is added to the image. The first time you do this, you may want to specify that only a single thread of execution should be used by providing `--nthreads 1` as command-line arguments to pbrt; doing so ensures that all computation is done in the main processing thread, which may make it easier to understand what is going on, depending on how easy your debugger makes it to step through the program when it is running multiple threads.

As you gain more understanding about the details of the system later in the book, repeat this process and trace through particular parts of the system more carefully.



# CHAPTER TWO

---

# 02 GEOMETRY AND TRANSFORMATIONS

Almost all nontrivial graphics programs are built on a foundation of geometric classes. These classes represent mathematical constructs like points, vectors, and rays. Because these classes are ubiquitous throughout the system, good abstractions and efficient implementations are critical. This chapter presents the interface to and implementation of pbrt’s geometric foundation. Note that these are not the classes that represent the actual scene geometry (triangles, spheres, etc.); those classes are the topic of Chapter 3.

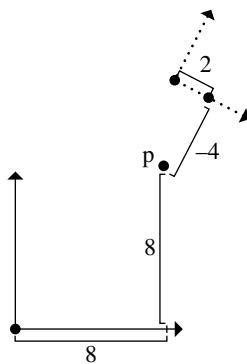
The geometric classes in this chapter are defined in the files `core/geometry.h` and `core/geometry.cpp` in the pbrt distribution, and the implementations of transformation matrices (Section 2.7) are in the files `core/transform.h` and `core/transform.cpp`.

## 2.1 COORDINATE SYSTEMS

As is typical in computer graphics, pbrt represents three-dimensional points, vectors, and normal vectors with three coordinate values:  $x$ ,  $y$ , and  $z$ . These values are meaningless without a *coordinate system* that defines the origin of the space and gives three linearly independent vectors that define the  $x$ ,  $y$ , and  $z$  axes of the space. Together, the origin and three vectors are called the *frame* that defines the coordinate system. Given an arbitrary point or direction in 3D, its  $(x, y, z)$  coordinate values depend on its relationship to the frame. Figure 2.1 shows an example that illustrates this idea in 2D.

In the general  $n$ -dimensional case, a frame’s origin  $p_o$  and its  $n$  linearly independent basis vectors define an  $n$ -dimensional *affine space*. All vectors  $v$  in the space can be expressed as a linear combination of the basis vectors. Given a vector  $v$  and the basis vectors  $v_i$ , there is a unique set of scalar values  $s_i$  such that

$$v = s_1 v_1 + \cdots + s_n v_n.$$



**Figure 2.1:** In 2D, the  $(x, y)$  coordinates of a point  $p$  are defined by the relationship of the point to a particular 2D coordinate system. Here, two coordinate systems are shown; the point might have coordinates  $(8, 8)$  with respect to the coordinate system with its coordinate axes drawn in solid lines but have coordinates  $(2, -4)$  with respect to the coordinate system with dashed axes. In either case, the 2D point  $p$  is at the same absolute position in space.

The scalars  $s_i$  are the *representation* of  $\mathbf{v}$  with respect to the basis  $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$  and are the coordinate values that we store with the vector. Similarly, for all points  $p$ , there are unique scalars  $s_i$  such that the point can be expressed in terms of the origin  $p_o$  and the basis vectors

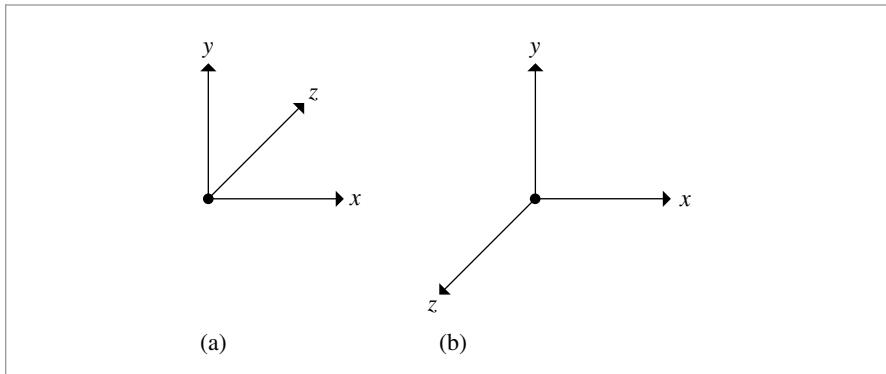
$$\mathbf{p} = \mathbf{p}_o + s_1 \mathbf{v}_1 + \cdots + s_n \mathbf{v}_n.$$

Thus, although points and vectors are both represented by  $x$ ,  $y$ , and  $z$  coordinates in 3D, they are distinct mathematical entities and are not freely interchangeable.

This definition of points and vectors in terms of coordinate systems reveals a paradox: to define a frame we need a point and a set of vectors, but we can only meaningfully talk about points and vectors with respect to a particular frame. Therefore, in three dimensions we need a *standard frame* with origin  $(0, 0, 0)$  and basis vectors  $(1, 0, 0)$ ,  $(0, 1, 0)$ , and  $(0, 0, 1)$ . All other frames will be defined with respect to this canonical coordinate system, which we call *world space*.

### 2.1.1 COORDINATE SYSTEM HANDEDNESS

There are two different ways that the three coordinate axes can be arranged, as shown in Figure 2.2. Given perpendicular  $x$  and  $y$  coordinate axes, the  $z$  axis can point in one of two directions. These two choices are called *left-handed* and *right-handed*. The choice between the two is arbitrary but has a number of implications for how some of the geometric operations throughout the system are implemented. pbrt uses a left-handed coordinate system.



**Figure 2.2:** (a) In a left-handed coordinate system, the *z* axis points into the page when the *x* and *y* axes are oriented with *x* pointing to the right and *y* pointing up. (b) In a right-handed system, the *z* axis points out of the page.

## 2.2 VECTORS

pbrt provides both 2D and 3D vector classes. Both are parameterized by the type of the underlying vector element, thus making it easy to instantiate vectors of both integer and floating-point types.

```

⟨Vector Declarations⟩ ≡
    template <typename T> class Vector2 {
        public:
            ⟨Vector2 Public Methods⟩
            ⟨Vector2 Public Data 59⟩
    };
}

⟨Vector Declarations⟩ +≡
    template <typename T> class Vector3 {
        public:
            ⟨Vector3 Public Methods 60⟩
            ⟨Vector3 Public Data 59⟩
    };
}

```

In the following, we will generally only include implementations of `Vector3` methods; all have `Vector2` parallels that have straightforward implementation differences.

A vectors is represented with a tuple of components that gives its representation in terms of the *x*, *y*, *z* (in 3D) axes of the space it is defined in. The individual components of a 3D vector  $\mathbf{v}$  will be written  $v_x$ ,  $v_y$ , and  $v_z$ .

```

⟨Vector2 Public Data⟩ ≡
    T x, y;

```

```

⟨Vector3 Public Data⟩ ≡
    T x, y, z;

```

An alternate implementation would be to have a single template class that is also parameterized with an integer number of dimensions and to represent the coordinates with an array of that many  $T$  values. While this approach would reduce the total amount of code, individual components of the vector couldn't be accessed as  $v.x$  and so forth. We believe that in this case, a bit more code in the vector implementations is worthwhile in return for more transparent access to elements.

However, some routines do find it useful to be able to easily loop over the components of vectors; the vector classes also provide a C++ operator to index into the components so that, given a vector  $v$ ,  $v[0] == v.x$  and so forth.

*(Vector3 Public Methods)  $\equiv$*

59

```
T operator[](int i) const {
    Assert(i >= 0 && i <= 2);
    if (i == 0) return x;
    if (i == 1) return y;
    return z;
}
T &operator[](int i) {
    Assert(i >= 0 && i <= 2);
    if (i == 0) return x;
    if (i == 1) return y;
    return z;
}
```

For convenience, a number of widely used types of vectors are given a `typedef`, so that they have more concise names in code elsewhere.

*(Vector Declarations)  $+ \equiv$*

```
typedef Vector2<Float> Vector2f;
typedef Vector2<int> Vector2i;
typedef Vector3<Float> Vector3f;
typedef Vector3<int> Vector3i;
```

Readers who have been exposed to object-oriented design may question our decision to make the vector element data publicly accessible. Typically, data members are only accessible inside their class, and external code that wishes to access or modify the contents of a class must do so through a well-defined API of selector and mutator functions. Although we generally agree with this design principle (though see the discussion of data-oriented design in the “Further Reading” section of Chapter 1), it is not appropriate here. The purpose of selector and mutator functions is to hide the class’s internal implementation details. In the case of vectors, hiding this basic part of their design gains nothing and adds bulk to code that uses them.

By default, the  $(x, y, z)$  values are set to zero, although the user of the class can optionally supply values for each of the components. If the user does supply values, we check that none of them has the floating-point “not a number” (NaN) value using the `Assert()` macro. When compiled in optimized mode, this macro disappears from the compiled code, saving the expense of verifying this case. NaNs almost certainly indicate a bug in the system; if a NaN is generated by some computation, we’d like to catch it as soon as

Assert() 1069

Float 1062

Vector2 59

Vector3 59

possible in order to make isolating its source easier. (See Section 3.9.1 for more discussion of NaN values.)

```
<Vector3 Public Methods> +≡
    Vector3() { x = y = z = 0; }
    Vector3(T x, T y, T z)
        : x(x), y(y), z(z) {
            Assert(!HasNaNs());
    }
```

59

The code to check for NaNs just calls the `std::isnan()` function on each of the  $x$ ,  $y$ , and  $z$  components.

```
<Vector3 Public Methods> +≡
    bool HasNaNs() const {
        return std::isnan(x) || std::isnan(y) || std::isnan(z);
    }
```

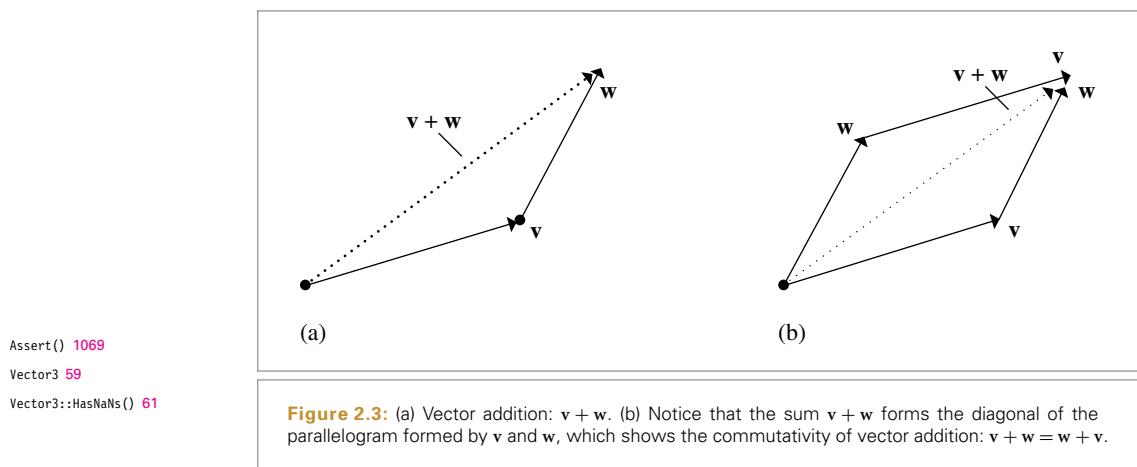
59

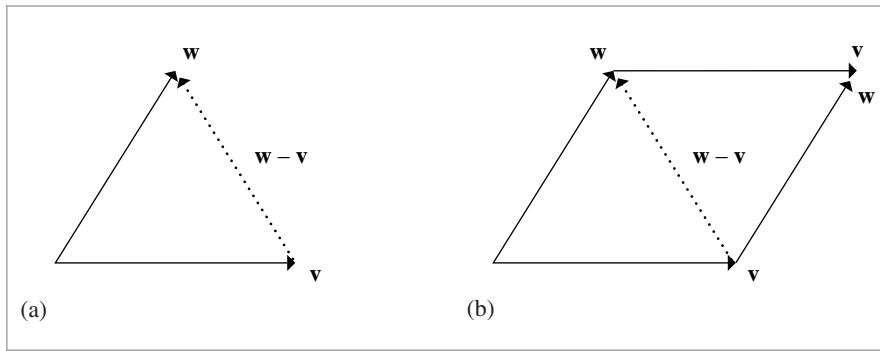
Addition and subtraction of vectors are done component-wise. The usual geometric interpretation of vector addition and subtraction is shown in Figures 2.3 and 2.4.

```
<Vector3 Public Methods> +≡
    Vector3<T> operator+(const Vector3<T> &v) const {
        return Vector3(x + v.x, y + v.y, z + v.z);
    }
    Vector3<T>& operator+=(const Vector3<T> &v) {
        x += v.x; y += v.y; z += v.z;
        return *this;
    }
```

59

The code for subtracting two vectors is similar and therefore not shown here.





**Figure 2.4:** (a) Vector subtraction. (b) The difference  $-v - w$  is the other diagonal of the parallelogram formed by  $v$  and  $w$ .

A vector can be multiplied component-wise by a scalar, thereby changing its length. Three functions are needed in order to cover all of the different ways that this operation may be written in source code (i.e.,  $v*s$ ,  $s*v$ , and  $v *= s$ ):

*(Vector3 Public Methods)*  $\equiv$

59

```
Vector3<T> operator*(T s) const { return Vector3<T>(s*x, s*y, s*z); }
Vector3<T> &operator*=(T s) {
    x *= s; y *= s; z *= s;
    return *this;
}
```

*(Geometry Inline Functions)*  $\equiv$

```
template <typename T> inline Vector3<T>
operator*(T s, const Vector3<T> &v) { return v * s; }
```

Similarly, a vector can be divided component-wise by a scalar. The code for scalar division is similar to scalar multiplication, although division of a scalar by a vector is not well defined and so is not permitted.

In the implementation of these methods, we use a single division to compute the scalar's reciprocal and then perform three component-wise multiplications. This is a useful trick for avoiding division operations, which are generally much slower than multiplies on modern CPUs.<sup>1</sup>

We use the `Assert()` macro to make sure that the provided divisor is not zero; this should never happen and would indicate a bug elsewhere in the system.

<sup>1</sup> It is a common misconception that these sorts of optimizations are unnecessary because the compiler will perform the necessary analysis. Compilers are generally restricted from performing many transformations of this type. For division, the IEEE floating-point standard requires that  $x/x = 1$  for all  $x$ , but if we compute  $1/x$  and store it in a variable and then multiply  $x$  by that value, it is not guaranteed that  $1$  will be the result. In this case, we are willing to lose that guarantee in exchange for higher performance. See Section 3.9 for more discussion of these issues.

Assert() 1069

Vector3 59

```
(Vector3 Public Methods) +≡
    Vector3<T> operator/(T f) const {
        Assert(f != 0);
        Float inv = (Float)1 / f;
        return Vector3<T>(x * inv, y * inv, z * inv);
    }

    Vector3<T> &operator/=(T f) {
        Assert(f != 0);
        Float inv = (Float)1 / f;
        x *= inv; y *= inv; z *= inv;
        return *this;
    }
```

The `Vector3` class also provides a unary negation operator that returns a new vector pointing in the opposite direction of the original one:

```
(Vector3 Public Methods) +≡
    Vector3<T> operator-() const { return Vector3<T>(-x, -y, -z); }
```

Finally, `Abs()` returns a vector with the absolute value operation applied to its components.

```
(Geometry Inline Functions) +≡
    template <typename T> Vector3<T> Abs(const Vector3<T> &v) {
        return Vector3<T>(std::abs(v.x), std::abs(v.y), std::abs(v.z));
    }
```

## 2.2.1 DOT AND CROSS PRODUCT

Two useful operations on vectors are the dot product (also known as the scalar or inner product) and the cross product. For two vectors  $\mathbf{v}$  and  $\mathbf{w}$ , their *dot product* ( $\mathbf{v} \cdot \mathbf{w}$ ) is defined as:

$$v_x w_x + v_y w_y + v_z w_z.$$

```
(Geometry Inline Functions) +≡
    template <typename T> inline T
    Dot(const Vector3<T> &v1, const Vector3<T> &v2) {
        return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
    }
```

The dot product has a simple relationship to the angle between the two vectors:

$$(\mathbf{v} \cdot \mathbf{w}) = \|\mathbf{v}\| \|\mathbf{w}\| \cos \theta, \quad [2.1]$$

Assert() 1069  
Float 1062  
Vector3 59

where  $\theta$  is the angle between  $\mathbf{v}$  and  $\mathbf{w}$ , and  $\|\mathbf{v}\|$  denotes the length of the vector  $\mathbf{v}$ . It follows from this that  $(\mathbf{v} \cdot \mathbf{w})$  is zero if and only if  $\mathbf{v}$  and  $\mathbf{w}$  are perpendicular, provided that neither  $\mathbf{v}$  nor  $\mathbf{w}$  is *degenerate*—equal to  $(0, 0, 0)$ . A set of two or more mutually perpendicular vectors is said to be *orthogonal*. An orthogonal set of unit vectors is called *orthonormal*.

It immediately follows from Equation (2.1) that if  $\mathbf{v}$  and  $\mathbf{w}$  are unit vectors, their dot product is the cosine of the angle between them. As the cosine of the angle between two vectors often needs to be computed for rendering, we will frequently make use of this property.

A few basic properties directly follow from the definition. For example, if  $\mathbf{u}$ ,  $\mathbf{v}$ , and  $\mathbf{w}$  are vectors and  $s$  is a scalar value, then:

$$\begin{aligned}(\mathbf{u} \cdot \mathbf{v}) &= (\mathbf{v} \cdot \mathbf{u}) \\(s\mathbf{u} \cdot \mathbf{v}) &= s(\mathbf{v} \cdot \mathbf{u}) \\(\mathbf{u} \cdot (\mathbf{v} + \mathbf{w})) &= (\mathbf{u} \cdot \mathbf{v}) + (\mathbf{u} \cdot \mathbf{w}).\end{aligned}$$

We will frequently need to compute the absolute value of the dot product as well. The `AbsDot()` function does this for us so that a separate call to `std::abs()` isn't necessary.

*(Geometry Inline Functions)*  $\equiv$

```
template <typename T>
inline T AbsDot(const Vector3<T> &v1, const Vector3<T> &v2) {
    return std::abs(Dot(v1, v2));
}
```

The *cross product* is another useful operation for 3D vectors. Given two vectors in 3D, the cross product  $\mathbf{v} \times \mathbf{w}$  is a vector that is perpendicular to both of them. Given orthogonal vectors  $\mathbf{v}$  and  $\mathbf{w}$ , then  $\mathbf{v} \times \mathbf{w}$  is defined to be a vector such that  $(\mathbf{v}, \mathbf{w}, \mathbf{v} \times \mathbf{w})$  form a coordinate system.

The cross product is defined as:

$$\begin{aligned}(\mathbf{v} \times \mathbf{w})_x &= \mathbf{v}_y \mathbf{w}_z - \mathbf{v}_z \mathbf{w}_y \\(\mathbf{v} \times \mathbf{w})_y &= \mathbf{v}_z \mathbf{w}_x - \mathbf{v}_x \mathbf{w}_z \\(\mathbf{v} \times \mathbf{w})_z &= \mathbf{v}_x \mathbf{w}_y - \mathbf{v}_y \mathbf{w}_x.\end{aligned}$$

A way to remember this is to compute the determinant of the matrix:

$$\mathbf{v} \times \mathbf{w} = \begin{vmatrix} i & j & k \\ \mathbf{v}_x & \mathbf{v}_y & \mathbf{v}_z \\ \mathbf{w}_x & \mathbf{w}_y & \mathbf{w}_z \end{vmatrix},$$

where  $i$ ,  $j$ , and  $k$  represent the axes  $(1, 0, 0)$ ,  $(0, 1, 0)$ , and  $(0, 0, 1)$ , respectively. Note that this equation is merely a memory aid and not a rigorous mathematical construction, since the matrix entries are a mix of scalars and vectors.

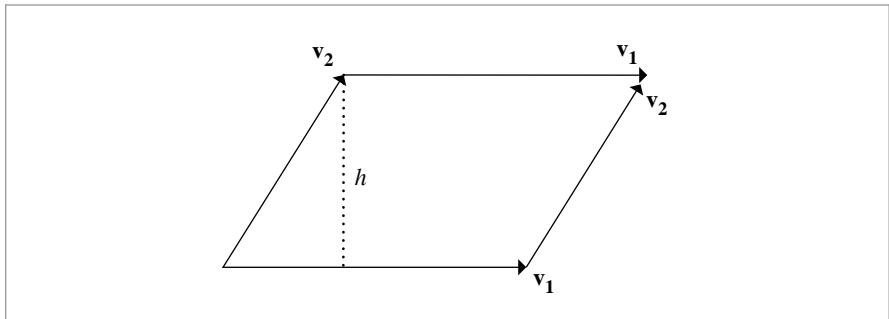
In the implementation here, the vector elements are converted to double-precision (regardless of the type of `Float`) before the subtractions in the `Cross()` function. Using extra precision for 32-bit floating-point values here protects against error from catastrophic cancellation, a type of floating-point error that can happen when subtracting two values that are very close together. This isn't a theoretical concern: this change was necessary to fix bugs that came up from this issue previously. See Section 3.9 for more information on floating-point rounding error.

`AbsDot()` 64

`Dot()` 63

`Float` 1062

`Vector3` 59



**Figure 2.5:** The area of a parallelogram with edges given by vectors  $v_1$  and  $v_2$  is equal to  $v_1 h$ . From Equation (2.2), the length of the cross product of  $v_1$  and  $v_2$  is equal to the product of the two vector lengths times the sine of the angle between them—the parallelogram area.

(*Geometry Inline Functions*)  $\equiv$

```
template <typename T> inline Vector3<T>
Cross(const Vector3<T> &v1, const Vector3<T> &v2) {
    double v1x = v1.x, v1y = v1.y, v1z = v1.z;
    double v2x = v2.x, v2y = v2.y, v2z = v2.z;
    return Vector3<T>((v1y * v2z) - (v1z * v2y),
                        (v1z * v2x) - (v1x * v2z),
                        (v1x * v2y) - (v1y * v2x));
}
```

From the definition of the cross product, we can derive

$$\|v \times w\| = \|v\| \|w\| \sin \theta, \quad (2.2)$$

where  $\theta$  is the angle between  $v$  and  $w$ . An important implication of this is that the cross product of two perpendicular unit vectors is itself a unit vector. Note also that the result of the cross product is a degenerate vector if  $v$  and  $w$  are parallel.

This definition also shows a convenient way to compute the area of a parallelogram (Figure 2.5). If the two edges of the parallelogram are given by vectors  $v_1$  and  $v_2$ , and it has height  $h$ , the area is given by  $\|v_1\| h$ . Since  $h = \sin \theta \|v_2\|$ , we can use Equation (2.2) to see that the area is  $\|v_1 \times v_2\|$ .

## 2.2.2 NORMALIZATION

It is often necessary to *normalize* a vector—that is, to compute a new vector pointing in the same direction but with unit length. A normalized vector is often called a *unit vector*. The notation used in this book for normalized vectors is that  $\hat{v}$  is the normalized version of  $v$ . To normalize a vector, it's first useful to be able to compute its length.

Float 1062

Vector3 59

(*Vector3 Public Methods*)  $\equiv$

```
Float LengthSquared() const { return x * x + y * y + z * z; }
Float Length() const { return std::sqrt(LengthSquared()); }
```

59

`Normalize()` normalizes a vector. It divides each component by the length of the vector,  $\|v\|$ . It returns a new vector; it does *not* normalize the vector in place:

```
(Geometry Inline Functions) +≡
template <typename T> inline Vector3<T>
Normalize(const Vector3<T> &v) { return v / v.Length(); }
```

### 2.2.3 MISCELLANEOUS OPERATIONS

A few additional operations are useful when working with vectors. The `MinComponent()` and `MaxComponent()` methods return the smallest and largest coordinate value, respectively.

```
(Geometry Inline Functions) +≡
template <typename T> T
MinComponent(const Vector3<T> &v) {
    return std::min(v.x, std::min(v.y, v.z));
}
template <typename T> T
MaxComponent(const Vector3<T> &v) {
    return std::max(v.x, std::max(v.y, v.z));
}
```

Related, `MaxDimension()` returns the index of the component with the largest value.

```
(Geometry Inline Functions) +≡
template <typename T> int
MaxDimension(const Vector3<T> &v) {
    return (v.x > v.y) ? ((v.x > v.z) ? 0 : 2) :
        ((v.y > v.z) ? 1 : 2);
}
```

Component-wise minimum and maximum operations are also available.

```
(Geometry Inline Functions) +≡
template <typename T> Vector3<T>
Min(const Vector3<T> &p1, const Vector3<T> &p2) {
    return Vector3<T>(std::min(p1.x, p2.x), std::min(p1.y, p2.y),
                       std::min(p1.z, p2.z));
}
template <typename T> Vector3<T>
Max(const Vector3<T> &p1, const Vector3<T> &p2) {
    return Vector3<T>(std::max(p1.x, p2.x), std::max(p1.y, p2.y),
                       std::max(p1.z, p2.z));
}
```

Vector3 59

Finally, `Permute()` permutes the coordinate values according to the index values provided.

```
(Geometry Inline Functions) +≡
    template <typename T> Vector3<T>
    Permute(const Vector3<T> &v, int x, int y, int z) {
        return Vector3<T>(v[x], v[y], v[z]);
    }
```

## 2.2.4 COORDINATE SYSTEM FROM A VECTOR

We will frequently want to construct a local coordinate system given only a single 3D vector. Because the cross product of two vectors is orthogonal to both, we can apply the cross product two times to get a set of three orthogonal vectors for the coordinate system. Note that the two vectors generated by this technique are unique only up to a rotation about the given vector.

The implementation of this function assumes that the vector passed in,  $v_1$ , has already been normalized. It first constructs a perpendicular vector by zeroing one of the components of the original vector, swapping the remaining two, and negating one of them. Inspection of the two cases should make clear that  $v_2$  will be normalized and that the dot product ( $v_1 \cdot v_2$ ) must be equal to zero. Given these two perpendicular vectors, a single cross product gives the third, which by definition will be perpendicular to the first two.

```
(Geometry Inline Functions) +≡
    template <typename T> inline void
    CoordinateSystem(const Vector3<T> &v1, Vector3<T> *v2, Vector3<T> *v3) {
        if (std::abs(v1.x) > std::abs(v1.y))
            *v2 = Vector3<T>(-v1.z, 0, v1.x) /
                std::sqrt(v1.x * v1.x + v1.z * v1.z);
        else
            *v2 = Vector3<T>(0, v1.z, -v1.y) /
                std::sqrt(v1.y * v1.y + v1.z * v1.z);
        *v3 = Cross(v1, *v2);
    }
```

## 2.3 POINTS

A point is a zero-dimensional location in 2D or 3D space. The `Point2` and `Point3` classes in `pbrt` represent points in the obvious way: using  $x$ ,  $y$ ,  $z$  (in 3D) coordinates with respect to a coordinate system. Although the same representation is used for vectors, the fact that a point represents a position whereas a vector represents a direction leads to a number of important differences in how they are treated. Points are denoted in text by  $p$ .

In this section, we'll continue the approach of only including implementations of the 3D point methods in the `Point3` class here.

```
(Point Declarations) ≡
Cross() 65
Point3 68
Vector3 59
    template <typename T> class Point2 {
        public:
            (Point2 Public Methods 68)
            (Point2 Public Data 68)
    };
```

```
(Point Declarations) +≡
template <typename T> class Point3 {
public:
    (Point3 Public Methods 68)
    (Point3 Public Data 68)
};
```

As with vectors, it's helpful to have shorter type names for commonly used point types.

```
(Point Declarations) +≡
typedef Point2<Float> Point2f;
typedef Point2<int> Point2i;
typedef Point3<Float> Point3f;
typedef Point3<int> Point3i;
```

<i>(Point2 Public Data)</i> ≡	67
T x, y;	

<i>(Point3 Public Data)</i> ≡	68
T x, y, z;	

Also like vectors, a Point3 constructor takes parameters to set the x, y, and z coordinate values.

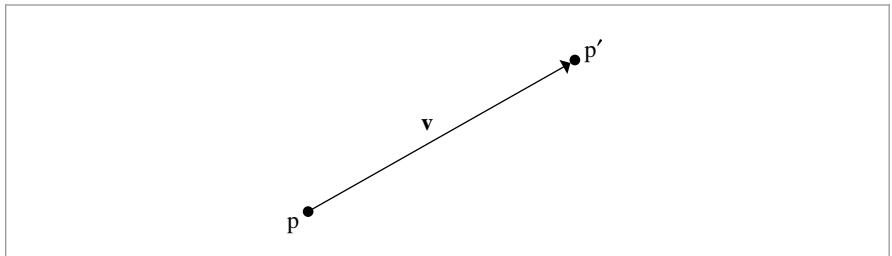
<i>(Point3 Public Methods)</i> ≡	68
Point3() { x = y = z = 0; }	
Point3(T x, T y, T z) : x(x), y(y), z(z) {	
Assert(!HasNaNs());	
}	

It can be useful to convert a Point3 to a Point2 by dropping the z coordinate. The constructor that does this conversion has the explicit qualifier so that this conversion can't happen without an explicit cast, lest it happen unintentionally.

<i>(Point2 Public Methods)</i> ≡	67
explicit Point2(const Point3<T> &p) : x(p.x), y(p.y) {	
Assert(!HasNaNs());	
}	

It's also useful to be able to convert a point with one element type (e.g., a Point3f) to a point of another one (e.g., Point3i) as well as to be able to convert a point to a vector with a different underlying element type. The following constructor and conversion operator provide these conversions. Both also require an explicit cast, to make it clear in source code when they are being used.

Float 1062  
Point2 67  
Point3 68



**Figure 2.6: Obtaining the Vector between Two Points.** The vector  $p' - p$  is given by the component-wise subtraction of the points  $p'$  and  $p$ .

*(Point3 Public Methods) +≡*

```
template <typename U> explicit Point3(const Point3<U> &p)
: x((T)p.x), y((T)p.y), z((T)p.z) {
    Assert(!HasNaNs());
}
template <typename U> explicit operator Vector3<U>() const {
    return Vector3<U>(x, y, z);
}
```

68

There are certain `Point3` methods that either return or take a `Vector3`. For instance, one can add a vector to a point, offsetting it in the given direction to obtain a new point.

*(Point3 Public Methods) +≡*

```
Point3<T> operator+(const Vector3<T> &v) const {
    return Point3<T>(x + v.x, y + v.y, z + v.z);
}
Point3<T> &operator+=(const Vector3<T> &v) {
    x += v.x; y += v.y; z += v.z;
    return *this;
}
```

68

Alternately, one can subtract one point from another, obtaining the vector between them, as shown in Figure 2.6.

*(Point3 Public Methods) +≡*

```
Vector3<T> operator-(const Point3<T> &p) const {
    return Vector3<T>(x - p.x, y - p.y, z - p.z);
}
Point3<T> operator-(const Vector3<T> &v) const {
    return Point3<T>(x - v.x, y - v.y, z - v.z);
}
```

68

Point3 68

Vector3 59

Subtracting a vector from a point gives a new point.

*(Point3 Public Methods) +≡*

```
Point3<T> &operator-=(const Vector3<T> &v) {
    x -= v.x; y -= v.y; z -= v.z;
    return *this;
}
```

68

The distance between two points can be computed by subtracting them to compute the vector between them and then finding the length of that vector:

*(Geometry Inline Functions) +≡*

```
template <typename T> inline Float
Distance(const Point3<T> &p1, const Point3<T> &p2) {
    return (p1 - p2).Length();
}

template <typename T> inline Float
DistanceSquared(const Point3<T> &p1, const Point3<T> &p2) {
    return (p1 - p2).LengthSquared();
}
```

Although in general it doesn't make sense mathematically to weight points by a scalar or add two points together, the point classes still allow these operations in order to be able to compute weighted sums of points, which is mathematically meaningful as long as the weights used all sum to one. The code for scalar multiplication and addition with points is identical to the corresponding code for vectors, so it is not shown here.

On a related note, it's useful to be able to linearly interpolate between two points. Lerp() returns p0 at t==0, p1 at t==1, and linearly interpolates between them at other values of t. For t<0 or t>1, Lerp() extrapolates.

*(Geometry Inline Functions) +≡*

```
template <typename T> Point3<T>
Lerp(Float t, const Point3<T> &p0, const Point3<T> &p1) {
    return (1 - t) * p0 + t * p1;
}
```

The Min() and Max() functions return points representing the component-wise minimums and maximums of the two given points.

*(Geometry Inline Functions) +≡*

```
template <typename T> Point3<T>
Min(const Point3<T> &p1, const Point3<T> &p2) {
    return Point3<T>(std::min(p1.x, p2.x), std::min(p1.y, p2.y),
                      std::min(p1.z, p2.z));
}

template <typename T> Point3<T>
Max(const Point3<T> &p1, const Point3<T> &p2) {
    return Point3<T>(std::max(p1.x, p2.x), std::max(p1.y, p2.y),
                      std::max(p1.z, p2.z));
}
```

Float 1062

Point3 68

Vector3 59

`Floor()`, `Ceil()`, and `Abs()` apply the corresponding operation component-wise to the given point.

```
(Geometry Inline Functions) +≡
template <typename T> Point3<T> Floor(const Point3<T> &p) {
    return Point3<T>(std::floor(p.x), std::floor(p.y), std::floor(p.z));
}
template <typename T> Point3<T> Ceil(const Point3<T> &p) {
    return Point3<T>(std::ceil(p.x), std::ceil(p.y), std::ceil(p.z));
}
template <typename T> Point3<T> Abs(const Point3<T> &p) {
    return Point3<T>(std::abs(p.x), std::abs(p.y), std::abs(p.z));
}
```

And finally, `Permute()` permutes the coordinate values according to the provided permutation.

```
(Geometry Inline Functions) +≡
template <typename T> Point3<T>
Permute(const Point3<T> &p, int x, int y, int z) {
    return Point3<T>(p[x], p[y], p[z]);
}
```

## 2.4 NORMALS

A *surface normal* (or just *normal*) is a vector that is perpendicular to a surface at a particular position. It can be defined as the cross product of any two nonparallel vectors that are tangent to the surface at a point. Although normals are superficially similar to vectors, it is important to distinguish between the two of them: because normals are defined in terms of their relationship to a particular surface, they behave differently than vectors in some situations, particularly when applying transformations. This difference is discussed in Section 2.8.

```
(Normal Declarations) ≡
template <typename T> class Normal3 {
public:
    (Normal3 Public Methods 72)
    (Normal3 Public Data)
};
```

```
(Normal Declarations) +≡
typedef Normal3<Float> Normal3f;
```

Float 1062  
Normal3 71  
Point3 68  
Vector3 59

The implementations of `Normal3s` and `Vector3s` are very similar. Like vectors, normals are represented by three components `x`, `y`, and `z`; they can be added and subtracted to compute new normals; and they can be scaled and normalized. However, a normal cannot be added to a point, and one cannot take the cross product of two normals. Note that, in an unfortunate turn of terminology, normals are *not* necessarily normalized.

`Normal3` provides an extra constructor that initializes a `Normal3` from a `Vector3`. Because `Normal3`s and `Vector3`s are different in subtle ways, we want to make sure that this conversion doesn't happen when we don't intend it to, so the C++ `explicit` keyword is again used here. `Vector3` also provides a constructor that converts the other way. Thus, given the declarations `Vector3f v;` and `Normal3f n;`, then the assignment `n = v` is illegal, so it is necessary to explicitly convert the vector, as in `n = Normal3f(v)`.

*(Normal3 Public Methods)  $\equiv$*  71

```
explicit Normal3<T>(const Vector3<T> &v) : x(v.x), y(v.y), z(v.z) {
    Assert(!v.HasNaNs());
}
```

*(Geometry Inline Functions)  $+ \equiv$*

```
template <typename T> inline
Vector3<T>::Vector3(const Normal3<T> &n) : x(n.x), y(n.y), z(n.z) {
    Assert(!n.HasNaNs());
}
```

The `Dot()` and `AbsDot()` functions are also overloaded to compute dot products between the various possible combinations of normals and vectors. This code won't be included in the text here. We also won't include implementations of all of the various other `Normal3` methods here, since they are similar to those for vectors.

One new operation to implement comes from the fact it's often necessary to flip a surface normal so that it lies in the same hemisphere as a given vector—for example, the surface normal that lies in the same hemisphere as an outgoing ray is frequently needed. The `Faceforward()` utility function encapsulates this small computation. (`pbrt` also provides variants of this function for the other three combinations of `Vector3`s and `Normal3`s as parameters.) Be careful when using the other instances, though: when using the version that takes two `Vector3`s, for example, ensure that the first parameter is the one that should be returned (possibly flipped) and the second is the one to test against. Reversing the two parameters will give unexpected results.

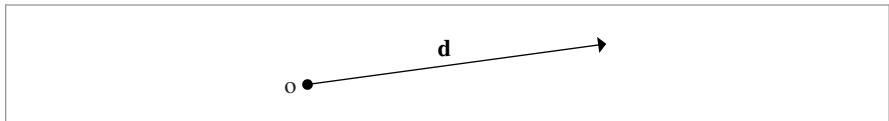
*(Geometry Inline Functions)  $+ \equiv$*

```
template <typename T> inline Normal3<T>
Faceforward(const Normal3<T> &n, const Vector3<T> &v) {
    return (Dot(n, v) < 0.f) ? -n : n;
}
```

## 2.5 RAYS

A *ray* is a semi-infinite line specified by its origin and direction. `pbrt` represents a `Ray` with a `Point3f` for the origin and a `Vector3f` for the direction. We only need rays with floating-point origins and directions, so `Ray` isn't a template class parameterized by an arbitrary type, as points, vectors, and normals were. A ray is denoted by `r`; it has origin `o` and direction `d`, as shown in Figure 2.7.

<code>AbsDot()</code>	64
<code>Dot()</code>	63
<code>Normal3</code>	71
<code>Point3f</code>	68
<code>Ray</code>	73
<code>Vector3</code>	59
<code>Vector3f</code>	60



**Figure 2.7:** A ray is a semi-infinite line defined by its origin  $\mathbf{o}$  and its direction vector  $\mathbf{d}$ .

```
<Ray Declarations> ≡
class Ray {
public:
    <Ray Public Methods 74>
    <Ray Public Data 73>
};
```

Because we will be referring to these variables often throughout the code, the origin and direction members of a Ray are succinctly named  $\mathbf{o}$  and  $\mathbf{d}$ . Note that we again make the data publicly available for convenience.

```
<Ray Public Data> ≡
Point3f o;
Vector3f d;
```

73

The *parametric form* of a ray expresses it as a function of a scalar value  $t$ , giving the set of points that the ray passes through:

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d} \quad 0 < t < \infty. \quad [2.3]$$

The Ray also includes a member variable that limits the ray to a segment along its infinite extent. This field,  $t_{\text{Max}}$ , allows us to restrict the ray to a segment of points  $[\mathbf{o}, \mathbf{r}(t_{\text{max}})]$ .

Notice that this field is declared as `mutable`, meaning that it can be changed even if the Ray that contains it is `const`—thus, when a ray is passed to a method that takes a `const Ray &`, that method is not allowed to modify its origin or direction but can modify its extent. This convention fits one of the most common uses of rays in the system, as parameters to ray–object intersection testing routines, which will record the offsets to the closest intersection in  $t_{\text{Max}}$ .

```
<Ray Public Data> += 
    mutable Float tMax;
```

73

Each ray has a time value associated with it. In scenes with animated objects, the rendering system constructs a representation of the scene at the appropriate time for each ray.

```
Float 1062
Medium 684
Point3f 68
Ray 73
Vector3f 60
<Ray Public Data> += 
    Float time;
```

73

Finally, each ray records the medium containing its origin. The `Medium` class, introduced in Section 11.3, encapsulates the (potentially spatially varying) properties of media such

as a foggy atmosphere, smoke, or scattering liquids like milk or shampoo. Associating this information with rays makes it possible for other parts of the system to account correctly for the effect of rays passing from one medium to another.

*(Ray Public Data)* +≡  
 const Medium \*medium;

73

Constructing Rays is straightforward. The default constructor relies on the Point3f and Vector3f constructors to set the origin and direction to (0, 0, 0). Alternately, a particular point and direction can be provided. If an origin and direction are provided, the constructor allows a value to be given for tMax, the ray's time and medium.

*(Ray Public Methods)* +≡  
 Ray() : tMax(Infinity), time(0.f), medium(nullptr) {}  
 Ray(const Point3f &o, const Vector3f &d, Float tMax = Infinity,  
     Float time = 0.f, const Medium \*medium = nullptr)  
     : o(o), d(d), tMax(tMax), time(time), medium(medium) {}

73

Because position along a ray can be thought of as a function of a single parameter  $t$ , the Ray class overloads the function application operator for rays. This way, when we need to find the point at a particular position along a ray, we can write code like:

```
Ray r(Point3f(0, 0, 0), Vector3f(1, 2, 3));  

Point3f p = r(1.7);
```

*(Ray Public Methods)* +≡  
 Point3f operator()(Float t) const { return o + d \* t; }

73

### 2.5.1 RAY DIFFERENTIALS

In order to be able to perform better antialiasing with the texture functions defined in Chapter 10, pbrt can keep track of some additional information with rays. In Section 10.1, this information will be used to compute values that are used by the Texture class to estimate the projected area on the image plane of a small part of the scene. From this, the Texture can compute the texture's average value over that area, leading to a higher-quality final image.

RayDifferential is a subclass of Ray that contains additional information about two auxiliary rays. These extra rays represent camera rays offset by one sample in the  $x$  and  $y$  direction from the main ray on the film plane. By determining the area that these three rays project to on an object being shaded, the Texture can estimate an area to average over for proper antialiasing.

Because the RayDifferential class inherits from Ray, geometric interfaces in the system can be written to take const Ray & parameters, so that either a Ray or RayDifferential can be passed to them. Only the routines that need to account for antialiasing and texturing require RayDifferential parameters.

Float 1062  
 Infinity 210  
 Medium 684  
 Point3f 68  
 Ray 73  
 Ray::d 73  
 Ray::o 73  
 RayDifferential 75  
 Texture 614  
 Vector3f 60

```
(Ray Declarations) +≡
    class RayDifferential : public Ray {
public:
    (RayDifferential Public Methods 75)
    (RayDifferential Public Data 75)
};
```

The RayDifferential constructors mirror the Ray's constructors.

```
(RayDifferential Public Methods) ≡
RayDifferential() { hasDifferentials = false; }
RayDifferential(const Point3f &o, const Vector3f &d,
    Float tMax = Infinity, Float time = 0.f,
    const Medium *medium = nullptr)
: Ray(o, d, tMax, time, medium) {
    hasDifferentials = false;
}
```

75

```
(RayDifferential Public Data) ≡
bool hasDifferentials;
Point3f rxOrigin, ryOrigin;
Vector3f rxDirection, ryDirection;
```

75

There is a constructor to create RayDifferentials from Rays. The constructor sets hasDifferentials to false initially because the neighboring rays, if any, are not known.

```
(RayDifferential Public Methods) +≡
RayDifferential(const Ray &ray) : Ray(ray) {
    hasDifferentials = false;
}
```

75

Camera 356  
 Float 1062  
 Infinity 210  
 Medium 684  
 Point3f 68  
 Ray 73  
 Ray::d 73  
 Ray::o 73  
 RayDifferential 75  
 RayDifferential::  
 hasDifferentials  
 75  
 RayDifferential::rxDirection  
 75  
 RayDifferential::rxOrigin 75  
 RayDifferential::ryDirection  
 75  
 RayDifferential::ryOrigin 75  
 SamplerIntegrator 25  
 Vector3f 60

## 2.6 BOUNDING BOXES

Many parts of the system operate on axis-aligned regions of space. For example, multi-threading in pbrt is implemented by subdividing the image into rectangular tiles that

can be processed independently, and the bounding volume hierarchy in Section 4.3 uses 3D boxes to bound geometric primitives in the scene. The `Bounds2` and `Bounds3` template classes are used to represent the extent of these sorts of regions. Both are parameterized by a type `T` that is used to represent the coordinates of its extents.

```
(Bounds Declarations) ≡
template <typename T> class Bounds2 {
public:
    (Bounds2 Public Methods)
    (Bounds2 Public Data)
};

(Bounds Declarations) +≡
template <typename T> class Bounds3 {
public:
    (Bounds3 Public Methods 76)
    (Bounds3 Public Data 77)
};

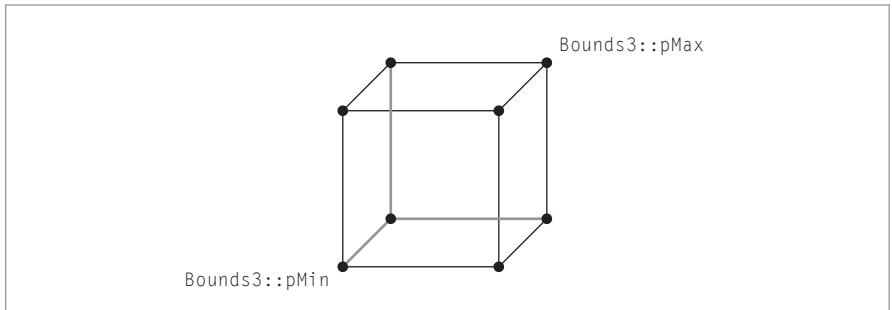
(Bounds Declarations) +≡
typedef Bounds2<Float> Bounds2f;
typedef Bounds2<int> Bounds2i;
typedef Bounds3<Float> Bounds3f;
typedef Bounds3<int> Bounds3i;
```

There are a few possible representations for these sorts of bounding boxes; pbrt uses *axis-aligned bounding boxes* (AABBs), where the box edges are mutually perpendicular and aligned with the coordinate system axes. Another possible choice is *oriented bounding boxes* (OBGs), where the box edges on different sides are still perpendicular to each other but not necessarily coordinate-system aligned. A 3D AABB can be described by one of its vertices and three lengths, each representing the distance spanned along the *x*, *y*, and *z* coordinate axes. Alternatively, two opposite vertices of the box can describe it. We chose the two-point representation for pbrt’s `Bounds2` and `Bounds3` classes; they store the positions of the vertex with minimum coordinate values and of the one with maximum coordinate values. A 2D illustration of a bounding box and its representation is shown in Figure 2.8.

The default constructors create an empty box by setting the extent to an invalid configuration, which violates the invariant that `pMin.x <= pMax.x` (and similarly for the other dimensions). By initializing two corner points with the largest and smallest representable number, any operations involving an empty box (e.g., `Union()`) will yield the correct result.

```
(Bounds3 Public Methods) ≡
Bounds3() {
    T minNum = std::numeric_limits<T>::lowest();
    T maxNum = std::numeric_limits<T>::max();
    pMin = Point3<T>(maxNum, maxNum, maxNum);
    pMax = Point3<T>(minNum, minNum, minNum);
}
```

76	Bounds2 <a href="#">76</a>
76	Bounds3 <a href="#">76</a>
77	Bounds3::pMax <a href="#">77</a>
77	Bounds3::pMin <a href="#">77</a>
1062	Float <a href="#">1062</a>
68	Point3 <a href="#">68</a>



**Figure 2.8: An Axis-Aligned Bounding Box.** The Bounds2 and Bounds3 classes store only the coordinates of the minimum and maximum points of the box; the other box corners are implicit in this representation.

```
(Bounds3 Public Data) ≡
    Point3<T> pMin, pMax;
```

76

It is also useful to be able to initialize a Bounds3 to enclose a single point:

```
(Bounds3 Public Methods) +≡
    Bounds3(const Point3<T> &p) : pMin(p), pMax(p) {}
```

76

If the caller passes two corner points ( $p_1$  and  $p_2$ ) to define the box, the constructor needs to find their component-wise minimum and maximum values since  $p_1$  and  $p_2$  are not necessarily chosen so that  $p_1.x \leq p_2.x$ , and so on.

```
(Bounds3 Public Methods) +≡
    Bounds3(const Point3<T> &p1, const Point3<T> &p2)
        : pMin(std::min(p1.x, p2.x), std::min(p1.y, p2.y),
               std::min(p1.z, p2.z)),
          pMax(std::max(p1.x, p2.x), std::max(p1.y, p2.y),
                std::max(p1.z, p2.z)) {}
```

76

In some cases, it's also useful to use array indexing to select between the two points at the corners of the box. The implementations of these methods select between  $p_{\text{Min}}$  and  $p_{\text{Max}}$  based on the value of  $i$ .

Bounds2 76  
 Bounds3 76  
 Bounds3::pMax 77  
 Bounds3::pMin 77  
 Point3 68

```
(Bounds3 Public Methods) +≡
    const Point3<T> &operator[](int i) const;
    Point3<T> &operator[](int i);
```

76

The Corner() method returns the coordinates of one of the eight corners of the bounding box.

```
(Bounds3 Public Methods) +≡
    Point3<T> Corner(int corner) const {
        return Point3<T>((*this)[(corner & 1)].x,
                           (*this)[(corner & 2) ? 1 : 0].y,
                           (*this)[(corner & 4) ? 1 : 0].z);
    }
```

76

Given a bounding box and a point, the `Union()` function returns a new bounding box that encompasses that point as well as the original box.

```
(Geometry Inline Functions) +≡
    template <typename T> Bounds3 <T>
    Union(const Bounds3<T> &b, const Point3<T> &p) {
        return Bounds3<T>(Point3<T>(std::min(b.pMin.x, p.x),
                                         std::min(b.pMin.y, p.y),
                                         std::min(b.pMin.z, p.z)),
                             Point3<T>(std::max(b.pMax.x, p.x),
                                         std::max(b.pMax.y, p.y),
                                         std::max(b.pMax.z, p.z)));
    }
```

It is similarly possible to construct a new box that bounds the space encompassed by two other bounding boxes. The definition of this function is similar to the earlier `Union()` method that takes a `Point3f`; the difference is that the `pMin` and `pMax` of the second box are used for the `std::min()` and `std::max()` tests, respectively.

```
(Geometry Inline Functions) +≡
    template <typename T> Bounds3<T>
    Union(const Bounds3<T> &b1, const Bounds3<T> &b2) {
        return Bounds3<T>(Point3<T>(std::min(b1.pMin.x, b2.pMin.x),
                                         std::min(b1.pMin.y, b2.pMin.y),
                                         std::min(b1.pMin.z, b2.pMin.z)),
                             Point3<T>(std::max(b1.pMax.x, b2.pMax.x),
                                         std::max(b1.pMax.y, b2.pMax.y),
                                         std::max(b1.pMax.z, b2.pMax.z)));
    }
```

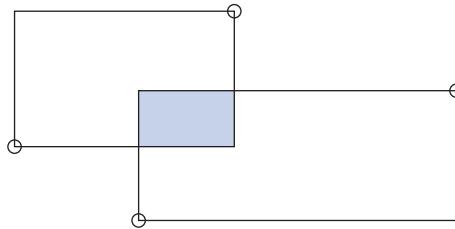
The intersection of two bounding boxes can be found by computing the maximum of their two respective minimum coordinates and the minimum of their maximum coordinates. (See Figure 2.9.)

```
(Geometry Inline Functions) +≡
    template <typename T> Bounds3<T>
    Intersect(const Bounds3<T> &b1, const Bounds3<T> &b2) {
        return Bounds3<T>(Point3<T>(std::max(b1.pMin.x, b2.pMin.x),
                                         std::max(b1.pMin.y, b2.pMin.y),
                                         std::max(b1.pMin.z, b2.pMin.z)),
                             Point3<T>(std::min(b1.pMax.x, b2.pMax.x),
                                         std::min(b1.pMax.y, b2.pMax.y),
                                         std::min(b1.pMax.z, b2.pMax.z)));
    }
```

Bounds3 76

Point3 68

Point3f 68



**Figure 2.9: Intersection of Two Bounding Boxes.** Given two bounding boxes with `pMin` and `pMax` points denoted by open circles, the bounding box of their area of intersection (shaded region) has a minimum point with coordinates given by the maximum of the coordinates of the minimum points of the two boxes in each dimension. Similarly, its maximum point is given by the minimums of the boxes' maximum coordinates.

We are able to determine if two bounding boxes overlap by seeing if their extents overlap in all of  $x$ ,  $y$ , and  $z$ :

```
(Geometry Inline Functions) +≡
template <typename T>
bool Overlaps(const Bounds3<T> &b1, const Bounds3<T> &b2) {
    bool x = (b1.pMax.x >= b2.pMin.x) && (b1.pMin.x <= b2.pMax.x);
    bool y = (b1.pMax.y >= b2.pMin.y) && (b1.pMin.y <= b2.pMax.y);
    bool z = (b1.pMax.z >= b2.pMin.z) && (b1.pMin.z <= b2.pMax.z);
    return (x && y && z);
}
```

Three 1D containment tests determine if a given point is inside the bounding box:

```
(Geometry Inline Functions) +≡
template <typename T>
bool Inside(const Point3<T> &p, const Bounds3<T> &b) {
    return (p.x >= b.pMin.x && p.x <= b.pMax.x &&
            p.y >= b.pMin.y && p.y <= b.pMax.y &&
            p.z >= b.pMin.z && p.z <= b.pMax.z);
}
```

The `InsideExclusive()` variant of `Inside()` doesn't consider points on the upper boundary to be inside the bounds. It is mostly useful with integer-typed bounds.

```
(Geometry Inline Functions) +≡
template <typename T>
bool InsideExclusive(const Point3<T> &p, const Bounds3<T> &b) {
    return (p.x >= b.pMin.x && p.x < b.pMax.x &&
            p.y >= b.pMin.y && p.y < b.pMax.y &&
            p.z >= b.pMin.z && p.z < b.pMax.z);
}
```

The `Expand()` function pads the bounding box by a constant factor in both dimensions.

*(Geometry Inline Functions) +≡*

```
template <typename T, typename U> inline Bounds3<T>
Expand(const Bounds3<T> &b, U delta) {
    return Bounds3<T>(b.pMin - Vector3<T>(delta, delta, delta),
                        b.pMax + Vector3<T>(delta, delta, delta));
}
```

Diagonal() returns the vector along the box diagonal from the minimum point to the maximum point.

*(Bounds3 Public Methods) +≡*

76

```
Vector3<T> Diagonal() const { return pMax - pMin; }
```

Methods for computing the surface area of the six faces of the box and the volume inside of it are also frequently useful.

*(Bounds3 Public Methods) +≡*

76

```
T SurfaceArea() const {
    Vector3<T> d = Diagonal();
    return 2 * (d.x * d.y + d.x * d.z + d.y * d.z);
}
```

*(Bounds3 Public Methods) +≡*

76

```
T Volume() const {
    Vector3<T> d = Diagonal();
    return d.x * d.y * d.z;
}
```

The Bounds3::MaximumExtent() method returns the index of which of the three axes is longest. This is useful, for example, when deciding which axis to subdivide when building some of the ray-intersection acceleration structures.

*(Bounds3 Public Methods) +≡*

76

```
int MaximumExtent() const {
    Vector3<T> d = Diagonal();
    if (d.x > d.y && d.x > d.z)
        return 0;
    else if (d.y > d.z)
        return 1;
    else
        return 2;
}
```

Bounds3 76

Bounds3::Diagonal() 80

Bounds3::MaximumExtent() 80

Bounds3::pMax 77

Bounds3::pMin 77

Lerp() 1079

Point3 68

Point3f 68

Vector3 59

The Lerp() method linearly interpolates between the corners of the box by the given amount in each dimension.

*(Bounds3 Public Methods) +≡*

76

```
Point3<T> Lerp(const Point3f &t) const {
    return Point3<T>(::Lerp(t.x, pMin.x, pMax.x),
                      ::Lerp(t.y, pMin.y, pMax.y),
                      ::Lerp(t.z, pMin.z, pMax.z));
```

`Offset()` returns the continuous position of a point relative to the corners of the box, where a point at the minimum corner has offset  $(0, 0, 0)$ , a point at the maximum corner has offset  $(1, 1, 1)$ , and so forth.

```
(Bounds3 Public Methods) +≡ 76
    Vector3<T> Offset(const Point3<T> &p) const {
        Vector3<T> o = p - pMin;
        if (pMax.x > pMin.x) o.x /= pMax.x - pMin.x;
        if (pMax.y > pMin.y) o.y /= pMax.y - pMin.y;
        if (pMax.z > pMin.z) o.z /= pMax.z - pMin.z;
        return o;
    }
```

`Bounds3` also provides a method that returns the center and radius of a sphere that bounds the bounding box. In general, this may give a far looser fit than a sphere that bounded the original contents of the `Bounds3` directly, although it is a useful method to have available.

```
(Bounds3 Public Methods) +≡ 76
    void BoundingSphere(Point3<T> *center, Float *radius) const {
        *center = (pMin + pMax) / 2;
        *radius = Inside(*center, *this) ? Distance(*center, pMax) : 0;
    }
```

Finally, for integer bounds, there is an iterator class that fulfills the requirements of a C++ forward iterator (i.e., it can only be advanced). The details are slightly tedious and not particularly interesting, so the code isn't included in the book. Having this definition makes it possible to write code using range-based `for` loops to iterate over integer coordinates in a bounding box:

```
Bounds2i b = ...;
for (Point2i p : b) {
    :
}
```

As implemented, the iteration goes up to but doesn't visit points equal to the maximum extent in each dimension.

## 2.7 TRANSFORMATIONS

`Bounds3` 76  
`Bounds3::Inside()` 79  
`Bounds3::pMax` 77  
`Bounds3::pMin` 77  
`Distance()` 70  
`Float` 1062  
`Point3` 68  
`Vector3` 59

In general, a *transformation*  $T$  is a mapping from points to points and from vectors to vectors:

$$p' = T(p) \quad v' = T(v).$$

The transformation  $T$  may be an arbitrary procedure. However, we will consider a subset of all possible transformations in this chapter. In particular, they will be

- *Linear*: If  $T$  is an arbitrary linear transformation and  $s$  is an arbitrary scalar, then  $T(sv) = sT(v)$  and  $T(v_1 + v_2) = T(v_1) + T(v_2)$ . These two properties can greatly simplify reasoning about transformations.

- *Continuous*: Roughly speaking,  $T$  maps the neighborhoods around  $p$  and  $v$  to neighborhoods around  $p'$  and  $v'$ .
- *One-to-one and invertible*: For each  $p$ ,  $T$  maps  $p$  to a single unique  $p'$ . Furthermore, there exists an inverse transform  $T^{-1}$  that maps  $p'$  back to  $p$ .

We will often want to take a point, vector, or normal defined with respect to one coordinate frame and find its coordinate values with respect to another frame. Using basic properties of linear algebra, a  $4 \times 4$  matrix can be shown to express the linear transformation of a point or vector from one frame to another. Furthermore, such a  $4 \times 4$  matrix suffices to express all linear transformations of points and vectors within a fixed frame, such as translation in space or rotation around a point. Therefore, there are two different (and incompatible!) ways that a matrix can be interpreted:

- *Transformation of the frame*: Given a point, the matrix could express how to compute a *new* point in the same frame that represents the transformation of the original point (e.g., by translating it in some direction).
- *Transformation from one frame to another*: A matrix can express the coordinates of a point or vector in a new frame in terms of the coordinates in the original frame.

Most uses of transformations in pb<sub>rt</sub> are for transforming points from one frame to another.

In general, transformations make it possible to work in the most convenient coordinate space. For example, we can write routines that define a virtual camera, assuming that the camera is located at the origin, looks down the  $z$  axis, and has the  $y$  axis pointing up and the  $x$  axis pointing right. These assumptions greatly simplify the camera implementation. Then, to place the camera at any point in the scene looking in any direction, we just construct a transformation that maps points in the scene's coordinate system to the camera's coordinate system. (See Section 6.1.1 for more information about camera coordinate spaces in pb<sub>rt</sub>.)

### 2.7.1 HOMOGENEOUS COORDINATES

Given a frame defined by  $(p, v_1, v_2, v_3)$ , there is ambiguity between the representation of a point  $(p_x, p_y, p_z)$  and a vector  $(v_x, v_y, v_z)$  with the same  $(x, y, z)$  coordinates. Using the representations of points and vectors introduced at the start of the chapter, we can write the point as the inner product  $[s_1 s_2 s_3 1][v_1 v_2 v_3 p_o]^T$  and the vector as the inner product  $[s'_1 s'_2 s'_3 0][v_1 v_2 v_3 p_o]^T$ . These four vectors of three  $s_i$  values and a zero or one are called the *homogeneous* representations of the point and the vector. The fourth coordinate of the homogeneous representation is sometimes called the *weight*. For a point, its value can be any scalar other than zero: the homogeneous points  $[1, 3, -2, 1]$  and  $[-2, -6, 4, -2]$  describe the same Cartesian point  $(1, 3, -2)$ . Converting homogeneous points into ordinary points entails dividing the first three components by the weight:

$$(x, y, z, w) = \left( \frac{x}{w}, \frac{y}{w}, \frac{z}{w} \right).$$

We will use these facts to see how a transformation matrix can describe how points and vectors in one frame can be mapped to another frame. Consider a matrix  $M$  that

describes the transformation from one coordinate system to another:

$$\mathbf{M} = \begin{pmatrix} m_{0,0} & m_{0,1} & m_{0,2} & m_{0,3} \\ m_{1,0} & m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,0} & m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,0} & m_{3,1} & m_{3,2} & m_{3,3} \end{pmatrix}.$$

(In this book, we define matrix element indices starting from zero, so that equations and source code correspond more directly.) Then if the transformation represented by  $\mathbf{M}$  is applied to the  $x$  axis vector  $(1, 0, 0)$ , we have

$$\mathbf{Mx} = \mathbf{M}[1 \ 0 \ 0 \ 0]^T = [m_{0,0} \ m_{1,0} \ m_{2,0} \ m_{3,0}]^T.$$

Thus, directly reading the columns of the matrix shows how the basis vectors and the origin of the current coordinate system are transformed by the matrix:

$$\begin{aligned} \mathbf{My} &= [m_{0,1} \ m_{1,1} \ m_{2,1} \ m_{3,1}]^T \\ \mathbf{Mz} &= [m_{0,2} \ m_{1,2} \ m_{2,2} \ m_{3,2}]^T \\ \mathbf{Mp} &= [m_{0,3} \ m_{1,3} \ m_{2,3} \ m_{3,3}]^T. \end{aligned}$$

In general, by characterizing how the basis is transformed, we know how any point or vector specified in terms of that basis is transformed. Because points and vectors in the current coordinate system are expressed in terms of the current coordinate system's frame, applying the transformation to them directly is equivalent to applying the transformation to the current coordinate system's basis and finding their coordinates in terms of the transformed basis.

We will not use homogeneous coordinates explicitly in our code; there is no `Homogeneous` class in `pbrt`. However, the various transformation routines in the next section will implicitly convert points, vectors, and normals to homogeneous form, transform the homogeneous points, and then convert them back before returning the result. This isolates the details of homogeneous coordinates in one place (namely, the implementation of transformations).

```
<Transform Declarations> ≡
    class Transform {
        public:
            <Transform Public Methods 84>
        private:
            <Transform Private Data 84>
    };
```

A transformation is represented by the elements of the matrix  $\mathbf{m}$ , a `Matrix4x4` object. The low-level `Matrix4x4` class is defined in Section A.5.3. The matrix  $\mathbf{m}$  is stored in *row-major* form, so element  $\mathbf{m}[i][j]$  corresponds to  $m_{i,j}$ , where  $i$  is the row number and  $j$  is the column number. For convenience, the `Transform` also stores the inverse of the matrix  $\mathbf{m}$  in the `Transform::mInv` member; for `pbrt`'s needs, it is better to have the inverse easily available than to repeatedly compute it as needed.

`Matrix4x4` 1081  
`Transform` 83  
`Transform::mInv` 84

This representation of transformations is relatively memory hungry: assuming 4 bytes of storage for a `Float` value, a `Transform` requires 128 bytes of storage. Used naively,

this approach can be wasteful; if a scene has millions of shapes but only a few thousand unique transformations, there's no reason to redundantly store the same transform many times in memory. Therefore, Shapes in pbrt store a pointer to a `Transform`, and the scene specification code defined in Section B.3.5 uses a `TransformCache` to ensure that all shapes that share the same transformation point to a single instance of that transformation in memory.

This decision to share transformations implies a loss of flexibility, however: the elements of a `Transform` shouldn't be modified after it is created if the `Transform` is shared by multiple objects in the scene (and those objects don't expect it to be changing.) This limitation isn't a problem in practice, since the transformations in a scene are typically created when pbrt parses the scene description file and don't need to change later at rendering time.

```
<Transform Private Data> ≡ 83
    Matrix4x4 m, mInv;
```

## 2.7.2 BASIC OPERATIONS

When a new `Transform` is created, it defaults to the *identity transformation*—the transformation that maps each point and each vector to itself. This transformation is represented by the *identity matrix*:

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The implementation here relies on the default `Matrix4x4` constructor to fill in the identity matrix for `m` and `mInv`.

```
<Transform Public Methods> ≡ 83
    Transform() { }
```

A `Transform` can also be created from a given matrix. In this case, the given matrix must be explicitly inverted.

```
<Transform Public Methods> +≡ 83
    Transform(const Float mat[4][4]) {
        m = Matrix4x4(mat[0][0], mat[0][1], mat[0][2], mat[0][3],
                      mat[1][0], mat[1][1], mat[1][2], mat[1][3],
                      mat[2][0], mat[2][1], mat[2][2], mat[2][3],
                      mat[3][0], mat[3][1], mat[3][2], mat[3][3]);
        mInv = Inverse(m);
    }
```

```
<Transform Public Methods> +≡ 83
    Transform(const Matrix4x4 &m) : m(m), mInv(Inverse(m)) { }
```

The most commonly used constructor takes a reference to the transformation matrix along with an explicitly provided inverse. This is a superior approach to computing the

Float 1062  
Inverse() 1081  
Matrix4x4 1081  
Shape 123  
Transform 83  
Transform::m 84  
Transform::mInv 84  
TransformCache 1124

inverse in the constructor because many geometric transformations have very simple inverses and we can avoid the expense and potential loss of numeric precision from computing a general  $4 \times 4$  matrix inverse. Of course, this places the burden on the caller to make sure that the supplied inverse is correct.

```
<Transform Public Methods> +≡ 83
    Transform(const Matrix4x4 &m, const Matrix4x4 &mInv)
        : m(m), mInv(mInv) {
    }
```

The `Transform` representing the inverse of a `Transform` can be returned by just swapping the roles of `mInv` and `m`.

```
<Transform Public Methods> +≡ 83
    friend Transform Inverse(const Transform &t) {
        return Transform(t.mInv, t.m);
    }
```

Transposing the two matrices in the transform to compute a new transform can also be useful.

```
<Transform Public Methods> +≡ 83
    friend Transform Transpose(const Transform &t) {
        return Transform(Transpose(t.m), Transpose(t.mInv));
    }
```

We provide `Transform` equality (and inequality) testing methods; their implementation is straightforward and not included here. `Transform` also provides an `IsIdentity()` method that checks to see if the transformation is the identity.

### 2.7.3 TRANSLATIONS

One of the simplest transformations is the *translation transformation*,  $T(\Delta x, \Delta y, \Delta z)$ . When applied to a point  $p$ , it translates  $p$ 's coordinates by  $\Delta x$ ,  $\Delta y$ , and  $\Delta z$ , as shown in Figure 2.10. As an example,  $T(2, 2, 1)(x, y, z) = (x + 2, y + 2, z + 1)$ .

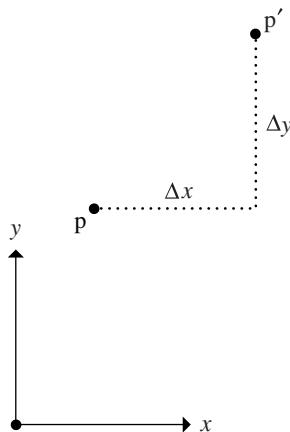
Translation has some simple properties:

$$\begin{aligned} T(0, 0, 0) &= I \\ T(x_1, y_1, z_1)T(x_2, y_2, z_2) &= T(x_1 + x_2, y_1 + y_2, z_1 + z_2) \\ T(x_1, y_1, z_1)T(x_2, y_2, z_2) &= T(x_2, y_2, z_2)T(x_1, y_1, z_1) \\ T^{-1}(x, y, z) &= T(-x, -y, -z). \end{aligned}$$

Matrix4x4 1081  
`Matrix4x4::Transpose()` 1081  
`Transform` 83  
`Transform::m` 84  
`Transform::mInv` 84

Translation only affects points, leaving vectors unchanged. In matrix form, the translation transformation is

$$T(\Delta x, \Delta y, \Delta z) = \begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$



**Figure 2.10: Translation in 2D.** Adding offsets  $\Delta x$  and  $\Delta y$  to a point's coordinates correspondingly changes its position in space.

When we consider the operation of a translation matrix on a point, we see the value of homogeneous coordinates. Consider the product of the matrix for  $T(\Delta x, \Delta y, \Delta z)$  with a point  $p$  in homogeneous coordinates  $[x \ y \ z \ 1]$ :

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + \Delta x \\ y + \Delta y \\ z + \Delta z \\ 1 \end{pmatrix}.$$

As expected, we have computed a new point with its coordinates offset by  $(\Delta x, \Delta y, \Delta z)$ . However, if we apply  $T$  to a vector  $v$ , we have

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix}.$$

The result is the same vector  $v$ . This makes sense because vectors represent directions, so translation leaves them unchanged.

We will define a routine that creates a new Transform matrix to represent a given translation—it is a straightforward application of the translation matrix equation. This routine fully initializes the Transform that is returned, also initializing the matrix that represents the inverse of the translation.

```
(Transform Method Definitions) ≡
    Transform Translate(const Vector3f &delta) {
        Matrix4x4 m(1, 0, 0, delta.x,
                    0, 1, 0, delta.y,
                    0, 0, 1, delta.z,
                    0, 0, 0,         1);
        Matrix4x4 minv(1, 0, 0, -delta.x,
                        0, 1, 0, -delta.y,
                        0, 0, 1, -delta.z,
                        0, 0, 0,         1);
        return Transform(m, minv);
    }
```

## 2.7.4 SCALING

Another basic transformation is the *scale transformation*,  $\mathbf{S}(s_x, s_y, s_z)$ . It has the effect of taking a point or vector and multiplying its components by scale factors in  $x$ ,  $y$ , and  $z$ :  $\mathbf{S}(2, 2, 1)(x, y, z) = (2x, 2y, z)$ . It has the following basic properties:

$$\begin{aligned}\mathbf{S}(1, 1, 1) &= \mathbf{I} \\ \mathbf{S}(x_1, y_1, z_1)\mathbf{S}(x_2, y_2, z_2) &= \mathbf{S}(x_1x_2, y_1y_2, z_1z_2) \\ \mathbf{S}^{-1}(x, y, z) &= \mathbf{S}\left(\frac{1}{x}, \frac{1}{y}, \frac{1}{z}\right).\end{aligned}$$

We can differentiate between *uniform scaling*, where all three scale factors have the same value, and *nonuniform scaling*, where they may have different values. The general scale matrix is

$$\mathbf{S}(x, y, z) = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

```
(Transform Method Definitions) +≡
    Transform Scale(Float x, Float y, Float z) {
        Matrix4x4 m(x, 0, 0, 0,
                    0, y, 0, 0,
                    0, 0, z, 0,
                    0, 0, 0, 1);
        Matrix4x4 minv(1/x, 0, 0, 0,
                        0, 1/y, 0, 0,
                        0, 0, 1/z, 0,
                        0, 0, 0, 1);
        return Transform(m, minv);
    }
```

It's useful to be able to test if a transformation has a scaling term in it; an easy way to do this is to transform the three coordinate axes and see if any of their lengths are appreciably different from one.

*(Transform Public Methods) +≡*

83

```
bool HasScale() const {
    Float la2 = (*this)(Vector3f(1, 0, 0)).LengthSquared();
    Float lb2 = (*this)(Vector3f(0, 1, 0)).LengthSquared();
    Float lc2 = (*this)(Vector3f(0, 0, 1)).LengthSquared();
#define NOT_ONE(x) ((x) < .999f || (x) > 1.001f)
    return (NOT_ONE(la2) || NOT_ONE(lb2) || NOT_ONE(lc2));
#undef NOT_ONE
}
```

## 2.7.5 $x$ , $y$ , AND $z$ AXIS ROTATIONS

Another useful type of transformation is the *rotation transformation*,  $R$ . In general, we can define an arbitrary axis from the origin in any direction and then rotate around that axis by a given angle. The most common rotations of this type are around the  $x$ ,  $y$ , and  $z$  coordinate axes. We will write these rotations as  $R_x(\theta)$ ,  $R_y(\theta)$ , and so on. The rotation around an arbitrary axis  $(x, y, z)$  is denoted by  $R_{(x,y,z)}(\theta)$ .

Rotations also have some basic properties:

$$\begin{aligned} R_a(0) &= I \\ R_a(\theta_1)R_a(\theta_2) &= R_a(\theta_1 + \theta_2) \\ R_a(\theta_1)R_a(\theta_2) &= R_a(\theta_2)R_a(\theta_1) \\ R_a^{-1}(\theta) &= R_a(-\theta) = R_a^T(\theta), \end{aligned}$$

where  $R^T$  is the matrix transpose of  $R$ . This last property, that the inverse of  $R$  is equal to its transpose, stems from the fact that  $R$  is an *orthogonal matrix*; its upper  $3 \times 3$  components are all orthogonal to each other. Fortunately, the transpose is much easier to compute than a full matrix inverse.

For a left-handed coordinate system, the matrix for rotation around the  $x$  axis is

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

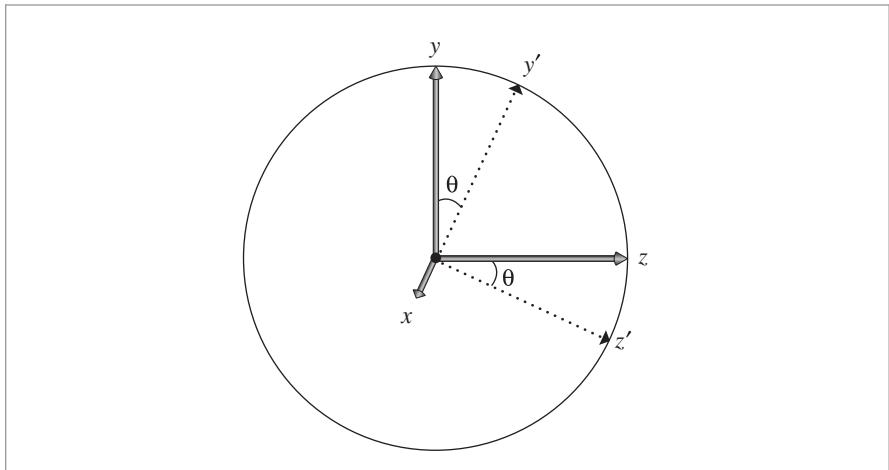
Figure 2.11 gives an intuition for how this matrix works. It's easy to see that it leaves the  $x$  axis unchanged:

$$R_x(\theta)[1 \ 0 \ 0 \ 0]^T = [1 \ 0 \ 0 \ 0]^T.$$

It maps the  $y$  axis  $(0, 1, 0)$  to  $(0, \cos \theta, \sin \theta)$  and the  $z$  axis to  $(0, -\sin \theta, \cos \theta)$ . The  $y$  and  $z$  axes remain in the same plane, perpendicular to the  $x$  axis, but are rotated by the given angle. An arbitrary point in space is similarly rotated about the  $x$  axis by this transformation while staying in the same  $yz$  plane as it was originally.

The implementation of the `RotateX()` function is straightforward.

```
Float 1062
RotateX() 89
Vector3::LengthSquared() 65
Vector3f 60
```



**Figure 2.11:** Rotation by an angle  $\theta$  about the  $x$  axis leaves the  $x$  coordinate unchanged. The  $y$  and  $z$  axes are mapped to the vectors given by the dashed lines;  $y$  and  $z$  coordinates move accordingly.

*{Transform Method Definitions} +≡*

```
Transform RotateX(Float theta) {
    Float sinTheta = std::sin(Radians(theta));
    Float cosTheta = std::cos(Radians(theta));
    Matrix4x4 m(1, 0, 0, 0,
                 0, cosTheta, -sinTheta, 0,
                 0, sinTheta, cosTheta, 0,
                 0, 0, 0, 1);
    return Transform(m, Transpose(m));
}
```

Similarly, for rotation around  $y$  and  $z$ , we have

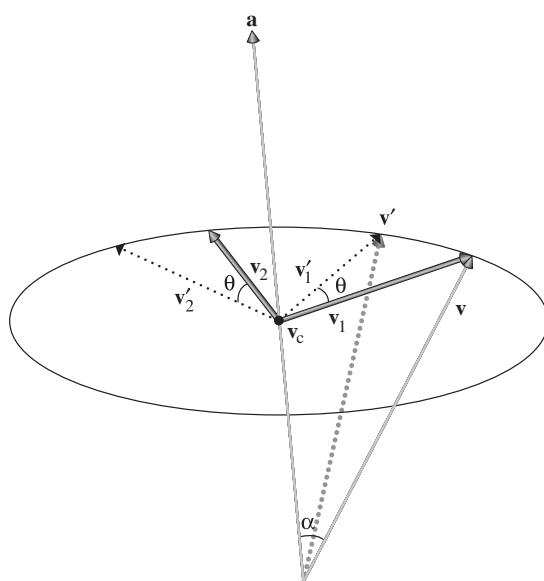
$$\mathbf{R}_y(\theta) = \begin{pmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{R}_z(\theta) = \begin{pmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The implementations of `RotateY()` and `RotateZ()` follow directly and are not included here.

## 2.7.6 ROTATION AROUND AN ARBITRARY AXIS

Float 1062  
Matrix4x4 1081  
Transform 83  
Transform::Transpose() 85

We also provide a routine to compute the transformation that represents rotation around an arbitrary axis. The usual derivation of this matrix is based on computing rotations that map the given axis to a fixed axis (e.g.,  $z$ ), performing the rotation there, and then rotating the fixed axis back to the original axis. A more elegant derivation can be constructed with vector algebra.



**Figure 2.12:** A vector  $\mathbf{v}$  can be rotated around an arbitrary axis  $\mathbf{a}$  by constructing a coordinate system  $(\mathbf{p}, \mathbf{v}_1, \mathbf{v}_2)$  in the plane perpendicular to the axis that passes through  $\mathbf{v}$ 's end point and rotating the vectors  $\mathbf{v}_1$  and  $\mathbf{v}_2$  about  $\mathbf{p}$ . Applying this rotation to the axes of the coordinate system  $(1, 0, 0)$ ,  $(0, 1, 0)$ , and  $(0, 0, 1)$  gives the general rotation matrix for this rotation.

Consider a normalized direction vector  $\mathbf{a}$  that gives the axis to rotate around by angle  $\theta$ , and a vector  $\mathbf{v}$  to be rotated (Figure 2.12). First, we can compute the vector  $\mathbf{v}_c$  along the axis  $\mathbf{a}$  that is in the plane through the end point of  $\mathbf{v}$  and is parallel to  $\mathbf{a}$ . Assuming  $\mathbf{v}$  and  $\mathbf{a}$  form an angle  $\alpha$ , we have

$$\mathbf{v}_c = \mathbf{a} \|\mathbf{v}\| \cos \alpha = \mathbf{a}(\mathbf{v} \cdot \mathbf{a}).$$

We now compute a pair of basis vectors  $\mathbf{v}_1$  and  $\mathbf{v}_2$  in this plane. Trivially, one of them is

$$\mathbf{v}_1 = \mathbf{v} - \mathbf{v}_c,$$

and the other can be computed with a cross product

$$\mathbf{v}_2 = (\mathbf{v}_1 \times \mathbf{a}).$$

Because  $\mathbf{a}$  is normalized,  $\mathbf{v}_1$  and  $\mathbf{v}_2$  have the same length, equal to the length of the vector between  $\mathbf{v}$  and  $\mathbf{v}_c$ . To now compute the rotation by an angle  $\theta$  about  $\mathbf{v}_c$  in the plane of rotation, the rotation formulas earlier give us

$$\mathbf{v}' = \mathbf{v}_c + \mathbf{v}_1 \cos \theta + \mathbf{v}_2 \sin \theta.$$

To convert this to a rotation matrix, we apply this formula to the basis vectors  $(1, 0, 0)$ ,  $(0, 1, 0)$ , and  $(0, 0, 1)$  to get the values of the rows of the matrix. The result of all this is

encapsulated in the following function. As with the other rotation matrices, the inverse is equal to the transpose.

```
<Transform Method Definitions> +≡
    Transform Rotate(Float theta, const Vector3f &axis) {
        Vector3f a = Normalize(axis);
        Float sinTheta = std::sin(Radians(theta));
        Float cosTheta = std::cos(Radians(theta));
        Matrix4x4 m;
        <Compute rotation of first basis vector 91>
        <Compute rotations of second and third basis vectors>
        return Transform(m, Transpose(m));
    }

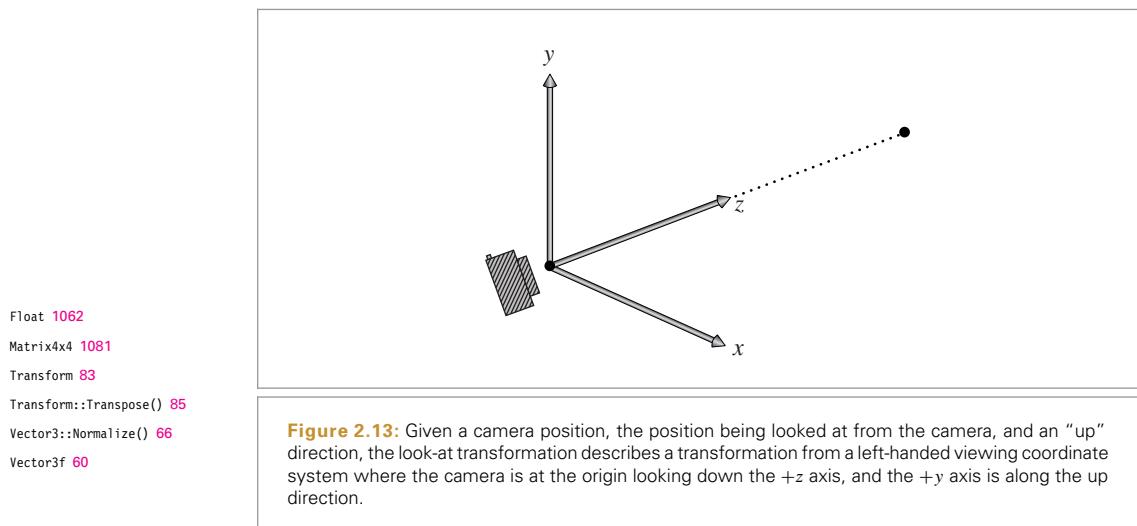
<Compute rotation of first basis vector> ≡
    m.m[0][0] = a.x * a.x + (1 - a.x * a.x) * cosTheta;
    m.m[0][1] = a.x * a.y * (1 - cosTheta) - a.z * sinTheta;
    m.m[0][2] = a.x * a.z * (1 - cosTheta) + a.y * sinTheta;
    m.m[0][3] = 0;
```

91

The code for the other two basis vectors follows similarly and isn't included here.

## 2.7.7 THE LOOK-AT TRANSFORMATION

The *look-at transformation* is particularly useful for placing a camera in the scene. The caller specifies the desired position of the camera, a point the camera is looking at, and an “up” vector that orients the camera along the viewing direction implied by the first two parameters. All of these values are given in world space coordinates. The look-at construction then gives a transformation between camera space and world space (Figure 2.13).



In order to find the entries of the look-at transformation matrix, we use principles described earlier in this section: the columns of a transformation matrix give the effect of the transformation on the basis of a coordinate system.

*(Transform Method Definitions)* +≡

```
Transform LookAt(const Point3f &pos, const Point3f &look,
    const Vector3f &up) {
    Matrix4x4 cameraToWorld;
    (Initialize fourth column of viewing matrix 92)
    (Initialize first three columns of viewing matrix 92)
    return Transform(Inverse(cameraToWorld), cameraToWorld);
}
```

The easiest column is the fourth one, which gives the point that the camera space origin,  $[0\ 0\ 0\ 1]^T$ , maps to in world space. This is clearly just the camera position, supplied by the user.

*(Initialize fourth column of viewing matrix)* ≡

92

```
cameraToWorld.m[0][3] = pos.x;
cameraToWorld.m[1][3] = pos.y;
cameraToWorld.m[2][3] = pos.z;
cameraToWorld.m[3][3] = 1;
```

The other three columns aren't much more difficult. First, `LookAt()` computes the normalized direction vector from the camera location to the look-at point; this gives the vector coordinates that the  $z$  axis should map to and, thus, the third column of the matrix. (In a left-handed coordinate system, camera space is defined with the viewing direction down the  $+z$  axis.) The first column, giving the world space direction that the  $+x$  axis in camera space maps to, is found by taking the cross product of the user-supplied “up” vector with the recently computed viewing direction vector. Finally, the “up” vector is recomputed by taking the cross product of the viewing direction vector with the transformed  $x$  axis vector, thus ensuring that the  $y$  and  $z$  axes are perpendicular and we have an orthonormal viewing coordinate system.

*(Initialize first three columns of viewing matrix)* ≡

92

```
Vector3f dir = Normalize(look - pos);
Vector3f left = Normalize(Cross(Normalize(up), dir));
Vector3f newUp = Cross(dir, left);
cameraToWorld.m[0][0] = left.x;
cameraToWorld.m[1][0] = left.y;
cameraToWorld.m[2][0] = left.z;
cameraToWorld.m[3][0] = 0.;

cameraToWorld.m[0][1] = newUp.x;
cameraToWorld.m[1][1] = newUp.y;
cameraToWorld.m[2][1] = newUp.z;
cameraToWorld.m[3][1] = 0.;

cameraToWorld.m[0][2] = dir.x;
cameraToWorld.m[1][2] = dir.y;
cameraToWorld.m[2][2] = dir.z;
cameraToWorld.m[3][2] = 0.;
```

Cross() 65  
Inverse() 1081  
LookAt() 92  
Matrix4x4 1081  
Point3f 68  
Transform 83  
Vector3::Normalize() 66  
Vector3f 60

## 2.8 APPLYING TRANSFORMATIONS

We can now define routines that perform the appropriate matrix multiplications to transform points and vectors. We will overload the function application operator to describe these transformations; this lets us write code like:

```
Point3f p = ...;
Transform T = ...;
Point3f pNew = T(p);
```

### 2.8.1 POINTS

The point transformation routine takes a point  $(x, y, z)$  and implicitly represents it as the homogeneous column vector  $[x \ y \ z \ 1]^T$ . It then transforms the point by premultiplying this vector with the transformation matrix. Finally, it divides by  $w$  to convert back to a nonhomogeneous point representation. For efficiency, this method skips the division by the homogeneous weight,  $w$ , when  $w = 1$ , which is common for most of the transformations that will be used in `pbrt`—only the projective transformations defined in Chapter 6 will require this division.

```
<Transform Inline Functions> ≡
template <typename T> inline Point3<T>
Transform::operator()(const Point3<T> &p) const {
    T x = p.x, y = p.y, z = p.z;
    T xp = m.m[0][0]*x + m.m[0][1]*y + m.m[0][2]*z + m.m[0][3];
    T yp = m.m[1][0]*x + m.m[1][1]*y + m.m[1][2]*z + m.m[1][3];
    T zp = m.m[2][0]*x + m.m[2][1]*y + m.m[2][2]*z + m.m[2][3];
    T wp = m.m[3][0]*x + m.m[3][1]*y + m.m[3][2]*z + m.m[3][3];
    if (wp == 1) return Point3<T>(xp, yp, zp);
    else         return Point3<T>(xp, yp, zp) / wp;
}
```

### 2.8.2 VECTORS

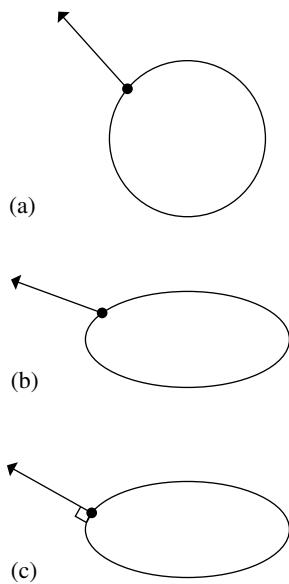
The transformations of vectors can be computed in a similar fashion. However, the multiplication of the matrix and the row vector is simplified since the implicit homogeneous  $w$  coordinate is zero.

```
<Transform Inline Functions> +≡
template <typename T> inline Vector3<T>
Transform::operator()(const Vector3<T> &v) const {
    T x = v.x, y = v.y, z = v.z;
    return Vector3<T>(m.m[0][0]*x + m.m[0][1]*y + m.m[0][2]*z,
                       m.m[1][0]*x + m.m[1][1]*y + m.m[1][2]*z,
                       m.m[2][0]*x + m.m[2][1]*y + m.m[2][2]*z);
```

Point3 68  
Transform 83  
Vector3 59

### 2.8.3 NORMALS

Normals do not transform in the same way that vectors do, as shown in Figure 2.14. Although tangent vectors transform in the straightforward way, normals require special



**Figure 2.14: Transforming Surface Normals.** (a) Original circle. (b) When scaling the circle to be half as tall in the  $y$  direction, simply treating the normal as a direction and scaling it in the same manner gives a normal that is no longer perpendicular to the surface. (c) A properly transformed normal.

treatment. Because the normal vector  $\mathbf{n}$  and any tangent vector  $\mathbf{t}$  on the surface are orthogonal by construction, we know that

$$\mathbf{n} \cdot \mathbf{t} = \mathbf{n}^T \mathbf{t} = 0.$$

When we transform a point on the surface by some matrix  $\mathbf{M}$ , the new tangent vector  $\mathbf{t}'$  at the transformed point is  $\mathbf{Mt}$ . The transformed normal  $\mathbf{n}'$  should be equal to  $\mathbf{S}\mathbf{n}$  for some  $4 \times 4$  matrix  $\mathbf{S}$ . To maintain the orthogonality requirement, we must have

$$\begin{aligned} 0 &= (\mathbf{n}')^T \mathbf{t}' \\ &= (\mathbf{S}\mathbf{n})^T \mathbf{Mt} \\ &= (\mathbf{n})^T \mathbf{S}^T \mathbf{Mt}. \end{aligned}$$

This condition holds if  $\mathbf{S}^T \mathbf{M} = \mathbf{I}$ , the identity matrix. Therefore,  $\mathbf{S}^T = \mathbf{M}^{-1}$ , and so  $\mathbf{S} = (\mathbf{M}^{-1})^T$ , and we see that normals must be transformed by the inverse transpose of the transformation matrix. This detail is one of the main reasons why Transforms maintain their inverses.

Note that this method does not explicitly compute the transpose of the inverse when transforming normals. It just indexes into the inverse matrix in a different order (compare to the code for transforming `Vector3fs`).

Transform 83

Vector3f 60

```
(Transform Inline Functions) +≡
template <typename T> inline Normal3<T>
Transform::operator()(const Normal3<T> &n) const {
    T x = n.x, y = n.y, z = n.z;
    return Normal3<T>(mInv.m[0][0]*x + mInv.m[1][0]*y + mInv.m[2][0]*z,
                        mInv.m[0][1]*x + mInv.m[1][1]*y + mInv.m[2][1]*z,
                        mInv.m[0][2]*x + mInv.m[1][2]*y + mInv.m[2][2]*z);
}
```

## 2.8.4 RAYS

Transforming rays is conceptually straightforward: it's a matter of transforming the constituent origin and direction and copying the other data members. (pbrt also provides a similar method for transforming RayDifferentials.)

The approach used in pbrt to manage floating-point round-off error introduces some subtleties that require a small adjustment to the transformed ray origin. The *(Offset ray origin to edge of error bounds)* fragment handles these details; it is defined in Section 3.9.4, where round-off error and pbrt's mechanisms for dealing with it are discussed.

```
(Transform Inline Functions) +≡
inline Ray Transform::operator()(const Ray &r) const {
    Vector3f oError;
    Point3f o = (*this)(r.o, &oError);
    Vector3f d = (*this)(r.d);
(Offset ray origin to edge of error bounds and compute tMax 233)
    return Ray(o, d, tMax, r.time, r.medium);
}
```

## 2.8.5 BOUNDING BOXES

The easiest way to transform an axis-aligned bounding box is to transform all eight of its corner vertices and then compute a new bounding box that encompasses those points. The implementation of this approach is shown below; one of the exercises for this chapter is to implement a technique to do this computation more efficiently.

```
(Transform Method Definitions) +≡
Bounds3f Transform::operator()(const Bounds3f &b) const {
    const Transform &M = *this;
    Bounds3f ret(M(Point3f(b.pMin.x, b.pMin.y, b.pMin.z)));
    ret = Union(ret, M(Point3f(b.pMax.x, b.pMin.y, b.pMin.z)));
    ret = Union(ret, M(Point3f(b.pMin.x, b.pMax.y, b.pMin.z)));
    ret = Union(ret, M(Point3f(b.pMin.x, b.pMin.y, b.pMax.z)));
    ret = Union(ret, M(Point3f(b.pMin.x, b.pMax.y, b.pMax.z)));
    ret = Union(ret, M(Point3f(b.pMax.x, b.pMax.y, b.pMin.z)));
    ret = Union(ret, M(Point3f(b.pMax.x, b.pMin.y, b.pMax.z)));
    ret = Union(ret, M(Point3f(b.pMax.x, b.pMax.y, b.pMax.z)));
    return ret;
}
```

Bounds3f 76

Normal3 71

Point3f 68

Ray 73

RayDifferential 75

Transform 83

Vector3f 60

## 2.8.6 COMPOSITION OF TRANSFORMATIONS

Having defined how the matrices representing individual types of transformations are constructed, we can now consider an aggregate transformation resulting from a series of individual transformations. Finally, we will see the real value of representing transformations with matrices.

Consider a series of transformations ABC. We'd like to compute a new transformation T such that applying T gives the same result as applying each of A, B, and C in reverse order; that is,  $A(B(C(p))) = T(p)$ . Such a transformation T can be computed by multiplying the matrices of the transformations A, B, and C together. In pbrt, we can write:

```
Transform T = A * B * C;
```

Then we can apply T to `Point3f`s `p` as usual, `Point3f pp = T(p)`, instead of applying each transformation in turn: `Point3f pp = A(B(C(p)))`.

We use the C++ `*` operator to compute the new transformation that results from post-multiplying a transformation with another transformation `t2`. In matrix multiplication, the  $(i, j)$ th element of the resulting matrix is the inner product of the  $i$ th row of the first matrix with the  $j$ th column of the second.

The inverse of the resulting transformation is equal to the product of `t2.mInv * mInv`. This is a result of the matrix identity

$$(AB)^{-1} = B^{-1}A^{-1}.$$

*(Transform Method Definitions) +≡*

```
Transform Transform::operator*(const Transform &t2) const {
    return Transform(Matrix4x4::Mul(m, t2.m),
                    Matrix4x4::Mul(t2.mInv, mInv));
}
```

## 2.8.7 TRANSFORMATIONS AND COORDINATE SYSTEM HANDEDNESS

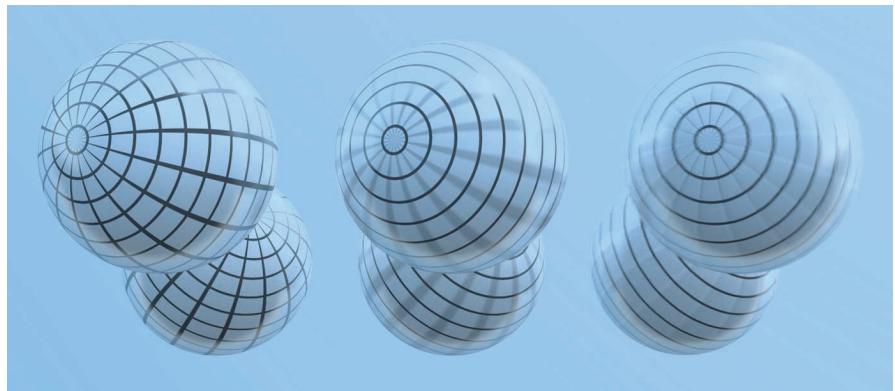
Certain types of transformations change a left-handed coordinate system into a right-handed one, or vice versa. Some routines will need to know if the handedness of the source coordinate system is different from that of the destination. In particular, routines that want to ensure that a surface normal always points “outside” of a surface might need to flip the normal’s direction after transformation if the handedness changes.

Fortunately, it is easy to tell if handedness is changed by a transformation: it happens only when the determinant of the transformation’s upper-left  $3 \times 3$  submatrix is negative.

*(Transform Method Definitions) +≡*

```
bool Transform::SwapsHandedness() const {
    Float det =
        m.m[0][0] * (m.m[1][1] * m.m[2][2] - m.m[1][2] * m.m[2][1]) -
        m.m[0][1] * (m.m[1][0] * m.m[2][2] - m.m[1][2] * m.m[2][0]) +
        m.m[0][2] * (m.m[1][0] * m.m[2][1] - m.m[1][1] * m.m[2][0]);
    return det < 0;
}
```

```
Float 1062
Matrix4x4 1081
Matrix4x4::Mul() 1081
Point3f 68
Transform 83
Transform:: 84
Transform::mInv 84
```



**Figure 2.15: Spinning Spheres.** Three spheres, spinning at different rates using the transformation animation code implemented in this section. Note that the reflections of the spheres are blurry as well as the spheres themselves.

## \* 2.9 ANIMATING TRANSFORMATIONS

`pbrt` supports keyframe matrix animation for cameras and geometric primitives in the scene. Rather than just supplying a single transformation to place the corresponding object in the scene, the user may supply a number of *keyframe* transformations, each one associated with a particular point in time. This makes it possible for the camera to move and for objects in the scene to be moving during the time the simulated camera's shutter is open. Figure 2.15 shows three spheres animated using keyframe matrix animation in `pbrt`.

In general, the problem of interpolating between keyframe matrices is under-defined. As one example, if we have a rotation about the  $x$  axis of 179 degrees followed by another of 181 degrees, does this represent a small rotation of 2 degrees or a large rotation of  $-358$  degrees? For another example, consider two matrices where one is the identity and the other is a 180-degree rotation about the  $z$  axis. There are an infinite number of ways to go from one orientation to the other.

Keyframe matrix interpolation is an important problem in computer animation, where a number of different approaches have been developed. Fortunately, the problem of matrix interpolation in a renderer is generally less challenging than it is for animation systems for two reasons.

First, in a renderer like `pbrt`, we generally have a keyframe matrix at the camera shutter open time and another at the shutter close time; we only need to interpolate between the two of them across the time of a single image. In animation systems, the matrices are generally available at a lower time frequency, so that there are many frames between

---

\* This section covers advanced topics and may be skipped on a first reading.



**Figure 2.16:** (left) Sphere with a grid of lines as a texture, not rotating. (middle) Sphere rotating 90 degrees during the course of the frame, using the technique for interpolating transformations implemented in this section. (right) Sphere rotating 90 degrees using direct interpolation of matrix components to interpolate transformations. In this case, the animated sphere incorrectly grows larger. Furthermore, the lines toward the outside of the sphere, which should remain sharp, incorrectly become blurry.

pairs of keyframe matrices; as such, there's more opportunity to notice shortcomings in the interpolation.

Second, in a physically based renderer, the longer the period of time over which we need to interpolate the pair of matrices, the longer the virtual camera shutter is open and the more motion blur there will be in the final image; the increased amount of motion blur often hides sins of the interpolation.

The most straightforward approach to interpolate transformations defined by keyframe matrices—directly interpolating the individual components of the matrices—is not a good one, as it will generally lead to unexpected and undesirable results. For example, if the transformations apply different rotations, then even if we have a rigid-body motion, the intermediate matrices may scale the object, which is clearly undesirable. (If the matrices have a full 180-degree rotation between them, the object may be scaled down to nothing at the middle of the interpolation!)

Figure 2.16 shows a sphere that rotates 90 degrees over the course of the frame; direct interpolation of matrix elements (on the right) gives a less accurate result than the approach implemented in this section (in the middle).

The approach used for transformation interpolation in pbrt is based on *matrix decomposition*—given an arbitrary transformation matrix  $M$ , we decompose it into a concatenation of scale ( $S$ ), rotation ( $R$ ), and translation ( $T$ ) transformations,

$$M = SRT,$$

where each of those components is independently interpolated and then the composite interpolated matrix is found by multiplying the three interpolated matrices together.

Interpolation of translation and scale can be performed easily and accurately with linear interpolation of the components of their matrices; interpolating rotations is more difficult. Before describing the matrix decomposition implementation in pbrt, we will

first introduce *quaternions*, an elegant representation of rotations that leads to effective methods for interpolating them.

### 2.9.1 QUATERNIONS

Quaternions were originally invented by Sir William Rowan Hamilton in 1843 as a generalization of complex numbers. He determined that just as in two dimensions ( $x, y$ ), where complex numbers could be defined as a sum of a real and an imaginary part  $x + yi$ , with  $i^2 = -1$ , a generalization could be made to four dimensions, giving quaternions.

A quaternion is a four-tuple,

$$\mathbf{q} = (x, y, z, w) = w + xi + yj + zk, \quad [2.4]$$

where  $i, j$ , and  $k$  are defined<sup>2</sup> so that  $i^2 = j^2 = k^2 = ijk = -1$ . Other important relationships between the components are that  $ij = k$  and  $ji = -k$ . This implies that quaternion multiplication is generally not commutative.

A quaternion can be represented as a quadruple  $\mathbf{q} = (q_x, q_y, q_z, q_w)$  or as  $\mathbf{q} = (q_{xyz}, q_w)$ , where  $q_{xyz}$  is an imaginary 3-vector and  $q_w$  is the real part. We will use both representations interchangeably in this section.

An expression for the product of two arbitrary quaternions can be found by expanding their definition in terms of real and imaginary components:

$$\mathbf{qq}' = (q_w + q_x i + q_y j + q_z k)(q'_w + q'_x i + q'_y j + q'_z k).$$

Collecting terms and using identities among the components like those listed above (e.g.,  $i^2 = -1$ ), the result can be expressed concisely using vector cross and dot products:

$$\begin{aligned} (\mathbf{qq}')_{xyz} &= q_{xyz} \times q'_{xyz} + q_w q'_{xyz} + q'_w q_{xyz} \\ (\mathbf{qq}')_w &= q_w q'_w - (q_{xyz} \cdot q'_{xyz}). \end{aligned} \quad [2.5]$$

There is a useful relationship between unit quaternions (quaternions whose components satisfy  $x^2 + y^2 + z^2 + w^2 = 1$ ) and the space of rotations in  $\mathbb{R}^3$ : specifically, a rotation of angle  $2\theta$  about a unit axis  $\hat{\mathbf{v}}$  can be mapped to a unit quaternion  $(\hat{\mathbf{v}} \sin \theta, \cos \theta)$ , in which case the following quaternion product is equivalent to applying the rotation to a point  $\mathbf{p}$  expressed in homogeneous coordinate form:

$$\mathbf{p}' = \mathbf{qpq}^{-1}.$$

Furthermore, the product of several rotation quaternions produces another quaternion that is equivalent to applying the rotations in sequence.

The implementation of the `Quaternion` class in `pbrt` is in the files `core/quaternion.h` and `core/quaternion.cpp`. The default constructor initializes a unit quaternion.

*(Quaternion Public Methods) ≡*

```
Quaternion 99
Quaternion::v 100
Quaternion::w 100
```

---

<sup>2</sup> Hamilton found the discovery of this relationship among the components compelling enough that he used a knife to carve the formula on the bridge he was crossing when it came to him.

We use a `Vector3f` to represent the  $xyz$  components of the quaternion; doing so lets us make use of various methods of `Vector3f` in the implementation of some of the methods below.

*(Quaternion Public Data)  $\equiv$*

```
Vector3f v;
Float w;
```

Addition and subtraction of quaternions is performed component-wise. This follows directly from the definition in Equation (2.4). For example,

$$\begin{aligned} \mathbf{q} + \mathbf{q}' &= w + xi + yj + zk + w' + x'i + y'j + z'k \\ &= (w + w') + (x + x')i + (y + y')j + (z + z')k. \end{aligned}$$

Other arithmetic methods (subtraction, multiplication, and division by a scalar) are defined and implemented similarly and won't be included here.

*(Quaternion Public Methods)  $+ \equiv$*

```
Quaternion &operator+=(const Quaternion &q) {
    v += q.v;
    w += q.w;
    return *this;
}
```

The inner product of two quaternions is implemented by its `Dot()` method, and a quaternion can be normalized by dividing by its length.

*(Quaternion Inline Functions)  $\equiv$*

```
inline Float Dot(const Quaternion &q1, const Quaternion &q2) {
    return Dot(q1.v, q2.v) + q1.w * q2.w;
}
```

*(Quaternion Inline Functions)  $+ \equiv$*

```
inline Quaternion Normalize(const Quaternion &q) {
    return q / std::sqrt(Dot(q, q));
}
```

It's useful to be able to compute the transformation matrix that represents the same rotation as a quaternion. In particular, after interpolating rotations with quaternions in the `AnimatedTransform` class, we'll need to convert the interpolated rotation back to a transformation matrix to compute the final composite interpolated transformation.

To derive the rotation matrix for a quaternion, recall that the transformation of a point by a quaternion is given by  $\mathbf{p}' = \mathbf{q}\mathbf{p}\mathbf{q}^{-1}$ . We want a matrix  $M$  that performs the same transformation, so that  $\mathbf{p}' = M\mathbf{p}$ . If we expand out the quaternion multiplication  $\mathbf{q}\mathbf{p}\mathbf{q}^{-1}$  using Equation (2.5), simplify with the quaternion basis identities, collect terms, and represent the result in a matrix, we can determine that the following  $3 \times 3$  matrix represents the same transformation:

`AnimatedTransform` 103

`Dot()` 63

`Float` 1062

`Quaternion` 99

`Quaternion::Dot()` 100

`Quaternion::v` 100

`Quaternion::w` 100

`Vector3f` 60

$$\mathbf{M} = \begin{pmatrix} 1 - 2(\mathbf{q}_y^2 + \mathbf{q}_z^2) & 2(\mathbf{q}_x\mathbf{q}_y + \mathbf{q}_z\mathbf{q}_w) & 2\mathbf{q}_x\mathbf{q}_z - \mathbf{q}_y\mathbf{q}_w \\ 2(\mathbf{q}_x\mathbf{q}_y - \mathbf{q}_z\mathbf{q}_w) & 1 - 2(\mathbf{q}_x^2 + \mathbf{q}_z^2) & 2(\mathbf{q}_y\mathbf{q}_z + \mathbf{q}_x\mathbf{q}_w) \\ 2(\mathbf{q}_x\mathbf{q}_z + \mathbf{q}_y\mathbf{q}_w) & 2(\mathbf{q}_y\mathbf{q}_z - \mathbf{q}_x\mathbf{q}_w) & 1 - 2(\mathbf{q}_x^2 + \mathbf{q}_y^2) \end{pmatrix}. \quad [2.6]$$

This computation is implemented in the method `Quaternion::ToTransform()`. We won't include its implementation here since it's a direct implementation of Equation (2.6).

```
<Quaternion Public Methods> +≡
    Transform ToTransform() const;
```

Note that we could alternatively use the fact that a unit quaternion represents a rotation  $(\mathbf{q}_{xyz} \sin \theta, \cos \theta)$  of angle  $2\theta$  around the unit axis  $\hat{\mathbf{q}}_{xyz}$  to compute a rotation matrix. First we would compute the angle of rotation  $\theta$  as  $\theta = 2 \arccos \mathbf{q}_w$ , and then we'd use the previously defined `Rotate()` function, passing it the axis  $\hat{\mathbf{q}}_{xyz}$  and the rotation angle  $\theta$ . However, this alternative would be substantially less efficient, requiring multiple calls to trigonometric functions, while the approach implemented here only uses floating-point addition, subtraction, and multiplication.

It is also useful to be able to create a quaternion from a rotation matrix. For this purpose, `Quaternion` provides a constructor that takes a `Transform`. The appropriate quaternion can be computed by making use of relationships between elements of the rotation matrix in Equation (2.6) and quaternion components. For example, if we subtract the transpose of this matrix from itself, then the (0, 1) component of the resulting matrix has the value  $-4\mathbf{q}_w\mathbf{q}_z$ . Thus, given a particular instance of a rotation matrix with known values, it's possible to use a number of relationships like this between the matrix values and the quaternion components to generate a system of equations that can be solved for the quaternion components.

We won't include the details of the derivation or the actual implementation here in the text; for more information about how to derive this technique, including handling numerical robustness, see Shoemake (1991).

```
<Quaternion Public Methods> +≡
    Quaternion(const Transform &t);
```

## 2.9.2 QUATERNION INTERPOLATION

The last quaternion function we will define, `Slerp()`, interpolates between two quaternions using spherical linear interpolation. Spherical linear interpolation gives constant speed motion along great circle arcs on the surface of a sphere and consequently has two desirable properties for interpolating rotations:

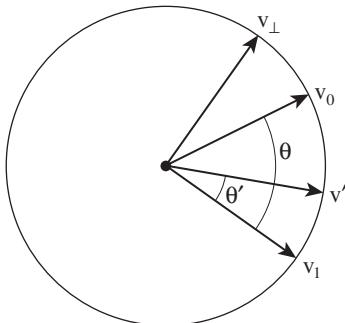
- The interpolated rotation path exhibits *torque minimization*: the path to get between two rotations is the shortest possible path in rotation space.
- The interpolation has *constant angular velocity*: the relationship between change in the animation parameter  $t$  and the change in the resulting rotation is constant over the course of interpolation (in other words, the speed of interpolation is constant across the interpolation range).

`Quaternion` 99

`Rotate()` 91

`Slerp()` 103

`Transform` 83



**Figure 2.17:** To understand quaternion spherical linear interpolation, consider in 2D two vectors on the unit sphere,  $v_0$  and  $v_1$ , with angle  $\theta$  between them. We'd like to be able to compute the interpolated vector at some angle  $\theta'$  between the two of them. To do so, we can find a vector that's orthogonal to  $v_1$ ,  $v_{\perp}$  and then apply the trigonometric identity  $v' = v_1 \cos \theta' + v_{\perp} \sin \theta'$ .

See the “Further Reading” section at the end of the chapter for references that discuss more thoroughly what characteristics a good interpolated rotation should have.

Spherical linear interpolation for quaternions was originally presented by Shoemake (1985) as follows, where two quaternions  $q_1$  and  $q_2$  are given and  $t \in [0, 1]$  is the parameter value to interpolate between them:

$$\text{slerp}(q_1, q_2, t) = \frac{q_1 \sin((1-t)\theta) + q_2 \sin(t\theta)}{\sin \theta}.$$

An intuitive way to understand Slerp() was presented by Blow (2004). As context, given the quaternions to interpolate between,  $q_1$  and  $q_2$ , denote by  $\theta$  the angle between them. Then, given a parameter value  $t \in [0, 1]$ , we'd like to find the intermediate quaternion  $q'$  that makes angle  $\theta' = \theta t$  between it and  $q_1$ , along the path from  $q_1$  to  $q_2$ .

An easy way to compute  $q'$  is to first compute an orthogonal coordinate system in the space of quaternions where one axis is  $q_1$  and the other is a quaternion orthogonal to  $q_1$  such that the two axes form a basis that spans  $q_1$  and  $q_2$ . Given such a coordinate system, we can compute rotations with respect to  $q_1$ . (See Figure 2.17, which illustrates the concept in the 2D setting.) An orthogonal vector  $q_{\perp}$  can be found by projecting  $q_1$  onto  $q_2$  and then subtracting the orthogonal projection from  $q_2$ ; the remainder is guaranteed to be orthogonal to  $q_1$ :

$$q_{\perp} = q_2 - (q_1 \cdot q_2)q_1. \quad [2.7]$$

Given the coordinate system, quaternions along the animation path are given by

$$q' = q_1 \cos(\theta t) + q_{\perp} \sin(\theta t). \quad [2.8]$$

The implementation of the Slerp() function checks to see if the two quaternions are nearly parallel, in which case it uses regular linear interpolation of quaternion components in order to avoid numerical instability. Otherwise, it computes an orthogonal

quaternion qerp using Equation (2.7) and then computes the interpolated quaternion with Equation (2.8).

*(Quaternion Method Definitions)* ≡

```
Quaternion Slerp(Float t, const Quaternion &q1,
                  const Quaternion &q2) {
    Float cosTheta = Dot(q1, q2);
    if (cosTheta > .9995f)
        return Normalize((1 - t) * q1 + t * q2);
    else {
        Float theta = std::acos(Clamp(cosTheta, -1, 1));
        Float thetap = theta * t;
        Quaternion qerp = Normalize(q2 - q1 * cosTheta);
        return q1 * std::cos(thetap) + qerp * std::sin(thetap);
    }
}
```

### 2.9.3 AnimatedTransform IMPLEMENTATION

Given the foundations of the quaternion infrastructure, we can now implement the `AnimatedTransform` class, which implements keyframe transformation interpolation in `pbrt`. Its constructor takes two transformations and the time values they are associated with.

As mentioned earlier, `AnimatedTransform` decomposes the given composite transformation matrices into scaling, rotation, and translation components. The decomposition is performed by the `AnimatedTransform::Decompose()` method.

*(AnimatedTransform Method Definitions)* ≡

```
AnimatedTransform::AnimatedTransform(const Transform *startTransform,
                                     Float startTime, const Transform *endTransform, Float endTime)
: startTransform(startTransform), endTransform(endTransform),
  startTime(startTime), endTime(endTime),
  actuallyAnimated(*startTransform != *endTransform) {
    Decompose(startTransform->m, &T[0], &R[0], &S[0]);
    Decompose(endTransform->m, &T[1], &R[1], &S[1]);
    {Flip R[1] if needed to select shortest path 106}
    hasRotation = Dot(R[0], R[1]) < 0.9995f;
    {Compute terms of motion derivative function}
}
```

*(AnimatedTransform Private Data)* ≡

```
const Transform *startTransform, *endTransform;
const Float startTime, endTime;
const bool actuallyAnimated;
Vector3f T[2];
Quaternion R[2];
Matrix4x4 S[2];
bool hasRotation;
```

AnimatedTransform 103  
 AnimatedTransform:::Decompose() 104  
 AnimatedTransform:::hasRotation 103  
 Clamp() 1062  
 Float 1062  
 Matrix4x4 1081  
 Quaternion 99  
 Quaternion::Dot() 100  
 Quaternion::Normalize() 100  
 Transform 83  
 Vector3f 60

Given the composite matrix for a transformation, information has been lost about any individual transformations that were composed to compute it. For example, given the matrix for the product of a translation and then a scale, an equal matrix could also be computed by first scaling and then translating (by different amounts). Thus, we need to choose a canonical sequence of transformations for the decomposition. For our needs here, the specific choice made isn't significant. (It would be more important in an animation system that was decomposing composite transformations in order to make them editable by changing individual components, for example.)

We will handle only affine transformations here, which is what is needed for animating cameras and geometric primitives in a rendering system; perspective transformations aren't generally relevant to animation of objects like these.

The transformation decomposition we will use is the following:

$$M = TRS, \quad [2.9]$$

where  $M$  is the given transformation,  $T$  is a translation,  $R$  is a rotation, and  $S$  is a scale.  $S$  is actually a generalized scale (Shoemake and Duff call it *stretch*) that represents a scale in *some* coordinate system, just not necessarily the current one. In any case, it can still be correctly interpolated with linear interpolation of its components. The `Decompose()` method computes the decomposition given a `Matrix4x4`.

*(AnimatedTransform Method Definitions)* +≡

```
void AnimatedTransform::Decompose(const Matrix4x4 &m, Vector3f *T,
    Quaternion *Rquat, Matrix4x4 *S) {
    <Extract translation T from transformation matrix 104>
    <Compute new transformation matrix M without translation 104>
    <Extract rotation R from transformation matrix 105>
    <Compute scale S using rotation and original matrix 105>
}
```

Extracting the translation  $T$  is easy; it can be found directly from the appropriate elements of the  $4 \times 4$  transformation matrix.

*(Extract translation T from transformation matrix)* ≡

104

```
T->x = m.m[0][3];
T->y = m.m[1][3];
T->z = m.m[2][3];
```

Since we are assuming an affine transformation (no projective components), after we remove the translation, what is left is the upper  $3 \times 3$  matrix that represents scaling and rotation together. This matrix is copied into a new matrix  $M$  for further processing.

*(Compute new transformation matrix M without translation)* ≡

104

```
Matrix4x4 M = m;
for (int i = 0; i < 3; ++i)
    M.m[i][3] = M.m[3][i] = 0.f;
M.m[3][3] = 1.f;
```

Matrix4x4 1081  
Matrix4x4::m 1081  
Quaternion 99  
Vector3f 60

Next we'd like to extract the pure rotation component of  $M$ . We'll use a technique called *polar decomposition* to do this. It can be shown that the polar decomposition of a matrix

$M$  into rotation  $R$  and scale  $S$  can be computed by successively averaging  $M$  with its inverse transpose

$$M_{i+1} = \frac{1}{2} \left( M_i + (M_i^T)^{-1} \right) \quad (2.10)$$

until convergence, at which point  $M_i = R$ . (It's easy to see that if  $M$  is a pure rotation, then averaging it with its inverse transpose will leave it unchanged, since its inverse is equal to its transpose. The "Further Reading" section has more references that discuss why this series converges to the rotation component of the original transformation.) Shoemake and Duff (1992) proved that the resulting matrix is the closest orthogonal matrix to  $M$ —a desirable property.

To compute this series, we iteratively apply Equation (2.10) until either the difference between successive terms is small or a fixed number of iterations have been performed. In practice, this series generally converges quickly.

```

⟨Extract rotation R from transformation matrix⟩ ≡ 104
    Float norm;
    int count = 0;
    Matrix4x4 R = M;
    do {
        ⟨Compute next matrix Rnext in series 105⟩
        ⟨Compute norm of difference between R and Rnext 105⟩
        R = Rnext;
    } while (++count < 100 && norm > .0001);
    *Rquat = Quaternion(R);

⟨Compute next matrix Rnext in series⟩ ≡ 105
    Matrix4x4 Rnext;
    Matrix4x4 Rit = Inverse(Transpose(R));
    for (int i = 0; i < 4; ++i)
        for (int j = 0; j < 4; ++j)
            Rnext.m[i][j] = 0.5f * (R.m[i][j] + Rit.m[i][j]);

⟨Compute norm of difference between R and Rnext⟩ ≡ 105
    norm = 0;
    for (int i = 0; i < 3; ++i) {
        Float n = std::abs(R.m[i][0] - Rnext.m[i][0]) +
                  std::abs(R.m[i][1] - Rnext.m[i][1]) +
                  std::abs(R.m[i][2] - Rnext.m[i][2]);
        norm = std::max(norm, n);
    }

Float 1062
Inverse() 1081
Matrix4x4 1081
Matrix4x4::m 1081
Matrix4x4::Mul() 1081
Quaternion 99
Transform::Transpose() 85

```

Once we've extracted the rotation from  $M$ , the scale is all that's left. We would like to find the matrix  $S$  that satisfies  $M = RS$ . Now that we know both  $R$  and  $M$ , we just solve for  $S = R^{-1}M$ .

```

⟨Compute scale S using rotation and original matrix⟩ ≡ 104
    *S = Matrix4x4::Mul(Inverse(R), M);

```

For every rotation matrix, there are two unit quaternions that correspond to the matrix that only differ in sign. If the dot product of the two rotations that we have extracted is negative, then a slerp between them won't take the shortest path between the two corresponding rotations. Negating one of them (here the second was chosen arbitrarily) causes the shorter path to be taken instead.

```
{Flip R[1] if needed to select shortest path} ≡ 103
if (Dot(R[0], R[1]) < 0)
    R[1] = -R[1];
```

The Interpolate() method computes the interpolated transformation matrix at a given time. The matrix is found by interpolating the previously extracted translation, rotation, and scale and then multiplying them together to get a composite matrix that represents the effect of the three transformations together.

```
{AnimatedTransform Method Definitions} +≡
void AnimatedTransform::Interpolate(Float time, Transform *t) const {
    {Handle boundary conditions for matrix interpolation 106}
    Float dt = (time - startTime) / (endTime - startTime);
    {Interpolate translation at dt 106}
    {Interpolate rotation at dt 107}
    {Interpolate scale at dt 107}
    {Compute interpolated matrix as product of interpolated components 107}
}
```

If the given time value is outside the time range of the two transformations stored in the AnimatedTransform, then the transformation at the start time or end time is returned, as appropriate. The AnimatedTransform constructor also checks whether the two Transforms stored are the same; if so, then no interpolation is necessary either. All of the classes in pbrt that support animation always store an AnimatedTransform for their transformation, rather than storing either a Transform or AnimatedTransform as appropriate. This simplifies their implementations, though it does make it worthwhile to check for this case here and not unnecessarily do the work to interpolate between two equal transformations.

```
{Handle boundary conditions for matrix interpolation} ≡ 106
if (!actuallyAnimated || time <= startTime) {
    *t = *startTransform;
    return;
}
if (time >= endTime) {
    *t = *endTransform;
    return;
}
```

```
AnimatedTransform 103
AnimatedTransform::
    actuallyAnimated
    103
AnimatedTransform::endTime
    103
AnimatedTransform::endTransform
    103
AnimatedTransform::R 103
AnimatedTransform::startTime
    103
AnimatedTransform::
    startTransform
    103
AnimatedTransform::T 103
Float 1062
Quaternion::Dot() 100
Transform 83
Vector3f 60
```

The dt variable stores the offset in the range from startTime to endTime; it is zero at startTime and one at endTime. Given dt, interpolation of the translation is trivial.

```
{Interpolate translation at dt} ≡ 106
Vector3f trans = (1 - dt) * T[0] + dt * T[1];
```

The rotation is interpolated between the start and end rotations using the `Slerp()` routine (Section 2.9.2).

```
(Interpolate rotation at dt) ≡ 106
    Quaternion rotate = Slerp(dt, R[0], R[1]);
```

Finally, the interpolated scale matrix is computed by interpolating the individual elements of the start and end scale matrices. Because the `Matrix4x4` constructor sets the matrix to the identity matrix, we don't need to initialize any of the other elements of scale.

```
(Interpolate scale at dt) ≡ 106
    Matrix4x4 scale;
    for (int i = 0; i < 3; ++i)
        for (int j = 0; j < 3; ++j)
            scale.m[i][j] = Lerp(dt, S[0].m[i][j], S[1].m[i][j]);
```

Given the three interpolated parts, the product of their three transformation matrices gives us the final result.

```
(Compute interpolated matrix as product of interpolated components) ≡ 106
    *t = Translate(trans) * rotate.ToTransform() * Transform(scale);
```

`AnimatedTransform` also provides a number of methods that apply interpolated transformations directly, using the provided time for `Point3fs` and `Vector3fs` and `Ray::time` for Rays. These methods are more efficient than calling `AnimatedTransform::Interpolate()` and then using the returned matrix when there is no actual animation since a copy of the transformation matrix doesn't need to be made in that case.

```
AnimatedTransform 103
AnimatedTransform::Interpolate() 106
AnimatedTransform::MotionBounds() 108
AnimatedTransform::R 103
AnimatedTransform::S 103
Float 1062
Matrix4x4 1081
Point3f 68
Quaternion 99
Quaternion::ToTransform() 101
Ray 73
Ray::time 73
RayDifferential 75
Slerp() 103
Transform 83
Translate() 87
Vector3f 60
```

*(AnimatedTransform Public Methods) ≡*

```
Ray operator()(const Ray &r) const;
RayDifferential operator()(const RayDifferential &r) const;
Point3f operator()(Float time, const Point3f &p) const;
Vector3f operator()(Float time, const Vector3f &v) const;
```

## 2.9.4 BOUNDING MOVING BOUNDING BOXES

Given a `Bounds3f` that is transformed by an animated transformation, it's useful to be able to compute a bounding box that encompasses all of its motion over the animation time period. For example, if we can bound the motion of an animated geometric primitive, then we can intersect rays with this bound to determine if the ray might intersect the object before incurring the cost of interpolating the primitive's bound to the ray's time to check that intersection. The `AnimatedTransform::MotionBounds()` method performs this computation, taking a bounding box and returning the bounding box of its motion over the `AnimatedTransform`'s time range.

There are two easy cases: first, if the keyframe matrices are equal, then we can arbitrarily apply only the starting transformation to compute the full bounds. Second, if the transformation only includes scaling and/or translation, then the bounding box that encompasses the bounding box's transformed positions at both the start time and the end time bounds all of its motion. To see why this is so, consider the position of a transformed

point  $p$  as a function of time; we'll denote this function of two matrices, a point, and a time by  $a(M_0, M_1, p, t)$ .

Since in this case the rotation component of the decomposition is the identity, then with our matrix decomposition we have

$$a(M_0, M_1, p, t) = T(t)S(t)p,$$

where the translation and scale are both written as functions of time. Assuming for simplicity that  $S(t)$  is a regular scale, we can find expressions for the components of  $a(M_0, M_1, p, t)$ . For example, for the  $x$  component, we have:

$$\begin{aligned} a(M_0, M_1, p, t)_x &= [(1-t)s_{0,0} + ts'_{0,0}]p_x + (1-t)d_{0,3} + td'_{0,3} \\ &= [s_{0,0}p_x + d_{0,3}] + [-s_{0,0}p_x + s'_{0,0}p_x - d_{0,3} + d'_{0,3}]t, \end{aligned}$$

where  $s_{0,0}$  is the corresponding element of the scale matrix for  $M_0$ ,  $s'_{0,0}$  is the same scale matrix element for  $M_1$ , and the translation matrix elements are similarly denoted by  $d$ . (We chose  $d$  for “delta” here since  $t$  is already claimed for time.) As a linear function of  $t$ , the extrema of this function are at the start and end times. The other coordinates and the case for a generalized scale follow similarly.

*(AnimatedTransform Method Definitions)  $\equiv$*

```
Bounds3f AnimatedTransform::MotionBounds(const Bounds3f &b) const {
    if (!actuallyAnimated)
        return (*startTransform)(b);
    if (hasRotation == false)
        return Union((*startTransform)(b), (*endTransform)(b));
    {Return motion bounds accounting for animated rotation 108}
}
```

For the general case with animated rotations, the motion function may have extrema at points in the middle of the time range. We know of no simple way to find these points. Many renderers address this issue by sampling a large number of times in the time range, computing the interpolated transformation at each one, and taking the union of all of the corresponding transformed bounding boxes. Here, we will develop a more well-grounded method that lets us robustly compute these motion bounds.

We use a slightly simpler conservative bound that entails computing the motion of the eight corners of the bounding box individually and finding the union of those bounds.

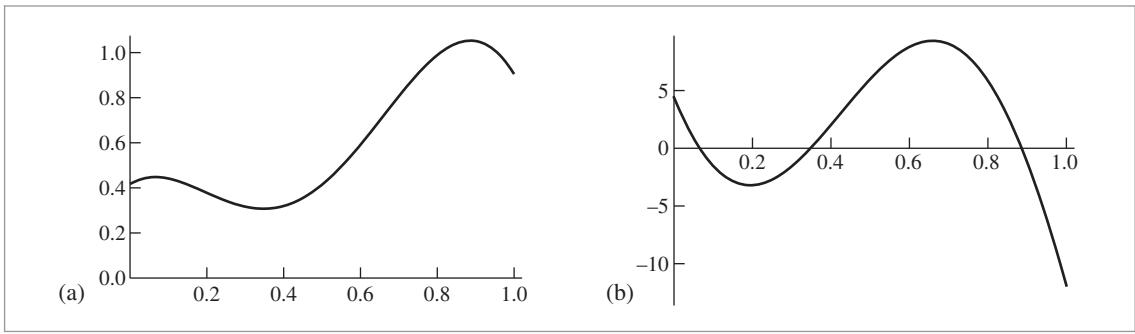
*(Return motion bounds accounting for animated rotation)  $\equiv$*

108

```
Bounds3f bounds;
for (int corner = 0; corner < 8; ++corner)
    bounds = Union(bounds, BoundPointMotion(b.Corner(corner)));
return bounds;
```

```
AnimatedTransform::  
actuallyAnimated  
103  
  
AnimatedTransform::  
BoundPointMotion()  
110  
  
AnimatedTransform::  
hasRotation  
103  
  
Bounds3::Corner() 78  
Bounds3::Union() 78  
Bounds3f 76
```

For each bounding box corner  $p$ , we need to find the extrema of  $a$  over the animation time range. Recall from calculus that the extrema of a continuous function over some domain are either at the boundary points of the domain or at points where the function's first derivative is zero. Thus, the overall bound is given by the union of the positions at the start and end of motion as well as the position at any extrema.



**Figure 2.18:** (a) Motion of the  $x$  coordinate of a point  $p$  as a function of time, as determined by two keyframe matrices. (b) The derivative of the motion function, Equation (2.12). Note that extrema of the motion function in the given time range correspond to zeros of the derivative.

Figure 2.18 shows a plot of one coordinate of the motion function and its derivative for an interesting motion path of a point. Note that the maximum value of the function over the time range is reached at a point where the derivative has a zero.

To bound the motion of a single point, we start our derivation by following the approach used for the no-rotation case, expanding out the three  $T$ ,  $R$ , and  $S$  components of Equation (2.9) as functions of time and finding their product. We have:

$$a(\mathbf{M}_0, \mathbf{M}_1, \mathbf{p}, t) = T(t)R(t)S(t)\mathbf{p}. \quad [2.11]$$

The result is quite complex when expanded out, mostly due to the slerp and the conversion of the resulting quaternion to a matrix; a computer algebra system is a requirement for working with this function.

The derivative  $\partial a(\mathbf{M}_0, \mathbf{M}_1, \mathbf{p}, t)/\partial t$  is also quite complex—in its full algebraic glory, over 2,000 operations are required to evaluate its value for a given pair of decomposed matrices, point and time. However, given specific transformation matrices and a specific point,  $a$  is simplified substantially; we'll denote the specialized function of  $t$  alone as  $a_{\mathbf{M}, \mathbf{p}}(t)$ . Evaluating its derivative requires roughly 10 floating-point operations, a sine, and a cosine to evaluate for each coordinate:

$$\frac{da_{\mathbf{M}, \mathbf{p}}(t)}{dt} = c_1 + (c_2 + c_3 t) \cos(2\theta t) + (c_4 + c_5 t) \sin(2\theta t), \quad [2.12]$$

where  $\theta$  is the arc cosine of the dot product of the two quaternions and where the five coefficients  $c_i$  are 3-vectors that depend on the two matrices and the position  $\mathbf{p}$ . This specialization works out well, since we will need to evaluate the function at many time values for a given point.

We now have two tasks: first, given a pair of keyframe matrices and a point  $\mathbf{p}$ , we first need to be able to efficiently compute the values of the coefficients  $c_i$ . Then, given the relatively simple function defined by  $c_i$  and  $\theta$ , we need to find the zeros of Equation (2.12), which may represent the times at which motion extrema occur.

For the first task, we will first factor out the contributions to the coefficients that depend on the keyframe matrices from those that depend on the point  $p$ , under the assumption that bounding boxes for multiple points' motion will be computed for each pair of keyframe matrices (as is the case here). The result is fortunately quite simple—the  $c_i$  vectors are linear functions of the point's  $x$ ,  $y$ , and  $z$  components.

$$c_i(p) = k_{i,c} + k_{i,x}p_x + k_{i,y}p_y + k_{i,z}p_z.$$

Thus, given the  $k_i$  coefficients and a particular point  $p$  we want to bound the motion of, we can efficiently compute the coefficients  $c_i$  of the derivative function in Equation (2.12). The `DerivativeTerm` structure encapsulates these coefficients and this computation.

```
(AnimatedTransform Private Data) +≡
struct DerivativeTerm {
    DerivativeTerm(Float c, Float x, Float y, Float z)
        : kc(c), kx(x), ky(y), kz(z) { }
    Float kc, kx, ky, kz;
    Float Eval(const Point3f &p) const {
        return kc + kx * p.x + ky * p.y + kz * p.z;
    }
};
```

The attributes `c1-c5` store derivative information corresponding to the five terms in Equation (2.12). The three array elements correspond to the three dimensions of space.

```
(AnimatedTransform Private Data) +≡
DerivativeTerm c1[3], c2[3], c3[3], c4[3], c5[3];
```

The fragment *(Compute terms of motion derivative function)* in the `AnimatedTransform` constructor, not included here, initializes these terms, via automatically generated code. Given that it requires a few thousand floating-point operations, doing this work once and amortizing over the multiple bounding box corners is helpful. The  $k_i$  coefficients are more easily computed if we assume a canonical time range  $[0, 1]$ ; later, we'll have to remap the  $t$  values of zeros of the motion function to the actual shutter time range.

Given the coefficients  $k_i$  based on the keyframe matrices, `BoundPointMotion()` computes a robust bound of the motion of  $p$ .

```
(AnimatedTransform Method Definitions) +≡
Bounds3f AnimatedTransform::BoundPointMotion(const Point3f &p) const {
    Bounds3f bounds((*startTransform)(p), (*endTransform)(p));
    Float cosTheta = Dot(R[0], R[1]);
    Float theta = std::acos(Clamp(cosTheta, -1, 1));
    for (int c = 0; c < 3; ++c) {
        <Find any motion derivative zeros for the component c 111>
        <Expand bounding box for any motion derivative zeros found 111>
    }
    return bounds;
}
```

`Bounds3f` 76

`DerivativeTerm` 110

`Float` 1062

`Point3f` 68

The `IntervalFindZeros()` function, to be introduced shortly, numerically finds zeros of Equation (2.12). Up to four are possible.

*(Find any motion derivative zeros for the component c) ≡*

```
110
Float zeros[4];
int nZeros = 0;
IntervalFindZeros(c1[c].Eval(p), c2[c].Eval(p), c3[c].Eval(p),
                   c4[c].Eval(p), c5[c].Eval(p), theta,
                   Interval(0., 1.), zeros, &nZeros);
```

The zeros are found over  $t \in [0, 1]$ , so we need to interpolate within the time range before calling the method to transform the point at the corresponding time. Note also that the extremum is only at one of the  $x$ ,  $y$ , and  $z$  dimensions, and so the bounds only need to be updated in that one dimension. For convenience, here we just use the `Union()` function, which considers all dimensions, even though two could be ignored.

*(Expand bounding box for any motion derivative zeros found) ≡*

```
110
for (int i = 0; i < nZeros; ++i) {
    Point3f pz = (*this)(Lerp(zeros[i], startTime, endTime), p);
    bounds = Union(bounds, pz);
}
```

Finding zeros of the motion derivative function, Equation (2.12), can't be done algebraically; numerical methods are necessary. Fortunately, the function is well behaved—it's fairly smooth and has a limited number of zeros. (Recall the plot in Figure 2.18, which was an unusually complex representative.)

While we could use a bisection-based search or Newton's method, we'd risk missing zeros when the function only briefly crosses the axis. Therefore, we'll use *interval arithmetic*, an extension of arithmetic that gives insight about the behavior of functions over ranges of values, which makes it possible to robustly find zeros of functions.

To understand the basic idea of interval arithmetic, consider, for example, the function  $f(x) = 2x$ . If we have an interval of values  $[a, b] \in \mathbb{R}$ , then we can see that over the interval, the range of  $f$  is the interval  $[2a, 2b]$ . In other words  $f([a, b]) \subset [2a, 2b]$ .

More generally, all of the basic operations of arithmetic have *interval extensions* that describe how they operate on intervals. For example, given two intervals  $[a, b]$  and  $[c, d]$ ,

$$[a, b] + [c, d] \subset [a + c, b + d].$$

In other words, if we add together two values where one is in the range  $[a, b]$  and the second is in  $[c, d]$ , then the result must be in the range  $[a + c, b + d]$ .

Interval arithmetic has the important property that the intervals that it gives are conservative. In particular, if  $f([a, b]) \subset [c, d]$  and if  $c > 0$ , then we know for sure that no value in  $[a, b]$  causes  $f$  to be negative. In the following, we will show how to compute Equation (2.12) over intervals and will take advantage of the conservative bounds of computed intervals to efficiently find small intervals with zero crossings where regular root finding methods can be reliably used.

First we will define an `Interval` class that represents intervals of real numbers.

Bounds3::Union() 78  
 Float 1062  
 Interval 112  
 IntervalFindZeros() 113  
 Lerp() 1079  
 Point3f 68

```
{Interval Definitions} ≡
class Interval {
public:
{Interval Public Methods 112}
    float low, high;
};
```

An interval can be initialized with a single value, representing a single point on the real number line, or with two values that specify an interval with non-zero width.

```
{Interval Public Methods} ≡ 112
    Interval(float v) : low(v), high(v) { }
    Interval(float v0, float v1)
        : low(std::min(v0, v1)), high(std::max(v0, v1)) { }
```

The class also provides overloads for the basic arithmetic operations. Note that for subtraction, the high value of the second interval is subtracted from the low value of the first.<sup>3</sup>

```
{Interval Public Methods} +≡ 112
    Interval operator+(const Interval &i) const {
        return Interval(low + i.low, high + i.high);
    }
    Interval operator-(const Interval &i) const {
        return Interval(low - i.high, high - i.low);
    }
```

For multiplication, which sides of each interval determine the minimum and maximum values of the result interval depend on the signs of the respective values. Multiplying the various possibilities and taking the overall minimum and maximum is easier than working through which ones to use and multiplying these.

```
{Interval Public Methods} *≡ 112
    Interval operator*(const Interval &i) const {
        return Interval(std::min(std::min(low * i.low, high * i.low),
                               std::min(low * i.high, high * i.high)),
                      std::max(std::max(low * i.low, high * i.low),
                               std::max(low * i.high, high * i.high)));
    }
```

We have also implemented `Sin()` and `Cos()` functions for `Intervals`. The implementations assume that the given interval is in  $[0, 2\pi]$ , which is the case for our use of these functions. Here we only include the implementation of `Sin()`; `Cos()` is quite similar in basic structure.

---

<sup>3</sup> Readers who have already read Section 3.9 or who are already familiar with floating-point round-off error may note a crack in our claims of robustness: when the floating-point value of one of the interval bounds is computed, the result is rounded to the nearest floating-point value, which may be larger or smaller than the fully precise result. To be fully robust, the floating-point rounding mode must be set to round down for the lower value of the extent and to round up for the upper. Changing the rounding mode is generally fairly expensive on modern CPUs, and this issue is a very minor one for this application; therefore our implementation leaves the rounding mode unchanged.

Float 1062

Interval 112

Interval::high 112

Interval::low 112

```

⟨Interval Definitions⟩ +≡
    inline Interval Sin(const Interval &i) {
        Float sinLow = std::sin(i.low), sinHigh = std::sin(i.high);
        if (sinLow > sinHigh)
            std::swap(sinLow, sinHigh);
        if (i.low < Pi / 2 && i.high > Pi / 2)
            sinHigh = 1.;
        if (i.low < (3.f / 2.f) * Pi && i.high > (3.f / 2.f) * Pi)
            sinLow = -1.;
        return Interval(sinLow, sinHigh);
    }
}

```

Given the interval machinery, we can now implement the `IntervalFindZeros()` function, which finds the  $t$  values of any zero crossings of Equation (2.12) over the given interval `tInterval`.

```

⟨Interval Definitions⟩ +≡
    void IntervalFindZeros(Float c1, Float c2, Float c3, Float c4,
                           Float c5, Float theta, Interval tInterval, Float *zeros,
                           int *zeroCount, int depth = 8) {
        ⟨Evaluate motion derivative in interval form, return if no zeros 113⟩
        if (depth > 0) {
            ⟨Split tInterval and check both resulting intervals 114⟩
        } else {
            ⟨Use Newton's method to refine zero 114⟩
        }
    }
}

```

The function starts by computing the interval range over `tInterval`. If the range doesn't span zero, then there are no zeros of the function over `tInterval` and the function can return.

```

⟨Evaluate motion derivative in interval form, return if no zeros⟩ ≡
    Interval range = Interval(c1) +
                    (Interval(c2) + Interval(c3) * tInterval) *
                    Cos(Interval(2 * theta) * tInterval) +
                    (Interval(c4) + Interval(c5) * tInterval) *
                    Sin(Interval(2 * theta) * tInterval);
    if (range.low > 0. || range.high < 0. || range.low == range.high)
        return;

```

Float 1062  
 Interval 112  
`Interval::high` 112  
`Interval::low` 112  
 Pi 1063

If the interval range does span zero, then there may be one or more zeros in the interval `tInterval`, but it's also possible that there actually aren't any, since the interval bounds are conservative but not as tight as possible. The function splits `tInterval` into two parts and recursively checks the two sub-intervals. Reducing the size of the interval domain generally reduces the extent of the interval range, which may allow us to determine that there are no zeros in one or both of the new intervals.

*(Split tInterval and check both resulting intervals) ≡*

113

```
Float mid = (tInterval.low + tInterval.high) * 0.5f;
IntervalFindZeros(c1, c2, c3, c4, c5, theta,
    Interval(tInterval.low, mid), zeros, zeroCount, depth - 1);
IntervalFindZeros(c1, c2, c3, c4, c5, theta,
    Interval(mid, tInterval.high), zeros, zeroCount, depth - 1);
```

Once we have a narrow interval where the interval value of the motion derivative function spans zero, the implementation switches to a few iterations of Newton's method to find the zero, starting at the midpoint of the interval. Newton's method requires the derivative of the function; since we're finding zeros of the motion derivative function, this is the second derivative of Equation (2.11):

$$\frac{d^2a_{M,p}(t)}{dt^2}_x = c_{3,x} + 2\theta(c_{4,x} + c_{5,x}t) \cos(2\theta t) + c_{5,x} - 2\theta(c_{2,x} + c_{3,x}t) \sin(2\theta t).$$

*(Use Newton's method to refine zero) ≡*

113

```
Float tNewton = (tInterval.low + tInterval.high) * 0.5f;
for (int i = 0; i < 4; ++i) {
    Float fNewton = c1 +
        (c2 + c3 * tNewton) * std::cos(2.f * theta * tNewton) +
        (c4 + c5 * tNewton) * std::sin(2.f * theta * tNewton);
    Float fPrimeNewton =
        (c3 + 2 * (c4 + c5 * tNewton) * theta) *
        std::cos(2.f * tNewton * theta) +
        (c5 - 2 * (c2 + c3 * tNewton) * theta) *
        std::sin(2.f * tNewton * theta);
    if (fNewton == 0 || fPrimeNewton == 0)
        break;
    tNewton = tNewton - fNewton / fPrimeNewton;
}
zeros[*zeroCount] = tNewton;
(*zeroCount)++;
```

Note that if there were multiple zeros of the function in `tInterval` when Newton's method is used, then we will only find one of them here. However, because the interval is quite small at this point, the impact of this error should be minimal. In any case, we haven't found this issue to be a problem in practice.

## 2.10 INTERACTIONS

The last abstraction in this chapter, `SurfaceInteraction`, represents local information at a point on a 2D surface. For example, the ray–shape intersection routines in Chapter 3 return information about the local differential geometry at intersection points in a `SurfaceInteraction`. Later, the texturing code in Chapter 10 computes material properties given a point on a surface represented by a `SurfaceInteraction`. The closely related `MediumInteraction` class is used to represent points where light scatters in participating media like smoke or clouds; it will be defined in Section 11.3 after additional pre-

Float 1062  
 Interval 112  
 Interval::high 112  
 Interval::low 112  
 IntervalFindZeros() 113  
 MediumInteraction 688

liminaries have been introduced. The implementations of these classes are in the files `core/interaction.h` and `core/interaction.cpp`.

Both `SurfaceInteraction` and `MediumInteraction` inherit from a generic `Interaction` class, which provides some common member variables and methods. Some parts of the system (notably the light source implementations) operate with respect to `Interactions`, as the differences between surface and medium interactions don't matter to them.

```
(Interaction Declarations) ≡
    struct Interaction {
        (Interaction Public Methods 115)
        (Interaction Public Data 115)
    };
```

A number of `Interaction` constructors are available; depending on what sort of interaction is being constructed and what sort of information about it is relevant, corresponding sets of parameters are accepted. This one is the most general of them.

```
(Interaction Public Methods) ≡
    Interaction(const Point3f &p, const Normal3f &n, const Vector3f &pError,
                const Vector3f &wo, Float time,
                const MediumInterface &mediumInterface)
        : p(p), time(time), pError(pError), wo(wo), n(n),
          mediumInterface(mediumInterface) { }
```

All interactions must have a point `p` and time associated with them.

```
(Interaction Public Data) ≡
    Point3f p;
    Float time;
```

For interactions where the point `p` was computed by ray intersection, some floating-point error is generally present in the `p` value. `pError` gives a conservative bound on this error; it's  $(0, 0, 0)$  for points in participating media. See Section 3.9 for more on `pbrt`'s approach to managing floating-point error and in particular Section 3.9.4 for how this bound is computed for various shapes.

```
Float 1062
Interaction::mediumInterface 116
Interaction::n 116
Interaction::p 115
Interaction::pError 115
Interaction::time 115
Interaction::wo 115
MediumInterface 688
MediumInterface 684
Normal3f 71
Point3f 68
Vector3f 60
```

For interactions that lie along a ray (either from a ray–shape intersection or from a ray passing through participating media), the negative ray direction is stored in `wo`, which corresponds to  $\omega_o$ , the notation we use for the outgoing direction when computing lighting at points. For other types of interaction points where the notion of an outgoing direction doesn't apply (e.g., those found by randomly sampling points on the surface of shapes), `wo` has the value  $(0, 0, 0)$ .

```
(Interaction Public Data) +≡
    Vector3f wo;
```

For interactions on surfaces, `n` stores the surface normal at the point.

*(Interaction Public Data)* +≡

115

```
Normal3f n;
```

*(Interaction Public Methods)* +≡

115

```
bool IsSurfaceInteraction() const {
    return n != Normal3f();
}
```

Interactions also need to record the scattering media at their point (if any); this is handled by an instance of the `MediumInterface` class, which is defined in Section 11.3.1.

*(Interaction Public Data)* +≡

115

```
MediumInterface mediumInterface;
```

## 2.10.1 SURFACE INTERACTION

The geometry of particular point on a surface (often a position found by intersecting a ray against the surface) is represented by a `SurfaceInteraction`. Having this abstraction lets most of the system work with points on surfaces without needing to consider the particular type of geometric shape the points lie on; the `SurfaceInteraction` abstraction supplies enough information about the surface point to allow the shading and geometric operations in the rest of `pbrt` to be implemented generically.

*(SurfaceInteraction Declarations)* ≡

```
class SurfaceInteraction : public Interaction {
public:
    (SurfaceInteraction Public Methods)
    (SurfaceInteraction Public Data 116)
};
```

In addition to the point  $p$  and surface normal  $n$  from the `Interaction` base class, the `SurfaceInteraction` also stores  $(u, v)$  coordinates from the parameterization of the surface and the parametric partial derivatives of the point  $\partial p / \partial u$  and  $\partial p / \partial v$ . See Figure 2.19 for a depiction of these values. It's also useful to have a pointer to the `Shape` that the point lies on (the `Shape` class will be introduced in the next chapter) as well as the partial derivatives of the surface normal.

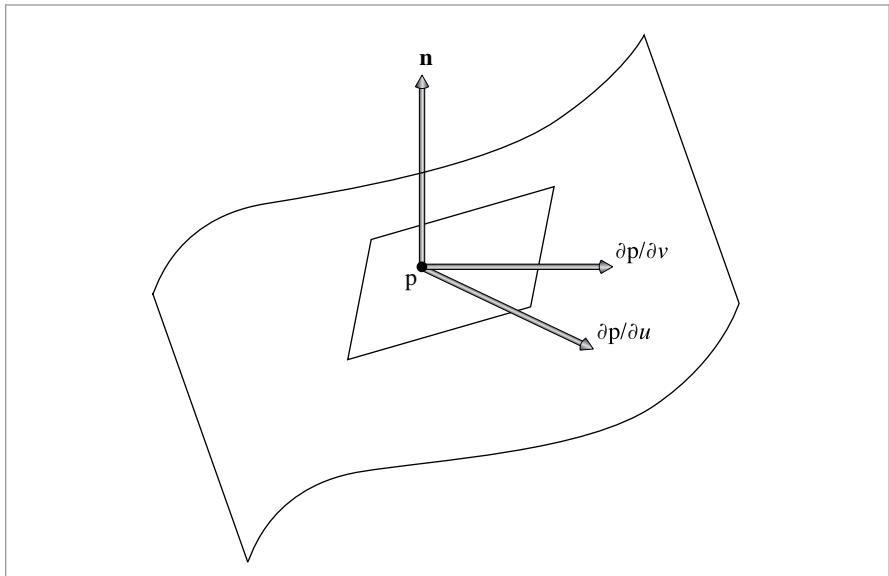
*(SurfaceInteraction Public Data)* ≡

116

```
Point2f uv;
Vector3f dpdu, dpdv;
Normal3f dndu, dndv;
const Shape *shape = nullptr;
```

This representation implicitly assumes that shapes have a parametric description—that for some range of  $(u, v)$  values, points on the surface are given by some function  $f$  such that  $p = f(u, v)$ . Although this isn't true for all shapes, all of the shapes that `pbrt` supports do have at least a local parametric description, so we will stick with the parametric representation since this assumption is helpful elsewhere (e.g., for antialiasing of textures in Chapter 10).

Interaction 115  
 Interaction::: 116  
 MediumInterface 684  
 Normal3f 71  
 Point2f 68  
 Shape 123  
 Vector3f 60



**Figure 2.19: The Local Differential Geometry around a Point  $p$ .** The parametric partial derivatives of the surface,  $\partial p / \partial u$  and  $\partial p / \partial v$ , lie in the tangent plane but are not necessarily orthogonal. The surface normal  $n$  is given by the cross product of  $\partial p / \partial u$  and  $\partial p / \partial v$ . The vectors  $\partial n / \partial u$  and  $\partial n / \partial v$  (not shown here) record the differential change in surface normal as we move  $u$  and  $v$  along the surface.

The `SurfaceInteraction` constructor takes parameters that set all of these values. It computes the normal as the cross product of the partial derivatives.

```
(SurfaceInteraction Method Definitions) ≡
    SurfaceInteraction::SurfaceInteraction(const Point3f &p,
        const Vector3f &pError, const Point2f &uv, const Vector3f &wo,
        const Vector3f &dpdu, const Vector3f &dpdv,
        const Normal3f &dndu, const Normal3f &dndv,
        float time, const Shape *shape)
    : Interaction(p, Normal3f(Normalize(Cross(dpdu, dpdv))), pError, wo,
        time, nullptr),
        uv(uv), dpdu(dpdu), dpdv(dpdv), dndu(dndu), dndv(dndv),
        shape(shape) {
(Initialize shading geometry from true geometry 118)
(Adjust normal based on orientation and handedness 119)
}
```

Cross() 65  
 Float 1062  
 Interaction 115  
 Normal3f 71  
 Point2f 68  
 Point3f 68  
 Shape 123  
 SurfaceInteraction 116  
 Vector3f::Normalize() 66  
 Vector3f 60

`SurfaceInteraction` stores a second instance of a surface normal and the various partial derivatives to represent possibly perturbed values of these quantities as can be generated by bump mapping or interpolated per-vertex normals with triangles. Some parts of the system use this shading geometry, while others need to work with the original quantities.

*(SurfaceInteraction Public Data) +≡*

116

```
struct {
    Normal3f n;
    Vector3f dpdu, dpdv;
    Normal3f dndu, dndv;
} shading;
```

The shading geometry values are initialized in the constructor to match the original surface geometry. If shading geometry is present, it generally isn't computed until some time after the SurfaceInteraction constructor runs. The SetShadingGeometry() method, to be defined shortly, updates the shading geometry.

*(Initialize shading geometry from true geometry) ≡*

117

```
shading.n = n;
shading.dpdu = dpdu;
shading.dpdv = dpdv;
shading.dndu = dndu;
shading.dndv = dndv;
```

The surface normal has special meaning to pbrt, which assumes that, for closed shapes, the normal is oriented such that it points to the outside of the shape. For geometry used as an area light source, light is emitted from only the side of the surface that the normal points toward; the other side is black. Because normals have this special meaning, pbrt provides a mechanism for the user to reverse the orientation of the normal, flipping it to point in the opposite direction. The ReverseOrientation directive in pbrt's input file flips the normal to point in the opposite, non-default direction. Therefore, it is necessary to check if the given Shape has the corresponding flag set and, if so, switch the normal's direction here.

However, one other factor plays into the orientation of the normal and must be accounted for here as well. If the Shape's transformation matrix has switched the handedness of the object coordinate system from pbrt's default left-handed coordinate system to a right-handed one, we need to switch the orientation of the normal as well. To see why this is so, consider a scale matrix  $S(1, 1, -1)$ . We would naturally expect this scale to switch the direction of the normal, although because we have computed the normal by  $\mathbf{n} = \partial \mathbf{p} / \partial u \times \partial \mathbf{p} / \partial v$ ,

$$\begin{aligned} S(1, 1, -1) \frac{\partial \mathbf{p}}{\partial u} \times S(1, 1, -1) \frac{\partial \mathbf{p}}{\partial v} &= S(-1, -1, 1) \frac{\partial \mathbf{p}}{\partial u} \times \frac{\partial \mathbf{p}}{\partial v} \\ &= S(-1, -1, 1) \mathbf{n} \\ &\neq S(1, 1, -1) \mathbf{n}. \end{aligned}$$

Therefore, it is also necessary to flip the normal's direction if the transformation switches the handedness of the coordinate system, since the flip won't be accounted for by the computation of the normal's direction using the cross product.

The normal's direction is swapped if one but not both of these two conditions is met; if both were met, their effect would cancel out. The exclusive-OR operation tests this condition.

Normal3f 71

SurfaceInteraction::
shading::dndu
118

SurfaceInteraction::
shading::dndv
118

SurfaceInteraction::
shading::dpdu
118

SurfaceInteraction::
shading::dpdv
118

SurfaceInteraction::
shading::n
118

Vector3f 60

```
<Adjust normal based on orientation and handedness> ≡ 117
    if (shape && (shape->reverseOrientation ^
                    shape->transformSwapsHandedness)) {
        n *= -1;
        shading.n *= -1;
    }
```

When a shading coordinate frame is computed, the `SurfaceInteraction` is updated via its `SetShadingGeometry()` method.

```
<SurfaceInteraction Method Definitions> +≡
void SurfaceInteraction::SetShadingGeometry(const Vector3f &dpdus,
                                             const Vector3f &dpdvs, const Normal3f &dndus,
                                             const Normal3f &dndvs, bool orientationIsAuthoritative) {
    <Compute shading.n for SurfaceInteraction 119>
    <Initialize shading partial derivative values 119>
}
```

After performing the same cross product (and possibly flipping the orientation of the normal) as before to compute an initial shading normal, the implementation then flips either the shading normal or the true geometric normal if needed so that the two normals lie in the hemisphere. Since the shading normal generally represents a relatively small perturbation of the geometric normal, the two of them should always be in the same hemisphere. Depending on the context, either the geometric normal or the shading normal may more authoritatively point toward the correct “outside” of the surface, so the caller passes a Boolean value that determines which should be flipped if needed.

```
Faceforward() 72
Interaction::: 116
Normal3f 71
Shape:::reverseOrientation 124
Shape:::transformSwapsHandedness 124
SurfaceInteraction:::shading:::dndu 118
SurfaceInteraction:::shading:::dndv 118
SurfaceInteraction:::shading:::dpdu 118
SurfaceInteraction:::shading:::dpdv 118
SurfaceInteraction:::shading:::n 118
SurfaceInteraction:::shape 116
Transform 83
Vector3f 60
```

```
<Compute shading.n for SurfaceInteraction> ≡ 119
    shading.n = Normalize((Normal3f)Cross(dpdus, dpdvs));
    if (shape && (shape->reverseOrientation ^
                    shape->transformSwapsHandedness))
        shading.n = -shading.n;
    if (orientationIsAuthoritative)
        n = Faceforward(n, shading.n);
    else
        shading.n = Faceforward(shading.n, n);
```

```
<Initialize shading partial derivative values> ≡ 119
    shading.dpdus = dpdus;
    shading.dpdvs = dpdvs;
    shading.dndus = dndus;
    shading.dndvs = dndvs;
```

We’ll add a method to `Transform` to transform `SurfaceInteractions`. Most members are either transformed directly or copied, as appropriate, but given the approach that `pbrt` uses for bounding floating-point error in computed intersection points, transforming the `p` and `pError` member variables requires special care. The fragment that handles this, *<Transform p and pError in SurfaceInteraction>* is defined in Section 3.9, when floating-point rounding error is discussed.

```
<Transform Method Definitions> +≡
  SurfaceInteraction
  Transform::operator()(const SurfaceInteraction &si) const {
    SurfaceInteraction ret;
    <Transform p and pError in SurfaceInteraction 229>
    <Transform remaining members of SurfaceInteraction>
    return ret;
}
```

## FURTHER READING

DeRose, Goldman, and their collaborators have argued for an elegant “coordinate-free” approach to describing vector geometry for graphics, where the fact that positions and directions happen to be represented by  $(x, y, z)$  coordinates with respect to a particular coordinate system is deemphasized and where points and vectors themselves record which coordinate system they are expressed in terms of (Goldman 1985; DeRose 1989; Mann, Litke, and DeRose 1997). This makes it possible for a software layer to ensure that common errors like adding a vector in one coordinate system to a point in another coordinate system are transparently handled by transforming them to a common coordinate system first. We have not followed this approach in *pbrt*, although the principles behind this approach are well worth understanding and keeping in mind when working with coordinate systems in computer graphics.

Schneider and Eberly’s *Geometric Tools for Computer Graphics* is influenced by the coordinate-free approach and covers the topics of this chapter in much greater depth (Schneider and Eberly 2003). It is also full of useful geometric algorithms for graphics. A classic and more traditional introduction to the topics of this chapter is *Mathematical Elements for Computer Graphics* by Rogers and Adams (1990). Note that their book uses a row-vector representation of points and vectors, however, which means that our matrices would be transposed when expressed in their framework, and that they multiply points and vectors by matrices to transform them ( $\mathbf{PM}$ ), rather than multiplying matrices by points as we do ( $\mathbf{Mp}$ ). Homogeneous coordinates were only briefly mentioned in this chapter, although they are the basis of projective geometry, where they are the foundation of many elegant algorithms. Stolfi’s book is an excellent introduction to this topic (Stolfi 1991).

There are many good books on linear algebra and vector geometry. We have found Lang (1986) and Buck (1978) to be good references on these respective topics. See also Akenine-Möller et al.’s *Real-Time Rendering* book (2008) for a solid graphics-based introduction to linear algebra.

The subtleties of how normal vectors are transformed were first widely understood in the graphics community after articles by Wallis (1990) and Turkowski (1990b).

Shoemake (1985) introduced quaternions to graphics and showed their utility for animating rotations. Using polar matrix decomposition for animating transformations was described by Shoemake and Duff (1992); Higham (1986) developed the algorithm for extracting the rotation from a composite rotation and scale matrix by successively adding the matrix to its inverse transpose. Shoemake’s chapters in *Graphics Gems* (1991, 1994,

SurfaceInteraction 116

Transform 83

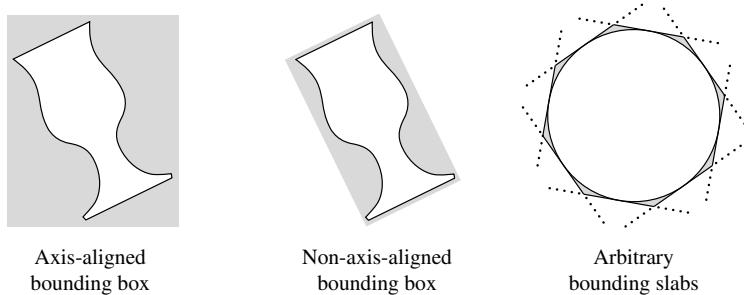
1994) respectively give more details on the derivation of the conversion from matrices to quaternions and the implementation of polar matrix decomposition.

We followed Blow’s derivation of spherical linear interpolation (2004) in our exposition in this chapter. Bloom et al. (2004) discuss desirable properties of interpolation of rotations for animation in computer graphics and which approaches deliver which of these properties. See Eberly (2011) for a more efficient implementation of a `Slerp()` function, based on approximating the trigonometric functions involved with polynomials. For more sophisticated approaches to rotation interpolation, see Ramamoorthi and Barr (1997) and Buss and Fillmore (2001). A technique to efficiently compute a matrix to rotate one vector to another was presented by Akenine-Möller and Hughes (1999).

Interval arithmetic is a tool that’s often useful in rendering; see Moore’s book (1966) for a well-written introduction.

## EXERCISES

- ① 2.1** Find a more efficient way to transform axis-aligned bounding boxes by taking advantage of the symmetries of the problem: because the eight corner points are linear combinations of three axis-aligned basis vectors and a single corner point, their transformed bounding box can be found much more efficiently than by the method we presented (Arvo 1990).
- ② 2.2** Instead of boxes, tighter bounds around objects could be computed by using the intersections of many nonorthogonal slabs. Extend the bounding box representation in `pbrt` to allow the user to specify a bound comprised of arbitrary slabs.



- ① 2.3** Change `pbrt` so that it transforms `Normal3f`s just like `Vector3f`s, and create a scene that gives a clearly incorrect image due to this bug. (Don’t forget to eliminate this change from your copy of the source code when you’re done!)
- ② 2.4** If only the translation components of a transformation are time varying, for example, then the `AnimatedTransform` implementation does unnecessary computation in interpolating between two rotations that are the same. Modify the `AnimatedTransform` implementation so that it avoids this work in cases where the full generality of its current implementation isn’t necessary. How much of a performance difference do you observe for scenes where your optimizations are applicable?

`AnimatedTransform` 103

`Normal3f` 71

`Slerp()` 103

`Vector3f` 60

# CHAPTER THREE

# 03 SHAPES

In this chapter, we will present pbrt’s abstraction for geometric primitives such as spheres and triangles. Careful abstraction of geometric shapes in a ray tracer is a key component of a clean system design, and shapes are the ideal candidate for an object-oriented approach. All geometric primitives implement a common interface, and the rest of the renderer can use this interface without needing any details about the underlying shape. This makes it possible to separate the geometric and shading subsystems of pbrt.

pbrt hides details about primitives behind a two-level abstraction. The `Shape` class provides access to the raw geometric properties of the primitive, such as its surface area and bounding box, and provides a ray intersection routine. The `Primitive` class encapsulates additional nongeometric information about the primitive, such as its material properties. The rest of the renderer then deals only with the abstract `Primitive` interface. This chapter will focus on the geometry-only `Shape` class; the `Primitive` interface is a key topic of Chapter 4.

## 3.1 BASIC SHAPE INTERFACE

The interface for Shapes is defined in the source file `core/shape.h`, and definitions of common `Shape` methods can be found in `core/shape.cpp`. The `Shape` base class defines the general `Shape` interface. It also exposes a few public data members that are useful for all `Shape` implementations.

```
{Shape Declarations} ≡  
class Shape {  
public:  
    {Shape Interface 125}  
    {Shape Public Data 124}  
};
```

All shapes are defined in object coordinate space; for example, all spheres are defined in a coordinate system where the center of the sphere is at the origin. In order to place a sphere at another position in the scene, a transformation that describes the mapping from object space to world space must be provided. The Shape class stores both this transformation and its inverse.

Shapes also take a Boolean parameter, `reverseOrientation`, that indicates whether their surface normal directions should be reversed from the default. This capability is useful because the orientation of the surface normal is used to determine which side of a shape is “outside.” For example, shapes that emit illumination are emissive only on the side the surface normal lies on. The value of this parameter is managed via the `ReverseOrientation` statement in pbrt input files.

Shapes also store the return value of the `Transform::SwapsHandedness()` call for their object-to-world transformation. This value is needed by the `SurfaceInteraction` constructor that is called each time a ray intersection is found, so the `Shape` constructor computes it once and stores it.

*(Shape Method Definitions) ≡*

```
Shape::Shape(const Transform *ObjectToWorld,
            const Transform *WorldToObject, bool reverseOrientation)
: ObjectToWorld(ObjectToWorld), WorldToObject(WorldToObject),
  reverseOrientation(reverseOrientation),
  transformSwapsHandedness(ObjectToWorld->SwapsHandedness()) {
}
```

An important detail is that shapes store pointers to their transformations rather than `Transform` objects directly. Recall from Section 2.7 that `Transform` objects are represented by a total of 32 floats, requiring 128 bytes of memory; because multiple shapes in the scene will frequently have the same transformation applied to them, pbrt keeps a pool of `Transform`s so that they can be re-used and passes pointers to the shared `Transform`s to the shapes. As such, the `Shape` destructor does not delete its `Transform` pointers, leaving the `Transform` management code to manage that memory instead.

*(Shape Public Data) ≡*

123

```
const Transform *ObjectToWorld, *WorldToObject;
const bool reverseOrientation;
const bool transformSwapsHandedness;
```

### 3.1.1 BOUNDING

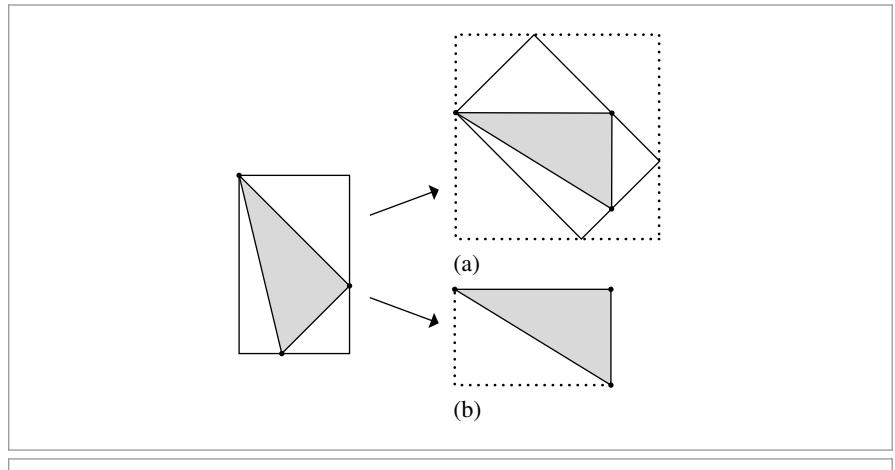
The scenes that pbrt will render will often contain objects that are computationally expensive to process. For many operations, it is often useful to have a 3D *bounding volume* that encloses an object. For example, if a ray does not pass through a particular bounding volume, pbrt can avoid processing all of the objects inside of it for that ray.

Axis-aligned bounding boxes are a convenient bounding volume, as they require only six floating-point values to store and fit many shapes well. Furthermore, it’s fairly inexpensive to test for the intersection of a ray with an axis-aligned bounding box. Each `Shape` implementation must therefore be capable of bounding itself with an axis-aligned

Shape 123

Transform 83

Transform::SwapsHandedness() 96



**Figure 3.1:** (a) A world space bounding box of a triangle is computed by transforming its object space bounding box to world space and then finding the bounding box that encloses the resulting bounding box; a sloppy bound may result. (b) However, if the triangle's vertices are first transformed from object space to world space and then bounded, the fit of the bounding box can be much better.

bounding box represented by a `Bounds3f`. There are two different bounding methods. The first, `ObjectBound()`, returns a bounding box in the shape's object space.

*(Shape Interface)*  $\equiv$

```
virtual Bounds3f ObjectBound() const = 0;
```

123

The second bounding method, `WorldBound()`, returns a bounding box in world space. `pbrt` provides a default implementation of this method that transforms the object space bound to world space. Shapes that can easily compute a tighter world space bound should override this method, however. An example of such a shape is a triangle (Figure 3.1).

*(Shape Method Definitions)*  $+ \equiv$

```
Bounds3f Shape::WorldBound() const {
    return (*ObjectToWorld)(ObjectBound());
}
```

123

### 3.1.2 RAY-BOUNDS INTERSECTIONS

Given the use of `Bounds3f` instances to bound shapes, we will add a `Bounds3` method, `Bounds3::IntersectP()`, that checks for a ray–box intersection and returns the two parametric  $t$  values of the intersection, if any.

One way to think of bounding boxes is as the intersection of three slabs, where a slab is the region of space between two parallel planes. To intersect a ray against a box, we intersect the ray against each of the box's three slabs in turn. Because the slabs are aligned with the three coordinate axes, a number of optimizations can be made in the ray–slab tests.

The basic ray–bounding box intersection algorithm works as follows: we start with a parametric interval that covers that range of positions  $t$  along the ray where we're interested in finding intersections; typically, this is  $(0, \infty)$ . We will then successively compute the two

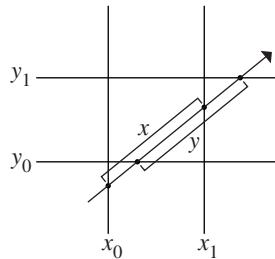
`Bounds3` 76

`Bounds3::IntersectP()` 127

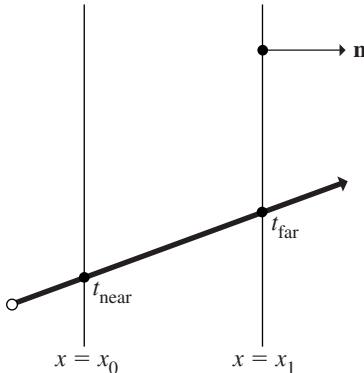
`Bounds3f` 76

`Shape::ObjectBound()` 125

`Shape::ObjectToWorld` 124



**Figure 3.2: Intersecting a Ray with an Axis-Aligned Bounding Box.** We compute intersection points with each slab in turn, progressively narrowing the parametric interval. Here, in 2D, the intersection of the  $x$  and  $y$  extents along the ray gives the extent where the ray is inside the box.



**Figure 3.3: Intersecting a Ray with an Axis-Aligned Slab.** The two planes shown here are described by  $x = c$  for constant values  $c$ . The normal of each plane is  $(1, 0, 0)$ . Unless the ray is parallel to the planes, it will intersect the slab twice, at parametric positions  $t_{\text{near}}$  and  $t_{\text{far}}$ .

parametric  $t$  positions where the ray intersects each axis-aligned slab. We compute the set intersection of the per-slab intersection interval with the current intersection interval, returning failure if we find that the resulting interval is degenerate. If, after checking all three slabs, the interval is nondegenerate, we have the parametric range of the ray that is inside the box. Figure 3.2 illustrates this process, and Figure 3.3 shows the basic geometry of a ray and a slab.

If the `Bounds3::IntersectP()` method returns `true`, the intersection's parametric range is returned in the optional arguments `hitt0` and `hitt1`. Intersections outside of the `(0, Ray::tMax)` range of the ray are ignored. If the ray's origin is inside the box, 0 is returned for `hitt0`.

`Bounds3::IntersectP()` 127

```
(Geometry Inline Functions) +≡
template <typename T>
inline bool Bounds3<T>::IntersectP(const Ray &ray, Float *hitt0,
    Float *hitt1) const {
    Float t0 = 0, t1 = ray.tMax;
    for (int i = 0; i < 3; ++i) {
        {Update interval for i-th bounding box slab 128}
    }
    if (*hitt0) *hitt0 = t0;
    if (*hitt1) *hitt1 = t1;
    return true;
}
```

For each pair of planes, this routine needs to compute two ray–plane intersections. For example, the slab described by two planes perpendicular to the  $x$  axis can be described by planes through points  $(x_1, 0, 0)$  and  $(x_2, 0, 0)$ , each with normal  $(1, 0, 0)$ . Consider the first  $t$  value for a plane intersection,  $t_1$ . The parametric  $t$  value for the intersection between a ray with origin  $\mathbf{o}$  and direction  $\mathbf{d}$  and a plane  $ax + by + cz + d = 0$  can be found by substituting the ray equation into the plane equation:

$$\begin{aligned} 0 &= a(\mathbf{o}_x + t\mathbf{d}_x) + b(\mathbf{o}_y + t\mathbf{d}_y) + c(\mathbf{o}_z + t\mathbf{d}_z) + d \\ &= (\mathbf{a}, \mathbf{b}, \mathbf{c}) \cdot \mathbf{o} + t(\mathbf{a}, \mathbf{b}, \mathbf{c}) \cdot \mathbf{d} + d. \end{aligned}$$

Solving for  $t$  gives

$$t = \frac{-d - ((\mathbf{a}, \mathbf{b}, \mathbf{c}) \cdot \mathbf{o})}{((\mathbf{a}, \mathbf{b}, \mathbf{c}) \cdot \mathbf{d})}.$$

Because the  $y$  and  $z$  components of the plane’s normal are zero,  $b$  and  $c$  are zero, and  $a$  is one. The plane’s  $d$  coefficient is  $-x_1$ . We can use this information and the definition of the dot product to simplify the calculation substantially:

$$t_1 = \frac{x_1 - \mathbf{o}_x}{\mathbf{d}_x}.$$

The code to compute the  $t$  values of the slab intersections starts by computing the reciprocal of the corresponding component of the ray direction so that it can multiply by this factor instead of performing multiple divisions. Note that, although it divides by this component, it is not necessary to verify that it is nonzero. If it is zero, then `invRayDir` will hold an infinite value, either  $-\infty$  or  $\infty$ , and the rest of the algorithm still works correctly.<sup>1</sup>

---

Float 1062

Ray 73

Ray::tMax 73

<sup>1</sup> This assumes that the architecture being used supports IEEE floating-point arithmetic (Institute of Electrical and Electronic Engineers 1985), which is universal on modern systems. The relevant properties of IEEE floating-point arithmetic are that for all  $v > 0$ ,  $v/0 = \infty$  and for all  $w < 0$ ,  $w/0 = -\infty$ , where  $\infty$  is a special value such that any positive number multiplied by  $\infty$  gives  $\infty$  and any negative number multiplied by  $\infty$  gives  $-\infty$ , and so on. See Section 3.9.1 for more information about floating-point arithmetic.

```
(Update interval for i'th bounding box slab) ≡
  Float invRayDir = 1 / ray.d[i];
  Float tNear = (pMin[i] - ray.o[i]) * invRayDir;
  Float tFar = (pMax[i] - ray.o[i]) * invRayDir;
  (Update parametric interval from slab intersection t values 128)
```

127

The two distances are reordered so that `tNear` holds the closer intersection and `tFar` the farther one. This gives a parametric range [`tNear`, `tFar`], which is used to compute the set intersection with the current range [`t0`, `t1`] to compute a new range. If this new range is empty (i.e., `t0 > t1`), then the code can immediately return failure.

There is another floating-point-related subtlety here: in the case where the ray origin is in the plane of one of the bounding box slabs and the ray lies in the plane of the slab, it is possible that `tNear` or `tFar` will be computed by an expression of the form 0/0, which results in an IEEE floating-point “not a number” (NaN) value. Like infinity values, NaNs have well-specified semantics: for example, any logical comparison involving a NaN always evaluates to false. Therefore, the code that updates the values of `t0` and `t1` is carefully written so that if `tNear` or `tFar` is NaN, then `t0` or `t1` won’t ever take on a NaN value but will always remain unchanged.

```
(Update parametric interval from slab intersection t values) ≡
  if (tNear > tFar) std::swap(tNear, tFar);
  (Update tFar to ensure robust ray–bounds intersection 221)
  t0 = tNear > t0 ? tNear : t0;
  t1 = tFar < t1 ? tFar : t1;
  if (t0 > t1) return false;
```

128

`Bounds3` also provides a specialized `IntersectP()` method that takes the reciprocal of the ray’s direction as an additional parameter, so that the three reciprocals don’t need to be computed each time `IntersectP()` is called.

This version also takes precomputed values that indicate whether each direction component is negative, which makes it possible to eliminate the comparisons of the computed `tNear` and `tFar` values in the original routine and just directly compute the respective near and far values. Because the comparisons that order these values from low to high in the original code are dependent on computed values, they can be inefficient for processors to execute, since the computation of their values must be completely finished before the comparison can be made.

This routine returns `true` if the ray segment is entirely inside the bounding box, even if the intersections are not within the ray’s (0, `tMax`) range.

```
(Geometry Inline Functions) +≡
  template <typename T>
  inline bool Bounds3<T>::IntersectP(const Ray &ray, const Vector3f &invDir,
    const int dirIsNeg[3]) const {
```

Bounds3 76

Bounds3::pMax 77

Bounds3::pMin 77

Bounds3f 76

Float 1062

Ray 73

Ray::d 73

Ray::o 73

Ray::tMax 73

Vector3f 60

```

    const Bounds3f &bounds = *this;
    ⟨Check for ray intersection against x and y slabs 129⟩
    ⟨Check for ray intersection against z slab⟩
    return (tMin < ray.tMax) && (tMax > 0);
}

```

If the ray direction vector is negative, the “near” parametric intersection will be found with the slab with the larger of the two bounding values, and the far intersection will be found with the slab with the smaller of them. The implementation can use this observation to compute the near and far parametric values in each direction directly.

```

⟨Check for ray intersection against x and y slabs⟩ ≡ 128
    Float tMin = (bounds[ dirIsNeg[0]].x - ray.o.x) * invDir.x;
    Float tMax = (bounds[1-dirIsNeg[0]].x - ray.o.x) * invDir.x;
    Float tyMin = (bounds[ dirIsNeg[1]].y - ray.o.y) * invDir.y;
    Float tyMax = (bounds[1-dirIsNeg[1]].y - ray.o.y) * invDir.y;
    ⟨Update tMax and tyMax to ensure robust bounds intersection⟩
    if (tMin > tyMax || tyMin > tMax)
        return false;
    if (tyMin > tMin) tMin = tyMin;
    if (tyMax < tMax) tMax = tyMax;

```

The fragment *⟨Check for ray intersection against z slab⟩* is analogous and isn’t included here.

This intersection test is at the heart of traversing the BVHAccel acceleration structure, which is introduced in Section 4.3. Because so many ray–bounding box intersection tests are performed while traversing the BVH tree, we found that this optimized method provided approximately a 15% performance improvement in overall rendering time compared to using the `Bounds3::IntersectP()` variant that didn’t take the precomputed direction reciprocals and signs.

### 3.1.3 INTERSECTION TESTS

Shape implementations must provide an implementation of one (and possibly two) methods that test for ray intersections with their shape. The first, `Shape::Intersect()`, returns geometric information about a single ray–shape intersection corresponding to the first intersection, if any, in the  $(0, \text{tMax})$  parametric range along the ray.

`Bounds3::IntersectP()` 127  
`BVHAccel` 256  
`Float` 1062  
`Ray` 73  
`Shape` 123  
`Shape::Intersect()` 129  
`SurfaceInteraction` 116

```

⟨Shape Interface⟩ +≡ 123
    virtual bool Intersect(const Ray &ray, Float *tHit,
                          SurfaceInteraction *isect, bool testAlphaTexture = true) const = 0;

```

There are a few important things to keep in mind when reading (and writing) intersection routines:

- The Ray structure contains a Ray::tMax member that defines the endpoint of the ray. Intersection routines must ignore any intersections that occur after this point.
- If an intersection is found, its parametric distance along the ray should be stored in the tHit pointer that is passed into the intersection routine. If there are multiple intersections along the ray, the closest one should be reported.
- Information about an intersection is stored in the SurfaceInteraction structure, which completely captures the local geometric properties of a surface. This class is used heavily throughout pbrt, and it serves to cleanly isolate the geometric portion of the ray tracer from the shading and illumination portions. The Surface Interaction class was defined in Section 2.10.<sup>2</sup>
- The rays passed into intersection routines are in world space, so shapes are responsible for transforming them to object space if needed for intersection tests. The intersection information returned should be in world space.

Some shape implementations support cutting away some of their surfaces using a texture; the testAlphaTexture parameter indicates whether those that do should perform this operation for the current intersection test.

The second intersection test method, Shape::IntersectP(), is a predicate function that determines whether or not an intersection occurs, without returning any details about the intersection itself. The Shape class provides a default implementation of the IntersectP() method that calls the Shape::Intersect() method and just ignores the additional information computed about intersection points. As this can be fairly wasteful, almost all shape implementations in pbrt provide a more efficient implementation for IntersectP() that determines whether an intersection exists without computing all of its details.

```
<Shape Interface> +≡
    virtual bool IntersectP(const Ray &ray,
                           bool testAlphaTexture = true) const {
        Float tHit = ray.tMax;
        SurfaceInteraction isect;
        return Intersect(ray, &tHit, &isect, testAlphaTexture);
    }
```

123

### 3.1.4 SURFACE AREA

In order to properly use Shapes as area lights, it is necessary to be able to compute the surface area of a shape in object space.

---

<sup>2</sup> Almost all ray tracers use this general idiom for returning geometric information about intersections with shapes. As an optimization, many will only partially initialize the intersection information when an intersection is found, storing just enough information so that the rest of the values can be computed later if actually needed. This approach saves work in the case where a closer intersection is later found with another shape. In our experience, the extra work to compute all the information isn't substantial, and for renderers that have complex scene data management algorithms (e.g., discarding geometry from main memory when too much memory is being used and writing it to disk), the deferred approach may fail because the shape is no longer in memory.

Float 1062

Ray 73

Ray::tMax 73

Shape::Intersect() 129

Shape::IntersectP() 130

SurfaceInteraction 116

```
<Shape Interface> +≡
    virtual Float Area() const = 0;
```

123

### 3.1.5 SIDEDNESS

Many rendering systems, particularly those based on scan line or  $z$ -buffer algorithms, support the concept of shapes being “one-sided”—the shape is visible if seen from the front but disappears when viewed from behind. In particular, if a geometric object is closed and always viewed from the outside, then the back-facing parts of it can be discarded without changing the resulting image. This optimization can substantially improve the speed of these types of hidden surface removal algorithms. The potential for improved performance is reduced when using this technique with ray tracing, however, since it is often necessary to perform the ray–object intersection before determining the surface normal to do the back-facing test. Furthermore, this feature can lead to a physically inconsistent scene description if one-sided objects are not in fact closed. For example, a surface might block light when a shadow ray is traced from a light source to a point on another surface, but not if the shadow ray is traced in the other direction. For all of these reasons, pbrt doesn’t support this feature.

## 3.2 SPHERES

Spheres are a special case of a general type of surface called *quadrics*—surfaces described by quadratic polynomials in  $x$ ,  $y$ , and  $z$ . They are the simplest type of curved surface that is useful to a ray tracer and are a good starting point for general ray intersection routines. pbrt supports six types of quadrics: spheres, cones, disks (a special case of a cone), cylinders, hyperboloids, and paraboloids.

Many surfaces can be described in one of two main ways: in *implicit form* and in *parametric form*. An implicit function describes a 3D surface as

$$f(x, y, z) = 0.$$

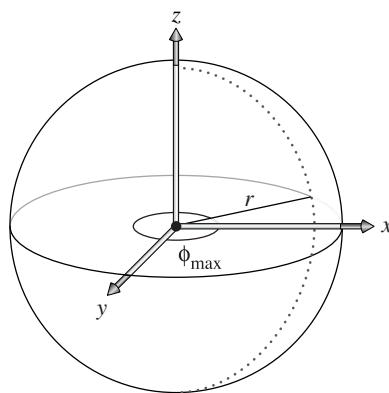
The set of all points  $(x, y, z)$  that fulfill this condition defines the surface. For a unit sphere at the origin, the familiar implicit equation is  $x^2 + y^2 + z^2 - 1 = 0$ . Only the set of points one unit from the origin satisfies this constraint, giving the unit sphere’s surface.

Many surfaces can also be described parametrically using a function to map 2D points to 3D points on the surface. For example, a sphere of radius  $r$  can be described as a function of 2D spherical coordinates  $(\theta, \phi)$ , where  $\theta$  ranges from 0 to  $\pi$  and  $\phi$  ranges from 0 to  $2\pi$  (Figure 3.4):

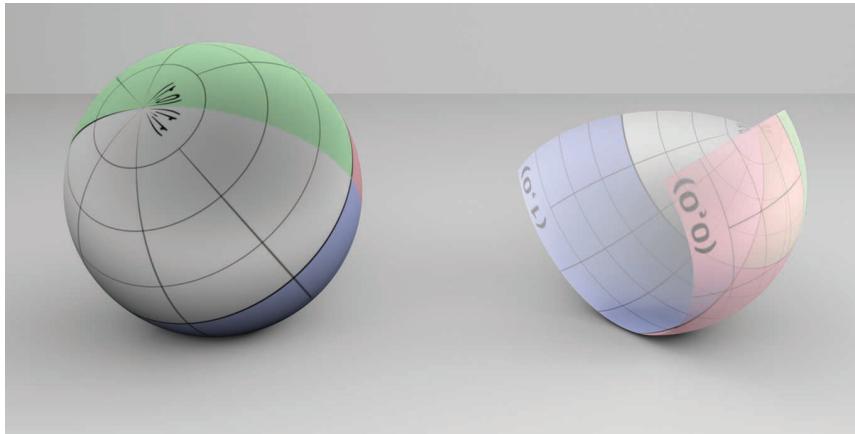
$$\begin{aligned} x &= r \sin \theta \cos \phi \\ y &= r \sin \theta \sin \phi \\ z &= r \cos \theta. \end{aligned}$$

We can transform this function  $f(\theta, \phi)$  into a function  $f(u, v)$  over  $[0, 1]^2$  and also generalize it slightly to allow partial spheres that only sweep out  $\theta \in [\theta_{\min}, \theta_{\max}]$  and  $\phi \in [0, \phi_{\max}]$  with the substitution

$$\begin{aligned} \phi &= u \phi_{\max} \\ \theta &= \theta_{\min} + v(\theta_{\max} - \theta_{\min}). \end{aligned}$$



**Figure 3.4: Basic Setting for the Sphere Shape.** It has a radius of  $r$  and is centered at the object space origin. A partial sphere may be described by specifying a maximum  $\phi$  value.



**Figure 3.5: Two Spheres.** On the left is a complete sphere, and on the right is a partial sphere ( $z_{\max} < r$  and  $\phi_{\max} < 2\pi$ ). Note that the texture map used shows the  $(u, v)$  parameterization of the shape; the singularity at one of the poles is visible in the complete sphere.

This form is particularly useful for texture mapping, where it can be directly used to map a texture defined over  $[0, 1]^2$  to the sphere. Figure 3.5 shows an image of two spheres; a grid image map has been used to show the  $(u, v)$  parameterization.

As we describe the implementation of the sphere shape, we will make use of both the implicit and parametric descriptions of the shape, depending on which is a more natural way to approach the particular problem we're facing.

The `Sphere` class represents a sphere that is centered at the origin in object space. Its implementation is in the files `shapes/sphere.h` and `shapes/sphere.cpp`.

```
(Sphere Declarations) ≡
class Sphere : public Shape {
public:
    (Sphere Public Methods 133)
private:
    (Sphere Private Data 133)
};
```

To place a sphere elsewhere in the scene, the user must apply an appropriate transformation when specifying the sphere in the input file. It takes both the object-to-world and world-to-object transformations as parameters to the constructor, passing them along to the parent `Shape` constructor.

The radius of the sphere can have an arbitrary positive value, and the sphere's extent can be truncated in two different ways. First, minimum and maximum  $z$  values may be set; the parts of the sphere below and above these planes, respectively, are cut off. Second, considering the parameterization of the sphere in spherical coordinates, a maximum  $\phi$  value can be set. The sphere sweeps out  $\phi$  values from 0 to the given  $\phi_{\max}$  such that the section of the sphere with spherical  $\phi$  values above  $\phi_{\max}$  is also removed.

```
(Sphere Public Methods) ≡ 133
Sphere(const Transform *ObjectToWorld, const Transform *WorldToObject,
       bool reverseOrientation, Float radius, Float zMin, Float zMax,
       Float phiMax)
: Shape(ObjectToWorld, WorldToObject, reverseOrientation),
  radius(radius), zMin(Clamp(std::min(zMin, zMax), -radius, radius)),
  zMax(Clamp(std::max(zMin, zMax), -radius, radius)),
  thetaMin(std::acos(Clamp(zMin / radius, -1, 1))),
  thetaMax(std::acos(Clamp(zMax / radius, -1, 1))),
  phiMax(Radians(Clamp(phiMax, 0, 360))) { }
```

`Clamp()` 1062  
`Float` 1062  
`Radians()` 1063  
`Shape` 123  
`Sphere` 133  
`Sphere::phiMax` 133  
`Sphere::radius` 133  
`Sphere::thetaMax` 133  
`Sphere::thetaMin` 133  
`Sphere::zMax` 133  
`Sphere::zMin` 133  
`Transform` 83

```
(Sphere Private Data) ≡ 133
const Float radius;
const Float zMin, zMax;
const Float thetaMin, thetaMax, phiMax;
```

### 3.2.1 BOUNDING

Computing an object space bounding box for a sphere is straightforward. The implementation here uses the values of  $z_{\min}$  and  $z_{\max}$  provided by the user to tighten up the bound when less than an entire sphere is being rendered. However, it doesn't do the extra work to compute a tighter bounding box when  $\phi_{\max}$  is less than  $3\pi/2$ . This improvement is left as an exercise.

```
(Sphere Method Definitions) ≡
    Bounds3f Sphere::ObjectBound() const {
        return Bounds3f(Point3f(-radius, -radius, zMin),
                        Point3f( radius,  radius, zMax));
    }
```

### 3.2.2 INTERSECTION TESTS

The task of deriving a ray–sphere intersection test is simplified by the fact that the sphere is centered at the origin. However, if the sphere has been transformed to another position in world space, then it is necessary to transform rays to object space before intersecting them with the sphere, using the world-to-object transformation. Given a ray in object space, the intersection computation can be performed in object space instead.<sup>3</sup>

The following fragment shows the entire intersection method:

```
(Sphere Method Definitions) +≡
bool Sphere::Intersect(const Ray &r, Float *tHit,
                      SurfaceInteraction *isect, bool testAlphaTexture) const {
    Float phi;
    Point3f pHit;
    <Transform Ray to object space 134>
    <Compute quadratic sphere coefficients 135>
    <Solve quadratic equation for t values 136>
    <Compute sphere hit position and φ 137>
    <Test sphere intersection against clipping parameters 137>
    <Find parametric representation of sphere hit 137>
    <Compute error bounds for sphere intersection 225>
    <Initialize SurfaceInteraction from parametric information 140>
    <Update tHit for quadric intersection 140>
    return true;
}
```

First, the given world space ray is transformed to the sphere’s object space. The remainder of the intersection test will take place in that coordinate system. The `oErr` and `dErr` variables respectively bound the floating-point round-off error in the transformed ray’s origin and direction that was introduced by applying the transformation. (See Section 3.9 for more information about floating-point arithmetic and its implications for accurate ray intersection calculations.)

*<Transform Ray to object space> ≡*

**134, 141, 144, 148, 173**

```
Vector3f oErr, dErr;
Ray ray = (*WorldToObject)(r, &oErr, &dErr);
```

Bounds3f 76  
 Float 1062  
 Point3f 68  
 Ray 73  
 Shape::WorldToObject 124  
 Sphere 133  
 Sphere::radius 133  
 Sphere::zMax 133  
 Sphere::zMin 133  
 SurfaceInteraction 116  
 Vector3f 60

---

<sup>3</sup> This is something of a classic theme in computer graphics. By transforming the problem to a particular restricted case, it is possible to more easily and efficiently do an intersection test: that is, many terms of the equations cancel out since the sphere is always at (0, 0, 0). No overall generality is lost, since an appropriate translation can be applied to the ray for spheres at other positions.

If a sphere is centered at the origin with radius  $r$ , its implicit representation is

$$x^2 + y^2 + z^2 - r^2 = 0.$$

By substituting the parametric representation of the ray from Equation (2.3) into the implicit sphere equation, we have

$$(o_x + t \mathbf{d}_x)^2 + (o_y + t \mathbf{d}_y)^2 + (o_z + t \mathbf{d}_z)^2 = r^2.$$

Note that all elements of this equation besides  $t$  are known values. The  $t$  values where the equation holds give the parametric positions along the ray where the implicit sphere equation holds and thus the points along the ray where it intersects the sphere. We can expand this equation and gather the coefficients for a general quadratic equation in  $t$ ,

$$at^2 + bt + c = 0,$$

where<sup>4</sup>

$$\begin{aligned} a &= \mathbf{d}_x^2 + \mathbf{d}_y^2 + \mathbf{d}_z^2 \\ b &= 2(\mathbf{d}_x o_x + \mathbf{d}_y o_y + \mathbf{d}_z o_z) \\ c &= o_x^2 + o_y^2 + o_z^2 - r^2. \end{aligned}$$

This result directly translates to this fragment of source code. Note that in this code, instances of the `EFloat` class, not `Floats`, are used to represent floating-point values. `EFloat` tracks accumulated floating-point rounding error; its use is discussed in Section 3.9. For now, it can just be read as being equivalent to `Float`.

```
(Compute quadratic sphere coefficients) ≡ 134, 141
(Initialize EFloat ray coordinate values 135)
EFloat a = dx * dx + dy * dy + dz * dz;
EFloat b = 2 * (dx * ox + dy * oy + dz * oz);
EFloat c = ox * ox + oy * oy + oz * oz - EFloat(radius) * EFloat(radius);
```

The ray origin and direction values used in the intersection test are initialized with the floating-point error bounds from transforming the ray to object space.

```
(Initialize EFloat ray coordinate values) ≡ 135, 144
EFloat ox(ray.o.x, oErr.x), oy(ray.o.y, oErr.y), oz(ray.o.z, oErr.z);
EFloat dx(ray.d.x, dErr.x), dy(ray.d.y, dErr.y), dz(ray.d.z, dErr.z);
```

There are two possible solutions to the quadratic equation, giving zero, one, or two nonimaginary  $t$  values where the ray intersects the sphere.

`EFloat` 218

`Sphere::radius` 133

<sup>4</sup> Some ray tracers require that the direction vector of a ray be normalized, meaning  $a = 1$ . This can lead to subtle errors, however, if the caller forgets to normalize the ray direction. Of course, these errors can be avoided by normalizing the direction in the ray constructor, but this wastes effort when the provided direction is *already* normalized. To avoid this needless complexity, `pbrt` never insists on vector normalization in intersection routines. This is particularly helpful since it reduces the amount of computation needed to transform rays to object space, because no normalization is necessary there.

*(Solve quadratic equation for t values) ≡*

134, 141, 144

```
EFloat t0, t1;
if (!Quadratic(a, b, c, &t0, &t1))
    return false;
(Check quadric shape t0 and t1 for nearest intersection 136)
```

The `Quadratic()` utility function solves a quadratic equation, returning `false` if there are no real solutions and returning `true` and setting `t0` and `t1` appropriately if there are solutions. It is defined later in Section 3.9.4, where we discuss how to implement it robustly using floating-point arithmetic.

The computed parametric distances `t0` and `t1` track uncertainty due to errors in the original ray parameters and errors accrued in `Quadratic()`; the lower and upper range of the uncertainty interval can be queried using the methods `EFloat::LowerBound()` and `EFloat::UpperBound()`.

The fragment *(Check quadric shape t0 and t1 for nearest intersection)* takes the two intersection *t* values and determines which, if any, is the closest valid intersection. For an intersection to be valid, its *t* value must be greater than zero and less than `ray.tMax`. The following code uses the error intervals provided by the `EFloat` class and only accepts intersections that are unequivocally in the range (0, `tMax`).

Since  $t_0$  is guaranteed to be less than or equal to  $t_1$  (and 0 is less than `tMax`), then if  $t_0$  is greater than `tMax` or  $t_1$  is less than 0, it is certain that both intersections are out of the range of interest. Otherwise,  $t_0$  is the tentative hit *t* value. It may be less than 0, however, in which case we ignore it and try  $t_1$ . If that is also out of range, we have no valid intersection. If there is an intersection, then `tShapeHit` is initialized to hold the parametric *t* value for the intersection.

*(Check quadric shape t0 and t1 for nearest intersection) ≡*

136

```
if (t0.UpperBound() > ray.tMax || t1.LowerBound() <= 0)
    return false;
EFloat tShapeHit = t0;
if (tShapeHit.LowerBound() <= 0) {
    tShapeHit = t1;
    if (tShapeHit.UpperBound() > ray.tMax)
        return false;
}
```

Given the parametric distance along the ray to the intersection with a full sphere, the intersection point `pHit` can be computed as that offset along the ray.

It is next necessary to handle partial spheres with clipped *z* or  $\phi$  ranges—intersections that are in clipped areas must be ignored. The implementation starts by computing the  $\phi$  value for the hit point. Using the parametric representation of the sphere,

$$\frac{y}{x} = \frac{r \sin \theta \sin \phi}{r \sin \theta \cos \phi} = \tan \phi,$$

so  $\phi = \arctan y/x$ . It is necessary to remap the result of the standard library's `std::atan()` function to a value between 0 and  $2\pi$ , to match the sphere's original definition.

EFloat 218  
`EFloat::LowerBound()` 220  
`EFloat::UpperBound()` 220  
`Quadratic()` 1079  
`Ray::tMax` 73

```

⟨Compute sphere hit position and  $\phi$ ⟩ ≡
    pHit = ray((Float)tShapeHit);
    ⟨Refine sphere intersection point 225⟩
    if (pHit.x == 0 && pHit.y == 0) pHit.x = 1e-5f * radius;
    phi = std::atan2(pHit.y, pHit.x);
    if (phi < 0) phi += 2 * Pi;

```

134, 137, 141

Due to floating-point precision limitations, this computed intersection point `pHit` may lie a bit to one side of the actual sphere surface; the `⟨Refine sphere intersection point⟩` fragment, which is defined in Section 3.9.4, improves the precision of this value.

The hit point can now be tested against the specified minima and maxima for  $z$  and  $\phi$ . One subtlety is that it's important to skip the  $z$  tests if the  $z$  range includes the entire sphere; the computed `pHit.z` value may be slightly out of the  $z$  range due to floating-point round-off, so we should only perform this test when the user expects the sphere to be partially incomplete. If the  $t_0$  intersection wasn't actually valid, the routine tries again with  $t_1$ .

```

⟨Test sphere intersection against clipping parameters⟩ ≡
    if ((zMin > -radius && pHit.z < zMin) ||
        (zMax < radius && pHit.z > zMax) || phi > phiMax) {
        if (tShapeHit == t1) return false;
        if (t1.UpperBound() > ray.tMax) return false;
        tShapeHit = t1;
        ⟨Compute sphere hit position and  $\phi$  137⟩
        if ((zMin > -radius && pHit.z < zMin) ||
            (zMax < radius && pHit.z > zMax) || phi > phiMax)
            return false;
    }

```

134, 141

At this point in the routine, it is certain that the ray hits the sphere. The method next computes  $u$  and  $v$  values by scaling the previously computed  $\phi$  value for the hit to lie between 0 and 1 and by computing a  $\theta$  value between 0 and 1 for the hit point, based on the range of  $\theta$  values for the given sphere. Then it finds the parametric partial derivatives of position  $\partial p/\partial u$  and  $\partial p/\partial v$  and surface normal  $\partial n/\partial u$  and  $\partial n/\partial v$ .

`Clamp()` 1062  
`EFloat::UpperBound()` 220  
`Float` 1062  
`Pi` 1063  
`Ray::tMax` 73  
`Sphere::phiMax` 133  
`Sphere::radius` 133  
`Sphere::thetaMax` 133  
`Sphere::thetaMin` 133  
`Sphere::zMax` 133  
`Sphere::zMin` 133

```

⟨Find parametric representation of sphere hit⟩ ≡
    Float u = phi / phiMax;
    Float theta = std::acos(Clamp(pHit.z / radius, -1, 1));
    Float v = (theta - thetaMin) / (thetaMax - thetaMin);
    ⟨Compute sphere  $\partial p/\partial u$  and  $\partial p/\partial v$  138⟩
    ⟨Compute sphere  $\partial n/\partial u$  and  $\partial n/\partial v$  139⟩

```

134

Computing the partial derivatives of a point on the sphere is a short exercise in algebra. Here we will show how the  $x$  component of  $\partial p/\partial u$ ,  $\partial p_x/\partial u$ , is calculated; the other components are found similarly. Using the parametric definition of the sphere, we have

$$\begin{aligned}
 x &= r \sin \theta \cos \phi \\
 \frac{\partial p_x}{\partial u} &= \frac{\partial}{\partial u} (r \sin \theta \cos \phi) \\
 &= r \sin \theta \frac{\partial}{\partial u} (\cos \phi) \\
 &= r \sin \theta (-\phi_{\max} \sin \phi).
 \end{aligned}$$

Using a substitution based on the parametric definition of the sphere's  $y$  coordinate, this simplifies to

$$\frac{\partial p_x}{\partial u} = -\phi_{\max} y.$$

Similarly,

$$\frac{\partial p_y}{\partial u} = \phi_{\max} x,$$

and

$$\frac{\partial p_z}{\partial u} = 0.$$

A similar process gives  $\partial p / \partial v$ . The complete result is

$$\begin{aligned}
 \frac{\partial p}{\partial u} &= (-\phi_{\max} y, \phi_{\max} x, 0) \\
 \frac{\partial p}{\partial v} &= (\theta_{\max} - \theta_{\min}) (z \cos \phi, z \sin \phi, -r \sin \theta).
 \end{aligned}$$

*(Compute sphere  $\partial p / \partial u$  and  $\partial p / \partial v$ )*

137

```

Float zRadius = std::sqrt(pHit.x * pHit.x + pHit.y * pHit.y);
Float invZRadius = 1 / zRadius;
Float cosPhi = pHit.x * invZRadius;
Float sinPhi = pHit.y * invZRadius;
Vector3f dpdu(-phiMax * pHit.y, phiMax * pHit.x, 0);
Vector3f dpdv = (thetaMax - thetaMin) *
    Vector3f(pHit.z * cosPhi, pHit.z * sinPhi,
              -radius * std::sin(theta));

```

### \* 3.2.3 PARTIAL DERIVATIVES OF NORMAL VECTORS

It is also useful to determine how the normal changes as we move along the surface in the  $u$  and  $v$  directions. For example, the antialiasing techniques in Chapter 10 are dependent on this information to antialias textures on objects that are seen reflected in curved surfaces. The differential changes in normal  $\partial \mathbf{n} / \partial u$  and  $\partial \mathbf{n} / \partial v$  are given by the *Weingarten equations* from differential geometry:

$$\begin{aligned}
 \frac{\partial \mathbf{n}}{\partial u} &= \frac{fF - eG}{EG - F^2} \frac{\partial \mathbf{p}}{\partial u} + \frac{eF - fE}{EG - F^2} \frac{\partial \mathbf{p}}{\partial v} \\
 \frac{\partial \mathbf{n}}{\partial v} &= \frac{gF - fG}{EG - F^2} \frac{\partial \mathbf{p}}{\partial u} + \frac{fF - gE}{EG - F^2} \frac{\partial \mathbf{p}}{\partial v},
 \end{aligned}$$

Float 1062  
 Sphere::phiMax 133  
 Sphere::radius 133  
 Sphere::thetaMax 133  
 Sphere::thetaMin 133  
 Vector3f 60

where  $E$ ,  $F$ , and  $G$  are coefficients of the *first fundamental form* and are given by

$$E = \left| \frac{\partial \mathbf{p}}{\partial u} \right|^2$$

$$F = \left( \frac{\partial \mathbf{p}}{\partial u} \cdot \frac{\partial \mathbf{p}}{\partial v} \right)$$

$$G = \left| \frac{\partial \mathbf{p}}{\partial v} \right|^2.$$

These are easily computed with the  $\partial \mathbf{p}/\partial u$  and  $\partial \mathbf{p}/\partial v$  values found earlier. The  $e$ ,  $f$ , and  $g$  are coefficients of the *second fundamental form*,

$$e = \left( \mathbf{n} \cdot \frac{\partial^2 \mathbf{p}}{\partial u^2} \right)$$

$$f = \left( \mathbf{n} \cdot \frac{\partial^2 \mathbf{p}}{\partial u \partial v} \right)$$

$$g = \left( \mathbf{n} \cdot \frac{\partial^2 \mathbf{p}}{\partial v^2} \right).$$

The two fundamental forms capture elementary metric properties of a surface, including notions of distance, angle, and curvature; see a differential geometry textbook such as Gray (1993) for details. To find  $e$ ,  $f$ , and  $g$ , it is necessary to compute the second-order partial derivatives  $\partial^2 \mathbf{p}/\partial u^2$  and so on.

For spheres, a little more algebra gives the second derivatives:

$$\frac{\partial^2 \mathbf{p}}{\partial u^2} = -\phi_{\max}^2(x, y, 0)$$

$$\frac{\partial^2 \mathbf{p}}{\partial u \partial v} = (\theta_{\max} - \theta_{\min}) z \phi_{\max}(-\sin \phi, \cos \phi, 0)$$

$$\frac{\partial^2 \mathbf{p}}{\partial v^2} = -(\theta_{\max} - \theta_{\min})^2(x, y, z).$$

*(Compute sphere  $\partial \mathbf{n}/\partial u$  and  $\partial \mathbf{n}/\partial v$ )* ≡

```

Vector3f d2Pduu = -phiMax * phiMax * Vector3f(pHit.x, pHit.y, 0);
Vector3f d2Pdvv = -(thetaMax - thetaMin) * (thetaMax - thetaMin) *
    Vector3f(pHit.x, pHit.y, pHit.z);
Vector3f d2Pdvv = -(thetaMax - thetaMin) * (thetaMax - thetaMin) *
    Vector3f(-sinPhi, cosPhi, 0.);

Sphere::phiMax 133
Sphere::thetaMax 133
Sphere::thetaMin 133
Vector3f 60
(Compute coefficients for fundamental forms 140)
(Compute  $\partial \mathbf{n}/\partial u$  and  $\partial \mathbf{n}/\partial v$  from fundamental form coefficients 140)

```

*(Compute coefficients for fundamental forms) ≡*

139, 146

```
Float E = Dot(dpdu, dpdu);
Float F = Dot(dpdu, dpdv);
Float G = Dot(dpdv, dpdv);
Vector3f N = Normalize(Cross(dpdu, dpdv));
Float e = Dot(N, d2Pduu);
Float f = Dot(N, d2Pdvv);
Float g = Dot(N, d2Pdvv);
```

*(Compute  $\partial n/\partial u$  and  $\partial n/\partial v$  from fundamental form coefficients) ≡*

139, 146

```
Float invEGF2 = 1 / (E * G - F * F);
Normal3f dndu = Normal3f((f * F - e * G) * invEGF2 * dpdu +
                           (e * F - f * E) * invEGF2 * dpdv);
Normal3f dndv = Normal3f((g * F - f * G) * invEGF2 * dpdu +
                           (f * F - g * E) * invEGF2 * dpdv);
```

### 3.2.4 SurfaceInteraction INITIALIZATION

Having computed the surface parameterization and all the relevant partial derivatives, the `SurfaceInteraction` structure can be initialized with the geometric information for this intersection. The `pError` value passed to the `SurfaceInteraction` constructor bounds the rounding error in the computed `pHit` point. It is initialized in the fragment *(Compute error bounds for sphere intersection)*, which is defined later, in Section 3.9.4.

*(Initialize SurfaceInteraction from parametric information) ≡*

134, 144, 148

```
*isect = (*ObjectToWorld)(
    SurfaceInteraction(pHit, pError, Point2f(u, v), -ray.d, dpdu, dpdv,
                      dndu, dndv, ray.time, this));
```

Since there is an intersection, the `tHit` parameter to the `Intersect()` method is updated with the parametric hit distance along the ray, which was stored in `tShapeHit`. Updating `*tHit` allows subsequent intersection tests to terminate early if the potential hit would be farther away than the existing intersection.

*(Update tHit for quadric intersection) ≡*

134, 144, 148

```
*tHit = (Float)tShapeHit;
```

A natural question to ask at this point is, “What effect does the world-to-object transformation have on the correct parametric distance to return?” Indeed, the intersection method has found a parametric distance to the intersection for the object space ray, which may have been translated, rotated, scaled, or worse when it was transformed from world space. However, it can be shown that the parametric distance to an intersection in object space is exactly the same as it would have been if the ray was left in world space and the intersection had been done there and, thus, `tHit` can be set directly. Note that if the object space ray’s direction had been normalized after the transformation, then this would no longer be the case and a correction factor related to the unnormalized ray’s length would be needed. This is another motivation for not normalizing the object space ray’s direction vector after transformation.

The `Sphere::IntersectP()` routine is almost identical to `Sphere::Intersect()`, but it does not initialize the `SurfaceInteraction` structure. Because the `Intersect()` and

Dot() 63  
 Float 1062  
 Normal3f 71  
 Point2f 68  
 Ray::d 73  
 Ray::time 73  
 Shape::ObjectToWorld 124  
 Sphere::Intersect() 134  
 SurfaceInteraction 116  
 Vector3f 60

`IntersectP()` methods are always so closely related, in the following we will not show implementations of `IntersectP()` for the remaining shapes.

```
(Sphere Method Definitions) +≡
bool Sphere::IntersectP(const Ray &r, bool testAlphaTexture) const {
    Float phi;
    Point3f pHit;
    (Transform Ray to object space 134)
    (Compute quadratic sphere coefficients 135)
    (Solve quadratic equation for t values 136)
    (Compute sphere hit position and φ 137)
    (Test sphere intersection against clipping parameters 137)
    return true;
}
```

### 3.2.5 SURFACE AREA

To compute the surface area of quadrics, we use a standard formula from integral calculus. If a curve  $y = f(x)$  from  $x = a$  to  $x = b$  is revolved around the  $x$  axis, the surface area of the resulting swept surface is

$$2\pi \int_a^b f(x) \sqrt{1 + (f'(x))^2} dx,$$

where  $f'(x)$  denotes the derivative  $df/dx$ .<sup>5</sup> Since most of our surfaces of revolution are only partially swept around the axis, we will instead use the formula

$$\phi_{\max} \int_a^b f(x) \sqrt{1 + (f'(x))^2} dx.$$

The sphere is a surface of revolution of a circular arc. The function that defines the profile curve along the  $z$  axis of the sphere is

$$f(z) = \sqrt{r^2 - z^2},$$

and its derivative is

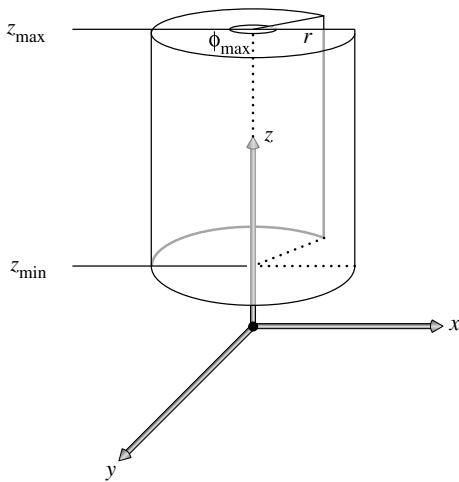
$$f'(z) = -\frac{z}{\sqrt{r^2 - z^2}}.$$

Recall that the sphere is clipped at  $z_{\min}$  and  $z_{\max}$ . The surface area is therefore

$$\begin{aligned} A &= \phi_{\max} \int_{z_{\min}}^{z_{\max}} \sqrt{r^2 - z^2} \sqrt{1 + \frac{z^2}{r^2 - z^2}} dz \\ &= \phi_{\max} \int_{z_{\min}}^{z_{\max}} \sqrt{r^2 - z^2 + z^2} dz \\ &= \phi_{\max} \int_{z_{\min}}^{z_{\max}} r dz \\ &= \phi_{\max} r (z_{\max} - z_{\min}). \end{aligned}$$

---

Float 1062  
Point3f 68  
Ray 73  
Sphere 133



**Figure 3.6: Basic Setting for the Cylinder Shape.** It has a radius of  $r$  and covers a range along the  $z$  axis. A partial cylinder may be swept by specifying a maximum  $\phi$  value.

For the full sphere  $\phi_{\max} = 2\pi$ ,  $z_{\min} = -r$ , and  $z_{\max} = r$ , so we have the standard formula  $A = 4\pi r^2$ , confirming that the formula makes sense.

*(Sphere Method Definitions)* +≡

```
Float Sphere::Area() const {
    return phiMax * radius * (zMax - zMin);
}
```

### 3.3 CYLINDERS

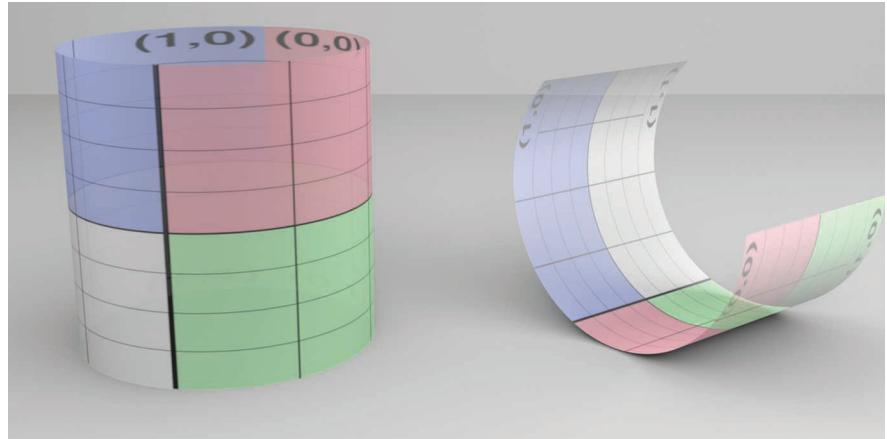
*(Cylinder Declarations)* ≡

```
class Cylinder : public Shape {
public:
    (Cylinder Public Methods 143)
protected:
    (Cylinder Private Data 143)
};
```

Another useful quadric is the cylinder; pbrt provides cylinder Shapes that are centered around the  $z$  axis. The implementation is in the files shapes/cylinder.h and shapes/cylinder.cpp. The user supplies a minimum and maximum  $z$  value for the cylinder, as well as a radius and maximum  $\phi$  sweep value (Figure 3.6).

In parametric form, a cylinder is described by the following equations:

```
Float 1062
Shape 123
Sphere 133
Sphere::phiMax 133
Sphere::radius 133
Sphere::zMax 133
Sphere::zMin 133
```



**Figure 3.7: Two Cylinders.** A complete cylinder is on the left, and a partial cylinder is on the right.

$$\begin{aligned}\phi &= u \phi_{\max} \\ x &= r \cos \phi \\ y &= r \sin \phi \\ z &= z_{\min} + v(z_{\max} - z_{\min}).\end{aligned}$$

Figure 3.7 shows a rendered image of two cylinders. Like the sphere image, the left cylinder is a complete cylinder, while the right one is a partial cylinder because it has a  $\phi_{\max}$  value less than  $2\pi$ .

$\langle \text{Cylinder Public Methods} \rangle \equiv$

142

```
Cylinder(const Transform *ObjectToWorld, const Transform *WorldToObject,
         bool reverseOrientation, Float radius, Float zMin, Float zMax,
         Float phiMax)
: Shape(ObjectToWorld, WorldToObject, reverseOrientation),
  radius(radius), zMin(std::min(zMin, zMax)),
  zMax(std::max(zMin, zMax)),
  phiMax(Radians(Clamp(phiMax, 0, 360))) { }
```

Bounds3f 76  
Clamp() 1062  
Cylinder 142  
Cylinder::phiMax 143  
Cylinder::radius 143  
Cylinder::zMax 143  
Cylinder::zMin 143  
Float 1062  
Point3f 68  
Radians() 1063  
Shape 123  
Transform 83

### 3.3.1 BOUNDING

As was done with the sphere, the cylinder bounding method computes a conservative bounding box using the  $z$  range but without taking into account the maximum  $\phi$ .

*{Cylinder Method Definitions}*  $\equiv$

```
    Bounds3f Cylinder::ObjectBound() const {
        return Bounds3f(Point3f(-radius, -radius, zMin),
                        Point3f( radius,  radius, zMax));
    }
```

### 3.3.2 INTERSECTION TESTS

The ray–cylinder intersection formula can be found by substituting the ray equation into the cylinder’s implicit equation, similarly to the sphere case. The implicit equation for an infinitely long cylinder centered on the  $z$  axis with radius  $r$  is

$$x^2 + y^2 - r^2 = 0.$$

Substituting the ray equation, Equation (2.3), we have

$$(o_x + t \mathbf{d}_x)^2 + (o_y + t \mathbf{d}_y)^2 = r^2.$$

When we expand this and find the coefficients of the quadratic equation  $at^2 + bt + c$ , we have

$$\begin{aligned} a &= \mathbf{d}_x^2 + \mathbf{d}_y^2 \\ b &= 2(\mathbf{d}_x o_x + \mathbf{d}_y o_y) \\ c &= o_x^2 + o_y^2 - r^2. \end{aligned}$$

*(Compute quadratic cylinder coefficients)* ≡

144

*(Initialize EFloat ray coordinate values 135)*  
 EFloat a = dx \* dx + dy \* dy;  
 EFloat b = 2 \* (dx \* ox + dy \* oy);  
 EFloat c = ox \* ox + oy \* oy - EFloat(radius) \* EFloat(radius);

The solution process for the quadratic equation is similar for all quadric shapes, so some fragments from the Sphere intersection method will be reused in the following.

*(Cylinder Method Definitions)* +≡

```
bool Cylinder::Intersect(const Ray &r, Float *tHit,
    SurfaceInteraction *isect, bool testAlphaTexture) const {
    Float phi;
    Point3f pHit;
    (Transform Ray to object space 134)
    (Compute quadratic cylinder coefficients 144)
    (Solve quadratic equation for t values 136)
    (Compute cylinder hit point and φ 145)
    (Test cylinder intersection against clipping parameters 145)
    (Find parametric representation of cylinder hit 145)
    (Compute error bounds for cylinder intersection 225)
    (Initialize SurfaceInteraction from parametric information 140)
    (Update tHit for quadric intersection 140)
    return true;
}
```

As with spheres, the implementation here refines the computed intersection point to ameliorate the effect of accumulated rounding error in the point computed by evaluating the ray equation; see Section 3.9.4. We can then invert the parametric description of the cylinder to compute  $\phi$  from  $x$  and  $y$ ; it turns out that the result is the same as for the sphere.

Cylinder 142  
 EFloat 218  
 Float 1062  
 Point3f 68  
 Ray 73  
 Sphere 133  
 SurfaceInteraction 116

```
(Compute cylinder hit point and  $\phi$ ) ≡
    pHit = ray((Float)tShapeHit);
(Refine cylinder intersection point 225)
    phi = std::atan2(pHit.y, pHit.x);
    if (phi < 0) phi += 2 * Pi;
```

144, 145

The next part of the intersection method makes sure that the hit is in the specified  $z$  range and that the angle  $\phi$  is acceptable. If not, it rejects the hit and checks  $t_1$  if it has not already been tried—this resembles the conditional logic in `Sphere::Intersect()`.

```
(Test cylinder intersection against clipping parameters) ≡
    if (pHit.z < zMin || pHit.z > zMax || phi > phiMax) {
        if (tShapeHit == t1) return false;
        tShapeHit = t1;
        if (t1.UpperBound() > ray.tMax) return false;
(Compute cylinder hit point and  $\phi$  145)
        if (pHit.z < zMin || pHit.z > zMax || phi > phiMax)
            return false;
    }
```

144

Again the  $u$  value is computed by scaling  $\phi$  to lie between 0 and 1. Straightforward inversion of the parametric equation for the cylinder’s  $z$  value gives the  $v$  parametric coordinate.

```
(Find parametric representation of cylinder hit) ≡
    Float u = phi / phiMax;
    Float v = (pHit.z - zMin) / (zMax - zMin);
(Compute cylinder  $\partial p/\partial u$  and  $\partial p/\partial v$  145)
(Compute cylinder  $\partial n/\partial u$  and  $\partial n/\partial v$  146)
```

144

The partial derivatives for a cylinder are quite easy to derive:

$$\begin{aligned}\frac{\partial p}{\partial u} &= (-\phi_{\max}y, \phi_{\max}x, 0) \\ \frac{\partial p}{\partial v} &= (0, 0, z_{\max} - z_{\min}).\end{aligned}$$

```
(Compute cylinder  $\partial p/\partial u$  and  $\partial p/\partial v$ ) ≡
    Vector3f dpdu(-phiMax * pHit.y, phiMax * pHit.x, 0);
    Vector3f dpdv(0, 0, zMax - zMin);
```

145

We again use the Weingarten equations to compute the parametric partial derivatives of the cylinder normal. The relevant partial derivatives are

```
Cylinder::phiMax 143
Cylinder::zMax 143
Cylinder::zMin 143
EFloat::UpperBound() 220
Float 1062
Pi 1063
Sphere::Intersect() 134
Vector3f 60
```

$$\begin{aligned}\frac{\partial^2 p}{\partial u^2} &= -\phi_{\max}^2(x, y, 0) \\ \frac{\partial^2 p}{\partial u \partial v} &= (0, 0, 0) \\ \frac{\partial^2 p}{\partial v^2} &= (0, 0, 0).\end{aligned}$$

*(Compute cylinder  $\partial \mathbf{n} / \partial u$  and  $\partial \mathbf{n} / \partial v$ )* ≡

145

```
Vector3f d2Pduu = -phiMax * phiMax * Vector3f(pHit.x, pHit.y, 0);
Vector3f d2Pdvv(0, 0, 0), d2Pdvv(0, 0, 0);
(Compute coefficients for fundamental forms 140)
(Compute  $\partial \mathbf{n} / \partial u$  and  $\partial \mathbf{n} / \partial v$  from fundamental form coefficients 140)
```

### 3.3.3 SURFACE AREA

A cylinder is just a rolled-up rectangle. If you unroll the rectangle, its height is  $z_{\max} - z_{\min}$ , and its width is  $r\phi_{\max}$ :

*(Cylinder Method Definitions)* +≡

```
Float Cylinder::Area() const {
    return (zMax - zMin) * radius * phiMax;
}
```

## 3.4 DISKS

*(Disk Declarations)* ≡

```
class Disk : public Shape {
public:
    (Disk Public Methods 146)
private:
    (Disk Private Data 147)
};
```

The disk is an interesting quadric since it has a particularly straightforward intersection routine that avoids solving the quadratic equation. In pbrt, a Disk is a circular disk of radius  $r$  at height  $h$  along the  $z$  axis. It is implemented in the files shapes/disk.h and shapes/disk.cpp.

In order to describe partial disks, the user may specify a maximum  $\phi$  value beyond which the disk is cut off (Figure 3.8). The disk can also be generalized to an annulus by specifying an inner radius,  $r_i$ . In parametric form, it is described by

$$\begin{aligned}\phi &= u \phi_{\max} \\ x &= ((1 - v)r_i + vr) \cos \phi \\ y &= ((1 - v)r_i + vr) \sin \phi \\ z &= h.\end{aligned}$$

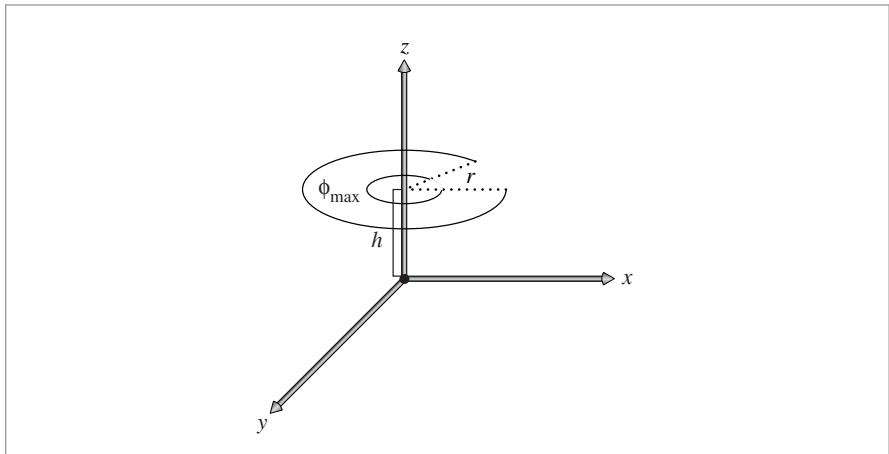
Figure 3.9 is a rendered image of two disks.

*(Disk Public Methods)* ≡

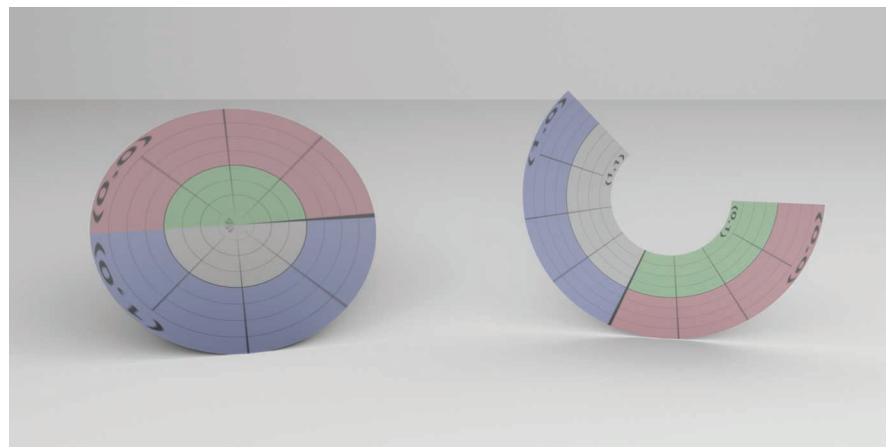
146

```
Disk(const Transform *ObjectToWorld, const Transform *WorldToObject,
      bool reverseOrientation, Float height, Float radius,
      Float innerRadius, Float phiMax)
: Shape(ObjectToWorld, WorldToObject, reverseOrientation),
  height(height), radius(radius), innerRadius(innerRadius),
  phiMax(Radians(Clamp(phiMax, 0, 360))) { }
```

Clamp() 1062  
 Cylinder 142  
 Cylinder::phiMax 143  
 Cylinder::radius 143  
 Cylinder::zMax 143  
 Cylinder::zMin 143  
 Disk 146  
 Disk::phiMax 147  
 Float 1062  
 Radians() 1063  
 Shape 123  
 Transform 83  
 Vector3f 60



**Figure 3.8: Basic Setting for the Disk Shape.** The disk has radius  $r$  and is located at height  $h$  along the  $z$  axis. A partial disk may be swept by specifying a maximum  $\phi$  value and an inner radius  $r_i$ .



**Figure 3.9: Two Disks.** A complete disk is on the left, and a partial disk is on the right.

$\langle$ Disk Private Data $\rangle \equiv$

```
const Float height, radius, innerRadius, phiMax;
```

146

### 3.4.1 BOUNDING

Float 1062

The bounding method is quite straightforward; it computes a bounding box centered at the height of the disk along  $z$ , with extent of radius in both the  $x$  and  $y$  directions.

```
(Disk Method Definitions) ≡
    Bounds3f Disk::ObjectBound() const {
        return Bounds3f(Point3f(-radius, -radius, height),
                        Point3f( radius,  radius, height));
    }
```

### 3.4.2 INTERSECTION TESTS

Intersecting a ray with a disk is also easy. The intersection of the ray with the  $z = h$  plane that the disk lies in is found and the intersection point is checked to see if it lies inside the disk.

```
(Disk Method Definitions) +≡
bool Disk::Intersect(const Ray &r, Float *tHit,
                     SurfaceInteraction *isect, bool testAlphaTexture) const {
    <Transform Ray to object space 134>
    <Compute plane intersection for disk 148>
    <See if hit point is inside disk radii and  $\phi_{\max}$  149>
    <Find parametric representation of disk hit 149>
    <Refine disk intersection point 225>
    <Compute error bounds for disk intersection 225>
    <Initialize SurfaceInteraction from parametric information 140>
    <Update tHit for quadric intersection 140>
    return true;
}
```

The first step is to compute the parametric  $t$  value where the ray intersects the plane that the disk lies in. We want to find  $t$  such that the  $z$  component of the ray's position is equal to the height of the disk. Thus,

$$h = \mathbf{o}_z + t \mathbf{d}_z$$

and

$$t = \frac{h - \mathbf{o}_z}{\mathbf{d}_z}.$$

The intersection method computes a  $t$  value checks to see if it is inside the legal range of values (0,  $t_{\text{Max}}$ ). If not, the routine can return false.

<i>(Compute plane intersection for disk) ≡</i>	148
<i>(Reject disk intersections for rays parallel to the disk's plane 149)</i>	
Float tShapeHit = (height - ray.o.z) / ray.d.z;	
if (tShapeHit <= 0    tShapeHit >= ray.tMax)	
return false;	

If the ray is parallel to the disk's plane (i.e., the  $z$  component of its direction is zero), no intersection is reported. The case where a ray is both parallel to the disk's plane and lies within the plane is somewhat ambiguous, but it's most reasonable to define intersecting

Bounds3f 76
Disk 146
Disk::height 147
Disk::radius 147
Float 1062
Point3f 68
Ray 73
SurfaceInteraction 116

a disk edge-on as “no intersection.” This case must be handled explicitly so that NaN floating-point values aren’t generated by the following code.

```
(Reject disk intersections for rays parallel to the disk's plane) ≡ 148
    if (ray.d.z == 0)
        return false;
```

Now the intersection method can compute the point `pHit` where the ray intersects the plane. Once the plane intersection is known, `false` is returned if the distance from the hit to the center of the disk is more than `Disk::radius` or less than `Disk::innerRadius`. This process can be optimized by actually computing the squared distance to the center, taking advantage of the fact that the  $x$  and  $y$  coordinates of the center point  $(0, 0, \text{height})$  are zero, and the  $z$  coordinate of `pHit` is equal to `height`.

```
(See if hit point is inside disk radii and  $\phi_{\max}$ ) ≡ 148
    Point3f pHit = ray(tShapeHit);
    Float dist2 = pHit.x * pHit.x + pHit.y * pHit.y;
    if (dist2 > radius * radius || dist2 < innerRadius * innerRadius)
        return false;
(Test disk  $\phi$  value against  $\phi_{\max}$  149)
```

If the distance check passes, a final test makes sure that the  $\phi$  value of the hit point is between zero and  $\phi_{\max}$ , specified by the caller. Inverting the disk’s parameterization gives the same expression for  $\phi$  as the other quadric shapes.

```
(Test disk  $\phi$  value against  $\phi_{\max}$ ) ≡ 149
    Float phi = std::atan2(pHit.y, pHit.x);
    if (phi < 0) phi += 2 * Pi;
    if (phi > phiMax)
        return false;
```

If we’ve gotten this far, there is an intersection with the disk. The parameter  $u$  is scaled to reflect the partial disk specified by  $\phi_{\max}$ , and  $v$  is computed by inverting the parametric equation. The equations for the partial derivatives at the hit point can be derived with a process similar to that used for the previous quadrics. Because the normal of a disk is the same everywhere, the partial derivatives  $\partial \mathbf{n} / \partial u$  and  $\partial \mathbf{n} / \partial v$  are both trivially  $(0, 0, 0)$ .

```
(Find parametric representation of disk hit) ≡ 148
Disk::innerRadius 147
Disk::phiMax 147
Disk::radius 147
Float 1062
Normal3f 71
Pi 1063
Point3f 68
Vector3f 60
    Float u = phi / phiMax;
    Float rHit = std::sqrt(dist2);
    Float oneMinusV = ((rHit - innerRadius) /
                        (radius - innerRadius));
    Float v = 1 - oneMinusV;
    Vector3f dpdu(-phiMax * pHit.y, phiMax * pHit.x, 0);
    Vector3f dpdv = Vector3f(pHit.x, pHit.y, 0.) * (innerRadius - radius) /
                    rHit;
    Normal3f dndu(0, 0, 0), dndv(0, 0, 0);
```

### 3.4.3 SURFACE AREA

Disks have trivially computed surface area, since they're just portions of an annulus:

$$A = \frac{\phi_{\max}}{2} (r^2 - r_i^2).$$

*(Disk Method Definitions)* +≡

```
Float Disk::Area() const {
    return phiMax * 0.5 * (radius * radius - innerRadius * innerRadius);
}
```

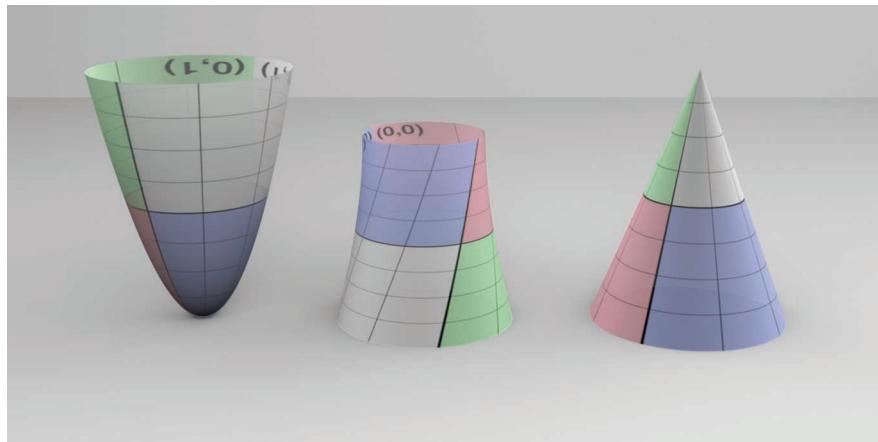
## 3.5 OTHER QUADRICS

pbrt supports three more quadrics: cones, paraboloids, and hyperboloids. They are implemented in the source files `shapes/cone.h`, `shapes/cone.cpp`, `shapes/paraboloid.h`, `shapes/paraboloid.cpp`, `shapes/hyperboloid.h`, and `shapes/hyperboloid.cpp`. We won't include their full implementations here, since the techniques used to derive their quadratic intersection coefficients, parametric coordinates, and partial derivatives should now be familiar. However, we will briefly summarize the implicit and parametric forms of these shapes. A rendered image of the three of them is in Figure 3.10.

### 3.5.1 CONES

The implicit equation of a cone centered on the  $z$  axis with radius  $r$  and height  $h$  is

$$\left(\frac{hx}{r}\right)^2 + \left(\frac{hy}{r}\right)^2 - (z - h)^2 = 0.$$



**Figure 3.10: The Remaining Quadric Shapes.** From left to right: the paraboloid, the hyperboloid, and the cone.

Disk 146

Disk::innerRadius 147

Disk::phiMax 147

Disk::radius 147

Float 1062

Cones are also described parametrically:

$$\begin{aligned}\phi &= u \phi_{\max} \\ x &= r(1-v) \cos \phi \\ y &= r(1-v) \sin \phi \\ z &= vh.\end{aligned}$$

The partial derivatives at a point on a cone are

$$\begin{aligned}\frac{\partial \mathbf{p}}{\partial u} &= (-\phi_{\max}y, \phi_{\max}x, 0) \\ \frac{\partial \mathbf{p}}{\partial v} &= \left(-\frac{x}{1-v}, -\frac{y}{1-v}, h\right),\end{aligned}$$

and the second partial derivatives are

$$\begin{aligned}\frac{\partial^2 \mathbf{p}}{\partial u^2} &= -\phi_{\max}^2(x, y, 0) \\ \frac{\partial^2 \mathbf{p}}{\partial u \partial v} &= \frac{\phi_{\max}}{1-v}(y, -x, 0) \\ \frac{\partial^2 \mathbf{p}}{\partial v^2} &= (0, 0, 0).\end{aligned}$$

### 3.5.2 PARABOLOIDS

The implicit equation of a paraboloid centered on the  $z$  axis with radius  $r$  and height  $h$  is

$$\frac{hx^2}{r^2} + \frac{hy^2}{r^2} - z = 0,$$

and its parametric form is

$$\begin{aligned}\phi &= u \phi_{\max} \\ z &= v(z_{\max} - z_{\min}) \\ r &= r_{\max} \sqrt{\frac{z}{z_{\max}}} \\ x &= r \cos \phi \\ y &= r \sin \phi.\end{aligned}$$

The partial derivatives are

$$\begin{aligned}\frac{\partial \mathbf{p}}{\partial u} &= (-\phi_{\max}y, \phi_{\max}x, 0) \\ \frac{\partial \mathbf{p}}{\partial v} &= (z_{\max} - z_{\min}) \left(\frac{x}{2z}, \frac{y}{2z}, 1\right),\end{aligned}$$

and

$$\begin{aligned}\frac{\partial^2 p}{\partial u^2} &= -\phi_{\max}^2(x, y, 0) \\ \frac{\partial^2 p}{\partial u \partial v} &= \phi_{\max}(z_{\max} - z_{\min}) \left( -\frac{y}{2z}, \frac{x}{2z}, 0 \right) \\ \frac{\partial^2 p}{\partial v^2} &= -(z_{\max} - z_{\min})^2 \left( \frac{x}{4z^2}, \frac{y}{4z^2}, 0 \right).\end{aligned}$$

### 3.5.3 HYPERBOLOID

Finally, the implicit form of the hyperboloid is

$$x^2 + y^2 - z^2 = -1,$$

and the parametric form is

$$\begin{aligned}\phi &= u \phi_{\max} \\ x_r &= (1 - v)x_1 + v x_2 \\ y_r &= (1 - v)y_1 + v y_2 \\ x &= x_r \cos \phi - y_r \sin \phi \\ y &= x_r \sin \phi + y_r \cos \phi \\ z &= (1 - v)z_1 + v z_2.\end{aligned}$$

The partial derivatives are

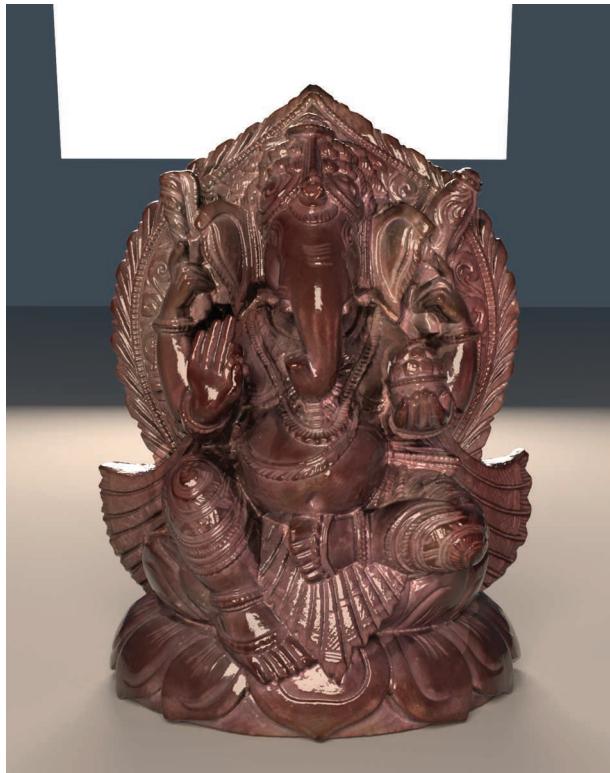
$$\begin{aligned}\frac{\partial p}{\partial u} &= (-\phi_{\max} y, \phi_{\max} x, 0) \\ \frac{\partial p}{\partial v} &= ((x_2 - x_1) \cos \phi - (y_2 - y_1) \sin \phi, (x_2 - x_1) \sin \phi + (y_2 - y_1) \cos \phi, z_2 - z_1),\end{aligned}$$

and

$$\begin{aligned}\frac{\partial^2 p}{\partial u^2} &= -\phi_{\max}^2(x, y, 0) \\ \frac{\partial^2 p}{\partial u \partial v} &= \phi_{\max} \left( -\frac{\partial p_y}{\partial v}, \frac{\partial p_x}{\partial v}, 0 \right) \\ \frac{\partial^2 p}{\partial v^2} &= (0, 0, 0).\end{aligned}$$

## 3.6 TRIANGLE MESHES

The triangle is one of the most commonly used shapes in computer graphics; complex scenes may be modeled using millions of triangles to achieve great detail. (Figure 3.11 shows an image of a complex triangle mesh of over four million triangles.) While a natural representation would be to have a `Triangle` shape implementation where each triangle stored the positions of its three vertices, a more memory-efficient representation is to separately store entire triangle meshes with an array of vertex positions where each individual triangle just stores three offsets into this array for its three vertices.



**Figure 3.11: Ganesha Model.** This triangle mesh contains over four million individual triangles. It was created from a real statue using a 3D scanner that uses structured light to determine shapes of objects.

To see why this is the case, consider the celebrated Euler-Poincaré formula, which relates the number of vertices  $V$ , edges  $E$ , and faces  $F$  on closed discrete meshes as

$$V - E + F = 2(1 - g),$$

where  $g \in \mathbb{N}$  is the *genus* of the mesh. The genus is usually a small number and can be interpreted as the number of “handles” in the mesh (analogous to a handle of a teacup). On a triangle mesh, the number of edges and vertices is furthermore related by the identity

$$E = \frac{3}{2}F.$$

This can be seen by dividing each edge into two parts associated with the two adjacent triangles. There are  $3F$  such half-edges, and all co-located pairs constitute the  $E$  mesh edges. For large closed triangle meshes, the overall effect of the genus usually becomes negligible and we can combine the previous two equations (with  $g = 0$ ) to obtain

$$V \approx 2F.$$

In other words, there are approximately twice as many vertices as faces. Since each face references three vertices, every vertex is (on average) referenced a total of six times. Thus, when vertices are shared, the total amortized storage required per triangle will be 12 bytes of memory for the offsets (at 4 bytes for three 32-bit integer offsets) plus half of the storage for one vertex—6 bytes, assuming three 4-byte floats are used to store the vertex position—for a total of 18 bytes per triangle. This is much better than the 36 bytes per triangle that storing the three positions directly would require. The relative storage savings are even better when there are per-vertex surface normals or texture coordinates in a mesh.

pbrt uses the `TriangleMesh` structure to store the shared information about a triangle mesh.

```
(Triangle Declarations) ≡
struct TriangleMesh {
    (TriangleMesh Public Methods)
    (TriangleMesh Data 155)
};
```

The arguments to the `TriangleMesh` constructor are as follows:

- `ObjectToWorld`: The object-to-world transformation for the mesh.
- `nTriangles`: The total number of triangles in the mesh.
- `vertexIndices`: A pointer to an array of vertex indices. For the  $i$ th triangle, its three vertex positions are  $P[\text{vertexIndices}[3*i]]$ ,  $P[\text{vertexIndices}[3*i+1]]$ , and  $P[\text{vertexIndices}[3*i+2]]$ .
- `nVertices`: The total number of vertices in the mesh.
- `P`: An array of `nVertices` vertex positions.
- `S`: An optional array of tangent vectors, one per vertex in the mesh. These are used to compute shading tangents.
- `N`: An optional array of normal vectors, one per vertex in the mesh. If present, these are interpolated across triangle faces to compute shading normals.
- `UV`: An optional array of parametric  $(u, v)$  values, one for each vertex.
- `alphaMask`: An optional *alpha mask* texture, which can be used to cut away parts of triangle surfaces.

Triangles have a dual role among the shapes in pbrt: not only are they frequently directly specified in scene description files, but other shapes often tessellate themselves into triangle meshes. For example, subdivision surfaces end up creating a mesh of triangles to approximate the smooth limit surface. Ray intersections are performed against these triangles, rather than directly against the subdivision surface (Section 3.8.2).

Due to this second role, it's important that code that creates triangle meshes be able to specify the parameterization of the triangles. If a triangle was created by evaluating the position of a parametric surface at three particular  $(u, v)$  coordinate values, for example, those  $(u, v)$  values should be interpolated to compute the  $(u, v)$  value at ray intersection points inside the triangle. Explicitly specified  $(u, v)$  values are also useful for texture mapping, where an external program that created a triangle mesh may want to assign  $(u, v)$  coordinates to the mesh so that a texture map assigns color to the mesh surface in the desired way.

The `TriangleMesh` constructor copies the relevant information and stores it in member variables. In particular, it makes its own copies of `vertexIndices`, `P`, `N`, `S`, and `UV`, allowing the caller to retain ownership of the data being passed in.

*(Triangle Method Definitions) ≡*

```
TriangleMesh::TriangleMesh(const Transform &ObjectToWorld,
    int nTriangles, const int *vertexIndices, int nVertices,
    const Point3f *P, const Vector3f *S, const Normal3f *N,
    const Point2f *UV,
    const std::shared_ptr<Texture<Float>> &alphaMask)
: nTriangles(nTriangles), nVertices(nVertices),
  vertexIndices(vertexIndices, vertexIndices + 3 * nTriangles),
  alphaMask(alphaMask) {
    <Transform mesh vertices to world space 155>
    <Copy UV, N, and S vertex data, if present>
}
```

*(TriangleMesh Data) ≡*

154

```
const int nTriangles, nVertices;
std::vector<int> vertexIndices;
std::unique_ptr<Point3f[]> p;
std::unique_ptr<Normal3f[]> n;
std::unique_ptr<Vector3f[]> s;
std::unique_ptr<Point2f[]> uv;
std::shared_ptr<Texture<Float>> alphaMask;
```

Unlike the other shapes that leave the shape description in object space and then transform incoming rays from world space to object space, triangle meshes transform the shape into world space and thus save the work of transforming incoming rays into object space and the work of transforming the intersection's geometric representation out to world space. This is a good idea because this operation can be performed once at start-up, avoiding transforming rays many times during rendering. Using this approach with quadrics is more complicated, although possible—see Exercise 3.1 at the end of the chapter for more information.

Float 1062  
Normal3f 71  
Point2f 68  
Point3f 68  
Shape 123  
Texture 614  
Transform 83  
Triangle 156  
TriangleMesh 154  
TriangleMesh::nVertices 155  
TriangleMesh::p 155  
Vector3f 60

*<Transform mesh vertices to world space> ≡*

155

```
p.reset(new Point3f[nVertices]);
for (int i = 0; i < nVertices; ++i)
    p[i] = ObjectToWorld(P[i]);
```

The fragment *<Copy uv, N, and S vertex data, if present>* just allocates the appropriate amount of space and copies the appropriate values. Normals and tangent vectors, if present, are also transformed to object space. This fragment's implementation isn't included here.

### 3.6.1 TRIANGLE

The `Triangle` class actually implements the `Shape` interface. It represents a single triangle.

```
(Triangle Declarations) +≡
class Triangle : public Shape {
public:
    (Triangle Public Methods 156)
private:
    (Triangle Private Methods 164)
    (Triangle Private Data 156)
};
```

Triangle doesn't store much data—just a pointer to the parent TriangleMesh that it came from and a pointer to its three vertex indices in the mesh.

*(Triangle Public Methods) ≡* 156

```
Triangle(const Transform *ObjectToWorld, const Transform *WorldToObject,
         bool reverseOrientation,
         const std::shared_ptr<TriangleMesh> &mesh, int triNumber)
: Shape(ObjectToWorld, WorldToObject, reverseOrientation),
  mesh(mesh) {
    v = &mesh->vertexIndices[3 * triNumber];
}
```

Note that the implementation stores a pointer to the first vertex *index*, instead of storing three pointers to the vertices themselves. This reduces the amount of storage required for each Triangle at a cost of another level of indirection.

*(Triangle Private Data) ≡* 156

```
std::shared_ptr<TriangleMesh> mesh;
const int *v;
```

Because a number of other shape representations in pbrt convert themselves into triangle meshes, the utility function CreateTriangleMesh() takes care of creating an underlying TriangleMesh as well as a Triangle for each triangle in the mesh. It returns a vector of triangle shapes.

*(Triangle Method Definitions) +≡*

```
std::vector<std::shared_ptr<Shape>> CreateTriangleMesh(
    const Transform *ObjectToWorld, const Transform *WorldToObject,
    bool reverseOrientation, int nTriangles,
    const int *vertexIndices, int nVertices, const Point3f *p,
    const Vector3f *s, const Normal3f *n, const Point2f *uv,
    const std::shared_ptr<Texture<Float>> &alphaMask) {
    std::shared_ptr<TriangleMesh> mesh = std::make_shared<TriangleMesh>(
        *ObjectToWorld, nTriangles, vertexIndices, nVertices, p, s, n, uv,
        alphaMask);
    std::vector<std::shared_ptr<Shape>> tris;
    for (int i = 0; i < nTriangles; ++i)
        tris.push_back(std::make_shared<Triangle>(*ObjectToWorld,
            *WorldToObject, reverseOrientation, mesh, i));
    return tris;
}
```

Float 1062  
 Normal3f 71  
 Point2f 68  
 Point3f 68  
 Shape 123  
 Texture 614  
 Transform 83  
 Triangle 156  
 TriangleMesh 154  
 TriangleMesh::vertexIndices 155  
 Vector3f 60

The object space bound of a triangle is easily found by computing a bounding box that encompasses its three vertices. Because the vertex positions  $p$  are transformed to world space in the constructor, the implementation here has to transform them back to object space before computing their bound.

*(Triangle Method Definitions) +≡*

```
Bounds3f Triangle::ObjectBound() const {
    <Get triangle vertices in p0, p1, and p2 157>
    return Union(Bounds3f((*WorldToObject)(p0), (*WorldToObject)(p1)),
                 (*WorldToObject)(p2));
}
```

*(Get triangle vertices in p0, p1, and p2) ≡*

157, 167, 839

```
const Point3f &p0 = mesh->p[v[0]];
const Point3f &p1 = mesh->p[v[1]];
const Point3f &p2 = mesh->p[v[2]];
```

The Triangle shape is one of the shapes that can compute a better world space bound than can be found by transforming its object space bounding box to world space. Its world space bound can be directly computed from the world space vertices.

*(Triangle Method Definitions) +≡*

```
Bounds3f Triangle::WorldBound() const {
    <Get triangle vertices in p0, p1, and p2 157>
    return Union(Bounds3f(p0, p1), p2);
}
```

### 3.6.2 TRIANGLE INTERSECTION

The structure of the triangle shape’s Intersect() method follows the form of earlier intersection test methods: a geometric test is applied to determine if there is an intersection and, if so, further information is computed about the intersection to return in the given SurfaceInteraction.

*(Triangle Method Definitions) +≡*

```
bool Triangle::Intersect(const Ray &ray, Float *tHit,
                        SurfaceInteraction *isect, bool testAlphaTexture) const {
    <Get triangle vertices in p0, p1, and p2 157>
    <Perform ray-triangle intersection test 158>
    <Compute triangle partial derivatives 164>
    <Compute error bounds for triangle intersection 227>
    <Interpolate (u, v) parametric coordinates and hit point 164>
    <Test intersection against alpha texture, if present 165>
    <Fill in SurfaceInteraction from triangle hit 165>
    *tHit = t;
    return true;
}
```

Bounds3f 76

Float 1062

Point3f 68

Ray 73

Shape::WorldToObject 124

SurfaceInteraction 116

Triangle 156

Triangle::mesh 156

TriangleMesh::p 155

pbrt's ray-triangle intersection test is based on first computing an affine transformation that transforms the ray such that its origin is at  $(0, 0, 0)$  in the transformed coordinate system and such that its direction is along the  $+z$  axis. Triangle vertices are also transformed into this coordinate system before the intersection test is performed. In the following, we'll see that applying this coordinate system transformation simplifies the intersection test logic since, for example, the  $x$  and  $y$  coordinates of any intersection point must be zero. Later, in Section 3.9.3, we'll see that this transformation makes it possible to have a *watertight* ray-triangle intersection algorithm, such that intersections with tricky rays like those that hit the triangle right on the edge are never incorrectly reported as misses.

*(Perform ray-triangle intersection test)  $\equiv$*

157

*(Transform triangle vertices to ray coordinate space 158)*

*(Compute edge function coefficients e0, e1, and e2 161)*

*(Fall back to double-precision test at triangle edges)*

*(Perform triangle edge and determinant tests 162)*

*(Compute scaled hit distance to triangle and test against ray t range 162)*

*(Compute barycentric coordinates and t value for triangle intersection 163)*

*(Ensure that computed triangle t is conservatively greater than zero 234)*

There are three steps to computing the transformation from world space to the ray-triangle intersection coordinate space: a translation  $T$ , a coordinate permutation  $P$ , and a shear  $S$ . Rather than computing explicit transformation matrices for each of these and then computing an aggregate transformation matrix  $M = SPT$  to transform vertices to the coordinate space, the following implementation applies each step of the transformation directly, which ends up being a more efficient approach.

*(Transform triangle vertices to ray coordinate space)  $\equiv$*

158

*(Translate vertices based on ray origin 158)*

*(Permute components of triangle vertices and ray direction 159)*

*(Apply shear transformation to translated vertex positions 159)*

The translation that places the ray origin at the origin of the coordinate system is:

$$T = \begin{pmatrix} 1 & 0 & 0 & -o_x \\ 0 & 1 & 0 & -o_y \\ 0 & 0 & 1 & -o_z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

This transformation doesn't need to be explicitly applied to the ray origin, but we will apply it to the three triangle vertices.

*(Translate vertices based on ray origin)  $\equiv$*

158

```
Point3f p0t = p0 - Vector3f(ray.o);
```

```
Point3f p1t = p1 - Vector3f(ray.o);
```

```
Point3f p2t = p2 - Vector3f(ray.o);
```

Point3f 68

Vector3f 60

Next, the three dimensions of the space are permuted so that the  $z$  dimension is the one where the absolute value of the ray's direction is largest. The  $x$  and  $y$  dimensions are arbitrarily assigned to the other two dimensions. This step ensures that if, for example,

the original ray's  $z$  direction is zero, then a dimension with non-zero magnitude is mapped to  $+z$ .

For example, if the ray's direction had the largest magnitude in  $x$ , the permutation would be:

$$T = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

As before, it's easiest to just permute the dimensions of the ray direction and the translated triangle vertices directly.

*(Permute components of triangle vertices and ray direction) ≡*

158

```
int kz = MaxDimension(Abs(ray.d));
int kx = kz + 1; if (kx == 3) kx = 0;
int ky = kx + 1; if (ky == 3) ky = 0;
Vector3f d = Permute(ray.d, kx, ky, kz);
p0t = Permute(p0t, kx, ky, kz);
p1t = Permute(p1t, kx, ky, kz);
p2t = Permute(p2t, kx, ky, kz);
```

Finally, a shear transformation aligns the ray direction with the  $+z$  axis:

$$S = \begin{pmatrix} 1 & 0 & -d_x/d_z & 0 \\ 0 & 1 & -d_y/d_z & 0 \\ 0 & 0 & 1/d_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

To see how this transformation works, consider its operation on the ray direction vector  $[d_x \ d_y \ d_z \ 0]^T$ .

For now, only the  $x$  and  $y$  dimensions are sheared; we can wait and shear the  $z$  dimension only if the ray actually intersects the triangle.

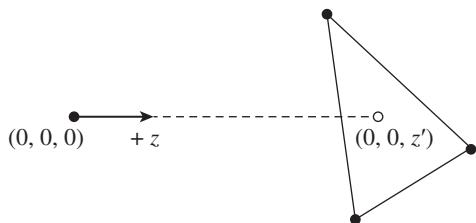
*(Apply shear transformation to translated vertex positions) ≡*

158

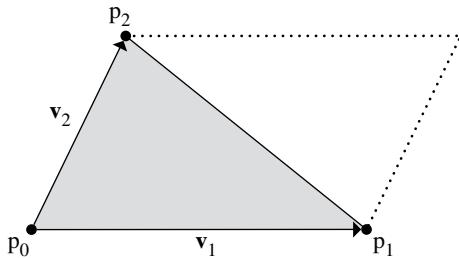
```
Float Sx = -d.x / d.z;
Float Sy = -d.y / d.z;
Float Sz = 1.f / d.z;
p0t.x += Sx * p0t.z;
p0t.y += Sy * p0t.z;
p1t.x += Sx * p1t.z;
p1t.y += Sy * p1t.z;
p2t.x += Sx * p2t.z;
p2t.y += Sy * p2t.z;
```

Float 1062  
Point3::Permute() 71  
Vector3::Abs() 63  
Vector3::MaxDimension() 66  
Vector3::Permute() 66  
Vector3f 60

Note that the calculations for the coordinate permutation and the shear coefficients only depend on the given ray; they are independent of the triangle. In a high-performance ray tracer, we might want to compute these values once and store them in the Ray class, rather than recomputing them for each triangle the ray is intersected with.



**Figure 3.12:** In the ray–triangle intersection coordinate system, the ray starts at the origin and goes along the  $+z$  axis. The intersection test can be performed by considering only the  $xy$  projection of the ray and the triangle vertices, which in turn reduces to determining if the 2D point  $(0, 0)$  is within the triangle.



**Figure 3.13:** The area of a triangle with two edges given by vectors  $v_1$  and  $v_2$  is one-half of the area of the parallelogram shown here. The parallelogram area is given by the length of the cross product of  $v_1$  and  $v_2$ .

With the triangle vertices transformed to this coordinate system, our task now is to find if the ray starting from the origin and traveling along the  $+z$  axis intersects the transformed triangle. Because of the way the coordinate system was constructed, this problem is equivalent to the 2D problem of determining if the  $x, y$  coordinates  $(0, 0)$  are inside the  $xy$  projection of the triangle (Figure 3.12).

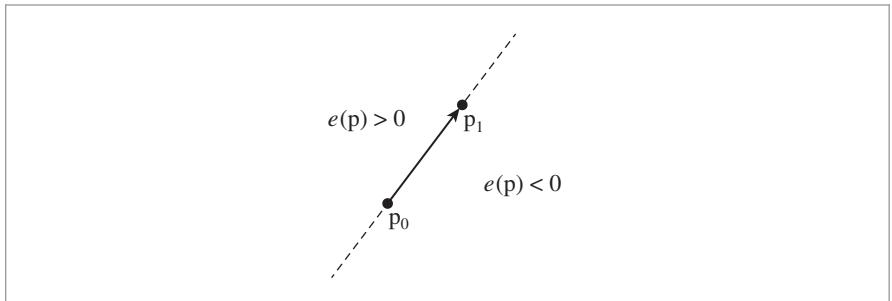
To understand how the intersection algorithm works, first recall from Figure 2.5 that the cross product of two vectors gives the area of the parallelogram that they define. In 2D, with vectors  $\mathbf{a}$  and  $\mathbf{b}$ , the area is

$$\mathbf{a}_x \mathbf{b}_y - \mathbf{b}_x \mathbf{a}_y.$$

Half of this area is the area of the triangle that they define. Thus, we can see that in 2D, the area of a triangle with vertices  $p_0, p_1$ , and  $p_2$  is

$$\frac{1}{2}(p_{1x} - p_{0x})(p_{2y} - p_{0y}) - (p_{2x} - p_{0x})(p_{1y} - p_{0y}).$$

Figure 3.13 visualizes this geometrically.



**Figure 3.14:** The edge function  $e(p)$  characterizes points with respect to an oriented line between two points  $p_0$  and  $p_1$ . The value of the edge function is positive for points  $p$  to the left of the line, zero for points on the line, and negative for points to the right of the line. The ray-triangle intersection algorithm uses an edge function that is twice the signed area of the triangle formed by the three points.

We'll use this expression of triangle area to define a signed *edge function*: given two triangle vertices  $p_0$  and  $p_1$ , then we can define the directed edge function  $e$  as the function that gives the area of the triangle given by  $p_0$ ,  $p_1$ , and a given third point  $p$ :

$$e(p) = (p_{1x} - p_{0x})(p_y - p_{0y}) - (p_x - p_{0x})(p_{1y} - p_{0y}). \quad [3.1]$$

(See Figure 3.14.) The edge function gives a positive value for points to the left of the line, and negative value for points to the right. Thus, if a point has edge function values of the same sign for all three edges of a triangle, it must be on the same side of all three edges and thus must be inside the triangle.

Thanks to the coordinate system transformation, the point we're testing  $p$  is  $(0, 0)$ . This simplifies the edge function expressions. For example, for the edge  $e_0$  from  $p_1$  to  $p_2$ , we have:

$$\begin{aligned} e_0(p) &= (p_{2x} - p_{1x})(p_y - p_{1y}) - (p_x - p_{1x})(p_{2y} - p_{1y}) \\ &= (p_{2x} - p_{1x})(-p_{1y}) - (-p_{1x})(p_{2y} - p_{1y}) \\ &= p_{1x} p_{2y} - p_{2x} p_{1y}. \end{aligned} \quad [3.2]$$

In the following, we'll use the indexing scheme that the edge function  $e_i$  corresponds to the directed edge from vertex  $p_i$  to  $p_{(i+1) \bmod 3}$ .

*(Compute edge function coefficients e0, e1, and e2) ≡*

158

```
Float e0 = p1t.x * p2t.y - p1t.y * p2t.x;
Float e1 = p2t.x * p0t.y - p2t.y * p0t.x;
Float e2 = p0t.x * p1t.y - p0t.y * p1t.x;
```

In the rare case that any of the edge function values is exactly zero, it's not possible to be sure if the ray hits the triangle or not, and the edge equations are reevaluated using double-precision floating-point arithmetic. (Section 3.9.3 discusses the need for this step in more detail.) The fragment that implements this computation, *(Fall back to double-precision test at triangle edges)*, is just a reimplementations of *(Compute edge function coefficients e0, e1, and e2)* using doubles and so isn't included here.

Given the values of the three edge functions, we have our first two opportunities to determine that there is no intersection. First, if the signs of the edge function values differ, then the point  $(0, 0)$  is not on the same side of all three edges and therefore is outside the triangle. Second, if the sum of the three edge function values is zero, then the ray is approaching the triangle edge-on, and we report no intersection. (For a closed triangle mesh, the ray will hit a neighboring triangle instead.)

*(Perform triangle edge and determinant tests) ≡*

158

```

if ((e0 < 0 || e1 < 0 || e2 < 0) && (e0 > 0 || e1 > 0 || e2 > 0))
    return false;
Float det = e0 + e1 + e2;
if (det == 0)
    return false;
```

Because the ray starts at the origin, has unit length, and is along the  $+z$  axis, the  $z$  coordinate value of the intersection point is equal to the intersection's parametric  $t$  value. To compute this  $z$  value, we first need to go ahead and apply the shear transformation to the  $z$  coordinates of the triangle vertices. Given these  $z$  values, the *barycentric coordinates* of the intersection point in the triangle can be used to interpolate them across the triangle. They are given by dividing each edge function value by the sum of edge function values:

$$b_i = \frac{e_i}{e_0 + e_1 + e_2}.$$

Thus, the  $b_i$  sum to one.

The interpolated  $z$  value is given by

$$z = b_0 z_0 + b_1 z_1 + b_2 z_2,$$

where  $z_i$  are the coordinates of the three vertices in the ray–triangle intersection coordinate system.

In order to save the cost of the floating-point division to compute  $b_i$  in cases where the final  $t$  value is out of the range of valid  $t$  values, the implementation here first computes  $t$  by interpolating  $z_i$  with  $e_i$  (in other words, not yet performing the division by  $d = e_0 + e_1 + e_2$ ). If the sign of  $d$  and the sign of the interpolated  $t$  value are different, then the final  $t$  value will certainly be negative and thus not a valid intersection.

Along similar lines,

$$t < t_{\max} = \begin{cases} \sum_i e_i z_i < t_{\max}(e_0 + e_1 + e_2) & \text{If } e_0 + e_1 + e_2 > 0 \\ \sum_i e_i z_i > t_{\max}(e_0 + e_1 + e_2) & \text{otherwise.} \end{cases}$$

*(Compute scaled hit distance to triangle and test against ray t range) ≡*

158

```

p0t.z *= Sz;
plt.z *= Sz;
p2t.z *= Sz;
Float tScaled = e0 * p0t.z + e1 * plt.z + e2 * p2t.z;
if (det < 0 && (tScaled >= 0 || tScaled < ray.tMax * det))
    return false;
else if (det > 0 && (tScaled <= 0 || tScaled > ray.tMax * det))
    return false;
```

Float 1062

We now know that there is a valid intersection and will go ahead and pay the cost of the floating-point division to compute actual barycentric coordinates as well as the actual  $t$  value for the intersection.

*(Compute barycentric coordinates and t value for triangle intersection)  $\equiv$*

```
Float invDet = 1 / det;
Float b0 = e0 * invDet;
Float b1 = e1 * invDet;
Float b2 = e2 * invDet;
Float t = tScaled * invDet;
```

158

In order to generate consistent tangent vectors over triangle meshes, it is necessary to compute the partial derivatives  $\partial p/\partial u$  and  $\partial p/\partial v$  using the parametric  $(u, v)$  values at the triangle vertices, if provided. Although the partial derivatives are the same at all points on the triangle, the implementation here recomputes them each time an intersection is found. Although this results in redundant computation, the storage savings for large triangle meshes can be significant.

A triangle can be described by the set of points

$$p_o + u \frac{\partial p}{\partial u} + v \frac{\partial p}{\partial v},$$

for some  $p_o$ , where  $u$  and  $v$  range over the parametric coordinates of the triangle. We also know the three vertex positions  $p_i$ ,  $i = 0, 1, 2$ , and the texture coordinates  $(u_i, v_i)$  at each vertex. From this it follows that the partial derivatives of  $p$  must satisfy

$$p_i = p_o + u_i \frac{\partial p}{\partial u} + v_i \frac{\partial p}{\partial v}.$$

In other words, there is a unique affine mapping from the 2D  $(u, v)$  space to points on the triangle (such a mapping exists even though the triangle is specified in 3D space because the triangle is planar). To compute expressions for  $\partial p/\partial u$  and  $\partial p/\partial v$ , we start by computing the differences  $p_0 - p_2$  and  $p_1 - p_2$ , giving the matrix equation

$$\begin{pmatrix} u_0 - u_2 & v_0 - v_2 \\ u_1 - u_2 & v_1 - v_2 \end{pmatrix} \begin{pmatrix} \partial p/\partial u \\ \partial p/\partial v \end{pmatrix} = \begin{pmatrix} p_0 - p_2 \\ p_1 - p_2 \end{pmatrix}.$$

Thus,

$$\begin{pmatrix} \partial p/\partial u \\ \partial p/\partial v \end{pmatrix} = \begin{pmatrix} u_0 - u_2 & v_0 - v_2 \\ u_1 - u_2 & v_1 - v_2 \end{pmatrix}^{-1} \begin{pmatrix} p_0 - p_2 \\ p_1 - p_2 \end{pmatrix}.$$

Inverting a  $2 \times 2$  matrix is straightforward. The inverse of the  $(u, v)$  differences matrix is

Float 1062

$$\frac{1}{(u_0 - u_2)(v_1 - v_2) - (v_0 - v_2)(u_1 - u_2)} \begin{pmatrix} v_1 - v_2 & -(v_0 - v_2) \\ -(u_1 - u_2) & u_0 - u_2 \end{pmatrix}.$$

```

⟨Compute triangle partial derivatives⟩ ≡ 157
Vector3f dpdu, dpdv;
Point2f uv[3];
GetUVs(uv);
⟨Compute deltas for triangle partial derivatives 164⟩
Float determinant = duv02[0] * d uv12[1] - d uv02[1] * d uv12[0];
if (determinant == 0) {
    ⟨Handle zero determinant for triangle partial derivative matrix 164⟩
} else {
    Float invdet = 1 / determinant;
    dpdu = (d uv12[1] * dp02 - d uv02[1] * dp12) * invdet;
    dpdv = (-d uv12[0] * dp02 + d uv02[0] * dp12) * invdet;
}

⟨Compute deltas for triangle partial derivatives⟩ ≡ 164
Vector2f d uv02 = uv[0] - uv[2], d uv12 = uv[1] - uv[2];
Vector3f dp02 = p0 - p2, dp12 = p1 - p2;

Finally, it is necessary to handle the case when the matrix is singular and therefore cannot be inverted. Note that this only happens when the user-supplied per-vertex parameterization values are degenerate. In this case, the Triangle just chooses an arbitrary coordinate system about the triangle's surface normal, making sure that it is orthonormal:

⟨Handle zero determinant for triangle partial derivative matrix⟩ ≡ 164
CoordinateSystem(Normalize(Cross(p2 - p0, p1 - p0)), &dpdu, &dpdv);

To compute the intersection point and the  $(u, v)$  parametric coordinates at the hit point, the barycentric interpolation formula is applied to the vertex positions and the  $(u, v)$  coordinates at the vertices. As we'll see in Section 3.9.4, this gives a more precise result for the intersection point than evaluating the parametric ray equation using  $t$ .

⟨Interpolate  $(u, v)$  parametric coordinates and hit point⟩ ≡ 157
Point3f pHit = b0 * p0 + b1 * p1 + b2 * p2;
Point2f uvHit = b0 * uv[0] + b1 * uv[1] + b2 * uv[2];

The utility routine GetUVs() returns the  $(u, v)$  coordinates for the three vertices of the triangle, either from the Triangle, if it has them, or returning default values if explicit  $(u, v)$  coordinates were not specified with the mesh.

⟨Triangle Private Methods⟩ ≡ 156
void GetUVs(Point2f uv[3]) const {
    if (mesh->uv) {
        uv[0] = mesh->uv[v[0]];
        uv[1] = mesh->uv[v[1]];
        uv[2] = mesh->uv[v[2]];
    } else {
        uv[0] = Point2f(0, 0);
        uv[1] = Point2f(1, 0);
        uv[2] = Point2f(1, 1);
    }
}

```

[CoordinateSystem\(\)](#) 67  
[Cross\(\)](#) 65  
[Float](#) 1062  
[Point2f](#) 68  
[Point3f](#) 68  
[Triangle](#) 156  
[Triangle::GetUVs\(\)](#) 164  
[TriangleMesh::uv](#) 155  
[Vector2f](#) 60  
[Vector3::Normalize\(\)](#) 66  
[Vector3f](#) 60

Before a successful intersection is reported, the intersection point is tested against an alpha mask texture, if one has been assigned to the shape. This texture can be thought of as a 1D function over the triangle's surface, where at any point where its value is zero, the intersection is ignored, effectively treating that point on the triangle as not being present. (Chapter 10 defines the texture interface and implementations in more detail.) Alpha masks can be helpful for representing objects like leaves: a leaf can be modeled as a single triangle, with an alpha mask “cutting out” the edges so that a leaf shape remains. This functionality is less often useful for other shapes, so pbrt only supports it for triangles.

```
<Test intersection against alpha texture, if present> ≡ 157
    if (testAlphaTexture && mesh->alphaMask) {
        SurfaceInteraction isectLocal(pHit, Vector3f(0,0,0), uvHit,
            Vector3f(0,0,0), dpdu, dpdv, Normal3f(0,0,0), Normal3f(0,0,0),
            ray.time, this);
        if (mesh->alphaMask->Evaluate(isectLocal) == 0)
            return false;
    }
```

Now we certainly have a valid intersection and can update the values pointed to by the pointers passed to the intersection routine. Unlike other shapes' implementations, the code that initializes the `SurfaceInteraction` structure here doesn't need to transform the partial derivatives to world space, since the triangle's vertices were already transformed to world space. Like the disk, the partial derivatives of the triangle's normal are also both  $(0, 0, 0)$ , since it is flat.

```
<Fill in SurfaceInteraction from triangle hit> ≡ 157
    *isect = SurfaceInteraction(pHit, pError, uvHit, -ray.d, dpdu, dpdv,
        Normal3f(0, 0, 0), Normal3f(0, 0, 0), ray.time, this);
<Override surface normal in isect for triangle 165>
    if (mesh->n || mesh->s) {
        <Initialize Triangle shading geometry 166>
    }
<Ensure correct orientation of the geometric normal 166>

Cross() 65
Interaction::n 116
Normal3::Normalize() 71
Normal3f 71
SurfaceInteraction 116
SurfaceInteraction::shading 118
SurfaceInteraction:::
    shading::n 118
Texture::Evaluate() 615
Triangle::mesh 156
TriangleMesh::alphaMask 155
TriangleMesh::n 155
TriangleMesh::s 155
Vector3f 60
```

The `SurfaceInteraction` constructor initializes the geometric normal `n` as the normalized cross product of `dpdu` and `dpdv`. This works well for most shapes, but in the case of triangle meshes it is preferable to rely on an initialization that does not depend on the underlying texture coordinates: it is fairly common to encounter meshes with bad parameterizations that do not preserve the orientation of the mesh, in which case the geometric normal would have an incorrect orientation.

We therefore initialize the geometric normal using the normalized cross product of the edge vectors `dp02` and `dp12`, which results in the same normal up to a potential sign difference that depends on the exact order of triangle vertices (also known as the triangle's *winding order*). 3D modeling packages generally try to ensure that triangles in a mesh have consistent winding orders, which makes this approach more robust.

```
<Override surface normal in isect for triangle> ≡ 165
    isect->n = isect->shading.n = Normal3f(Normalize(Cross(dp02, dp12)));
```

When interpolated normals are available, then we consider those to be the most authoritative source of orientation information. In this case, we flip the orientation of `isect->n` if the angle between it and the interpolated normal is greater than 180 degrees.

```
(Ensure correct orientation of the geometric normal) ≡ 165
if (mesh->n)
    isect->n = Faceforward(isect->n, isect->shading.n);
else if (reverseOrientation ^ transformSwapsHandedness)
    isect->n = isect->shading.n = -isect->n;
```

### 3.6.3 SHADING GEOMETRY

With Triangles, the user can provide normal vectors and tangent vectors at the vertices of the mesh that are interpolated to give normals and tangents at points on the faces of triangles. Shading geometry with interpolated normals can make otherwise faceted triangle meshes appear to be smoother than they geometrically are. If either shading normals or shading tangents have been provided, they are used to initialize the shading geometry in the SurfaceInteraction.

```
(Initialize Triangle shading geometry) ≡ 165
(Compute shading normal ns for triangle 166)
(Compute shading tangent ss for triangle 166)
(Compute shading bitangent ts for triangle and adjust ss 167)
(Compute  $\partial\mathbf{n}/\partial u$  and  $\partial\mathbf{n}/\partial v$  for triangle shading geometry)
isect->SetShadingGeometry(ss, ts, dndu, dndv, true);
```

Given the barycentric coordinates of the intersection point, it's straightforward to compute the shading normal by interpolating among the appropriate vertex normals, if present.

```
(Compute shading normal ns for triangle) ≡ 166
Normal3f ns;
if (mesh->n) ns = Normalize(b0 * mesh->n[v[0]] +
                           b1 * mesh->n[v[1]] +
                           b2 * mesh->n[v[2]]);

else
    ns = isect->n;
```

The shading tangent is computed similarly.

```
(Compute shading tangent ss for triangle) ≡ 166
Vector3f ss;
if (mesh->s) ss = Normalize(b0 * mesh->s[v[0]] +
                           b1 * mesh->s[v[1]] +
                           b2 * mesh->s[v[2]]);

else
    ss = Normalize(isect->dpdu);
```

The bitangent vector `ts` is found using the cross product of `ss` and `ns`, giving a vector orthogonal to the two of them. Next, `ss` is overwritten with the cross product of `ns` and `ts`; this ensures that the cross product of `ss` and `ts` gives `ns`. Thus, if per-vertex `n` and `s`

```
Faceforward() 72
Interaction::n 116
Normal3f 71
Shape::reverseOrientation 124
Shape:::transformSwapsHandedness 124
SurfaceInteraction 116
SurfaceInteraction:::SetShadingGeometry() 119
SurfaceInteraction::shading 118
SurfaceInteraction:::shading::n 118
Triangle::mesh 156
TriangleMesh::n 155
TriangleMesh::s 155
Vector3::Normalize() 66
Vector3f 60
```

values are provided and if the interpolated  $n$  and  $s$  values aren't perfectly orthogonal,  $n$  will be preserved and  $s$  will be modified so that the coordinate system is orthogonal.

*(Compute shading bitangent ts for triangle and adjust ss) ≡*

```
Vector3f ts = Cross(ss, ns);
if (ts.LengthSquared() > 0.f) {
    ts = Normalize(ts);
    ss = Cross(ts, ns);
}
else
    CoordinateSystem((Vector3f)ns, &ss, &ts);
```

166

The code to compute the partial derivatives  $\partial n/\partial u$  and  $\partial n/\partial v$  of the shading normal is almost identical to the code to compute the partial derivatives  $\partial p/\partial u$  and  $\partial p/\partial v$ . Therefore, it has been elided from the text here.

### 3.6.4 SURFACE AREA

Using the fact that the area of a parallelogram is given by the length of the cross product of the two vectors along its sides, the `Area()` method computes the triangle area as half the area of the parallelogram formed by two of its edge vectors (Figure 3.13).

*(Triangle Method Definitions) +≡*

```
Float Triangle::Area() const {
    (Get triangle vertices in p0, p1, and p2 157)
    return 0.5 * Cross(p1 - p0, p2 - p0).Length();
}
```

## ★ 3.7 CURVES

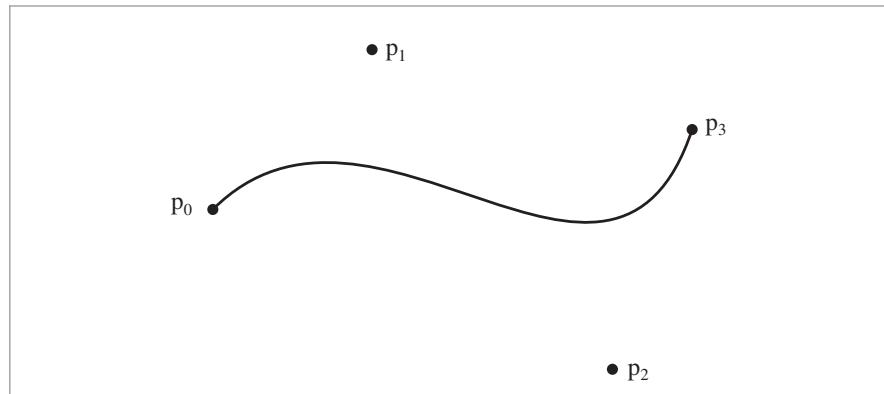
While triangles can be used to represent thin shapes for modeling fine geometry like hair, fur, or fields of grass, it's worthwhile to have a specialized `Shape` in order to more efficiently render these sorts of objects, since many individual instances of them are often present. The `Curve` shape, introduced in this section, represents thin geometry modeled with cubic Bézier splines, which are defined by four control points,  $p_0$ ,  $p_1$ ,  $p_2$ , and  $p_3$ . The Bézier spline passes through the first and last control points; intermediate points are given by the polynomial

CoordinateSystem() 67  
 Cross() 65  
 Curve 168  
 Float 1062  
 Triangle 156  
 Vector3f::Length() 65  
 Vector3f::LengthSquared() 65  
 Vector3f 60

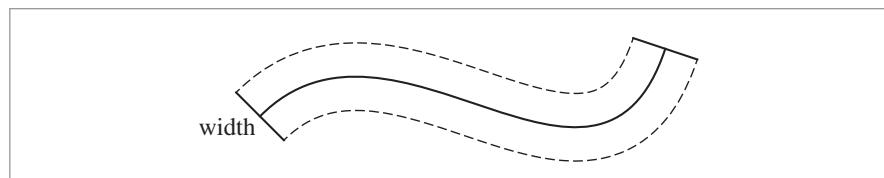
$$p(u) = (1-u)^3 p_0 + 3(1-u)^2 u p_1 + 3(1-u)u^2 p_2 + u^3 p_3. \quad [3.3]$$

(See Figure 3.15.) Given a curve specified in another cubic basis, such as a Hermite spline, it's easy enough to convert to Bézier basis, so the implementation here leaves that burden on the user. This functionality could be easily added if it was frequently needed.

The `Curve` shape is defined by a 1D Bézier curve along with a width that is linearly interpolated from starting and ending widths along its extent. Together, these define a flat



**Figure 3.15:** A cubic Bézier curve is defined by four control points,  $p_i$ . The curve  $p(u)$ , defined in Equation (3.3), passes through the first and last control points at  $u = 0$  and  $u = 1$ , respectively.



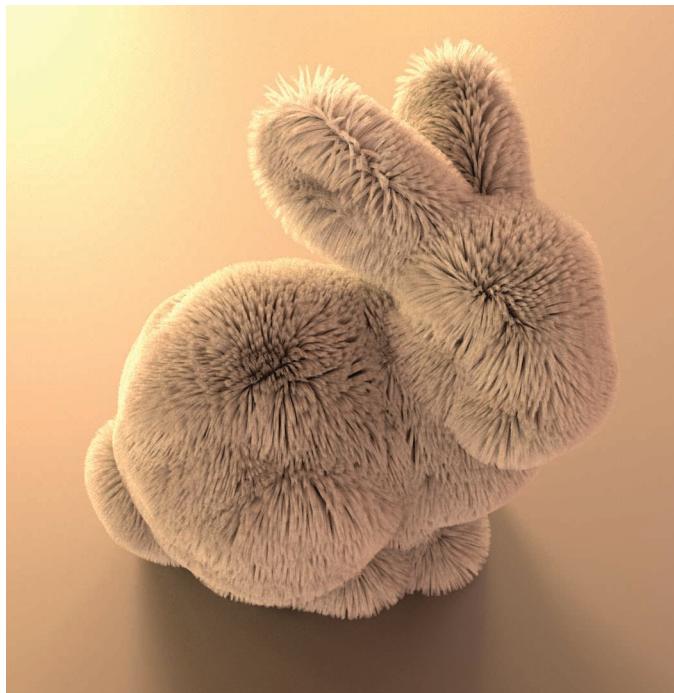
**Figure 3.16: Basic Geometry of the Curve Shape.** A 1D Bézier curve is offset by half of the specified width in both the directions orthogonal to the curve at each point along it. The resulting area represents the curve's surface.

2D surface (Figure 3.16).<sup>6</sup> It’s possible to directly intersect rays with this representation without tessellating it, which in turn makes it possible to efficiently render smooth curves without using too much storage. Figure 3.17 shows a bunny model with fur modeled with over one million Curves.

*(Curve Declarations) ≡*

```
class Curve : public Shape {
public:
    (Curve Public Methods 170)
private:
    (Curve Private Methods)
    (Curve Private Data 170)
};
```

<sup>6</sup> Note the abuse of terminology: while a curve is a 1D mathematical entity, a Curve shape represents a 2D surface. In the following, we’ll generally refer to the Shape as a curve. The 1D entity will be distinguished by the name “Bézier curve” when the distinction wouldn’t otherwise be clear.



**Figure 3.17: Furry Bunny.** Bunny model with over one million Curve shapes used to model fur. Here, we've used unrealistically long curves to better show off the Curve's capabilities.

There are three types of curves that the Curve shape can represent, shown in Figure 3.18.

- Flat: Curves with this representation are always oriented to face the ray being intersected with them; they are useful for modeling fine swept cylindrical shapes like hair or fur.
- Cylinder: For curves that span a few pixels on the screen (like spaghetti seen from not too far away), the Curve shape can compute a shading normal that makes the curve appear to actually be a cylinder.
- Ribbon: This variant is useful for modeling shapes that don't actually have a cylindrical cross section (such as a blade of grass).

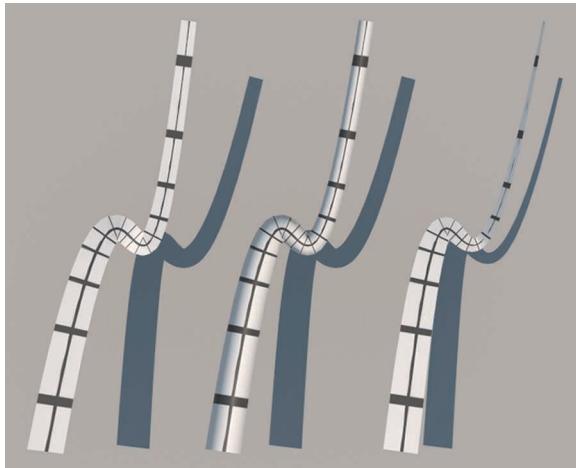
The `CurveType` enumerant records which of them a given `Curve` instance models.

The flat and cylinder curve variants are intended to be used as convenient approximations of deformed cylinders. It should be noted that intersections found with respect to them do not correspond to a physically realizable 3D shape, which can potentially lead to minor inconsistencies when taking a scene with true cylinders as a reference.

```
(CurveType Declarations) ≡
enum class CurveType { Flat, Cylinder, Ribbon };
```

Curve 168

Given a curve specified in a pbrt scene description file, it can be worthwhile to split it into a few segments, each covering part of the  $u$  parametric range of the curve. (One reason



**Figure 3.18: The Three Types of Curves That the Curve Shape Can Represent.** On the left is a flat curve that is always oriented to be perpendicular to a ray approaching it. The middle is a variant of this curve where the shading normal is set so that the curve appears to be cylindrical. On the right is a ribbon, which has a fixed orientation at its starting and ending points; intermediate orientations are smoothly interpolated between them.

for doing so is that axis-aligned bounding boxes don't tightly bound wiggly curves, but subdividing Bézier splines makes them less wiggly—the *variation diminishing property* of polynomial splines.) Therefore, the `Curve` constructor takes both a parametric range of  $u$  values,  $[u_{\min}, u_{\max}]$ , as well as a pointer to a `CurveCommon` structure, which stores the control points and other information about the curve that is shared across curve segments. In this way, the memory footprint for individual curve segments is minimized, which makes it easier to keep many of them in memory.

*(Curve Public Methods) ≡*

168

```
Curve(const Transform *ObjectToWorld, const Transform *WorldToObject,
      bool reverseOrientation, const std::shared_ptr<CurveCommon> &common,
      Float uMin, Float uMax)
: Shape(ObjectToWorld, WorldToObject, reverseOrientation),
  common(common), uMin(uMin), uMax(uMax) { }
```

*(Curve Private Data) ≡*

168

```
const std::shared_ptr<CurveCommon> common;
const Float uMin, uMax;
```

The `CurveCommon` constructor mostly just initializes member variables with values passed into it for the control points, the curve width, etc. The control points provided to it should be in the curve's object space.

For Ribbon curves, `CurveCommon` stores a surface normal to orient the curve at each endpoint. The constructor precomputes the angle between the two normal vectors and one over the sine of this angle; these values will be useful when computing the orientation of the curve at arbitrary points along its extent.

`Curve` 168

`CurveCommon` 171

`Float` 1062

`Shape` 123

`Transform` 83

*(Curve Method Definitions)* ≡

```
CurveCommon::CurveCommon(const Point3f c[4], Float width0, Float width1,
    CurveType type, const Normal3f *norm)
: type(type), cpObj{c[0], c[1], c[2], c[3]},
  width{width0, width1} {
if (norm) {
    n[0] = Normalize(norm[0]);
    n[1] = Normalize(norm[1]);
    normalAngle = std::acos(Clamp(Dot(n[0], n[1]), 0, 1));
    invSinNormalAngle = 1 / std::sin(normalAngle);
}
}
```

*(CurveCommon Declarations)* ≡

```
struct CurveCommon {
    const CurveType type;
    const Point3f cpObj[4];
    const Float width[2];
    Normal3f n[2];
    Float normalAngle, invSinNormalAngle;
};
```

Bounding boxes of Curves can be computed by taking advantage of the *convex hull property*, a property of Bézier curves that says that they must lie within the convex hull of their control points. Therefore, the bounding box of the control points gives a conservative bound of the underlying curve. The `ObjectBound()` method first computes a bounding box of the control points of the 1D Bézier segment to bound the spline along the center of the curve. These bounds are then expanded by half the maximum width the curve takes on over its parametric extent to get the 3D bounds of the Shape that the Curve represents.

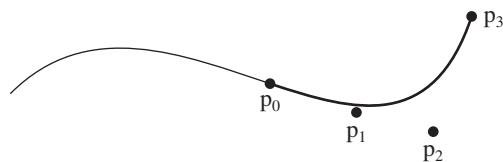
*(Curve Method Definitions)* +≡

```
Bounds3f Curve::ObjectBound() const {
    (Compute object-space control points for curve segment, cpObj 172)
    Bounds3f b = Union(Bounds3f(cpObj[0], cpObj[1]),
        Bounds3f(cpObj[2], cpObj[3]));
    Float width[2] = { Lerp(uMin, common->width[0], common->width[1]),
        Lerp(uMax, common->width[0], common->width[1]) };
    return Expand(b, std::max(width[0], width[1]) * 0.5f);
}
```

Bounds3f 76  
 Curve 168  
 Curve::common 170  
 Curve::uMax 170  
 Curve::uMin 170  
 CurveCommon 171  
 CurveCommon::  
 invSinNormalAngle 171  
 CurveCommon::n 171  
 CurveCommon::normalAngle 171  
 CurveCommon::width 171  
 CurveType 169  
 Float 1062  
 Normal3f 71  
 Point3f 68

The `CurveCommon` class stores the control points for the full curve, but `Curve` instances generally need the four control points that represent the Bézier curve for its  $u$  extent. These control points are computed using a technique called *blossoming*. The blossom  $p(u_0, u_1, u_2)$  of a cubic Bézier spline is defined by three stages of linear interpolation, starting with the original control points:

$$\begin{aligned} a_i &= (1 - u_0)p_i + u_0 p_{i+1} \quad i \in [0, 1, 2] \\ b_j &= (1 - u_1)a_j + u_1 a_{j+1} \quad j \in [0, 1] \\ c &= (1 - u_2)b_0 + u_2 b_1 \end{aligned} \tag{3.4}$$



**Figure 3.19: Blossoming to Find Control Points for a Segment of a Bézier Curve.** The four blossoms in Equation (3.5) give the control points for the curve from  $u_{\min}$  to  $u_{\max}$ . Blossoming provides an elegant method to compute the Bézier control points of the curve that represent a subset of the overall curve.

The blossom  $p(u, u, u)$  gives the curve's value at position  $u$ . (To verify this for yourself, expand Equation (3.4) using  $u_i = u$ , simplify, and compare to Equation (3.3).)

`BlossomBezier()` implements this computation.

*(Curve Utility Functions)* ≡

```
static Point3f BlossomBezier(const Point3f p[4], Float u0, Float u1,
    Float u2) {
    Point3f a[3] = { Lerp(u0, p[0], p[1]),
        Lerp(u0, p[1], p[2]),
        Lerp(u0, p[2], p[3]) };
    Point3f b[2] = { Lerp(u1, a[0], a[1]), Lerp(u1, a[1], a[2]) };
    return Lerp(u2, b[0], b[1]);
}
```

The four control points for the curve segment over the range  $u_{\min}$  to  $u_{\max}$  are given by the blossoms:

$$\begin{aligned} P_0 &= p(u_{\min}, u_{\min}, u_{\min}) \\ P_1 &= p(u_{\min}, u_{\min}, u_{\max}) \\ P_2 &= p(u_{\min}, u_{\max}, u_{\max}) \\ P_3 &= p(u_{\max}, u_{\max}, u_{\max}) \end{aligned} \quad (3.5)$$

(Figure 3.19).

Given this machinery, it's straightforward to compute the four control points for the curve segment that a `Curve` is responsible for.

*(Compute object-space control points for curve segment, cp0bj)* ≡

171, 173

```
Point3f cp0bj[4];
cp0bj[0] = BlossomBezier(common->cp0bj, uMin, uMin, uMin);
cp0bj[1] = BlossomBezier(common->cp0bj, uMin, uMin, uMax);
cp0bj[2] = BlossomBezier(common->cp0bj, uMin, uMax, uMax);
cp0bj[3] = BlossomBezier(common->cp0bj, uMax, uMax, uMax);
```

`BlossomBezier()` 172

`Curve::common` 170

`Curve::uMax` 170

`Curve::uMin` 170

`CurveCommon::cp0bj` 171

`Float` 1062

`Lerp()` 1079

`Point3f` 68

The Curve intersection algorithm is based on discarding curve segments as soon as it can be determined that the ray definitely doesn't intersect them and otherwise recursively splitting the curve in half to create two smaller segments that are then tested. Eventually, the curve is linearly approximated for an efficient intersection test. After some preparation, the recursiveIntersect() call starts this process with the full segment that the Curve represents.

```
<Curve Method Definitions> +≡
    bool Curve::Intersect(const Ray &r, Float *tHit,
        SurfaceInteraction *isect, bool testAlphaTexture) const {
        (Transform Ray to object space 134)
        (Compute object-space control points for curve segment, cpObj 172)
        (Project curve control points to plane perpendicular to ray 173)
        (Compute refinement depth for curve, maxDepth)
        return recursiveIntersect(ray, tHit, isect, cp, Inverse(objectToRay),
            uMin, uMax, maxDepth);
    }
```

Like the ray–triangle intersection algorithm from Section 3.6.2, the ray–curve intersection test is based on transforming the curve to a coordinate system with the ray's origin at the origin of the coordinate system and the ray's direction aligned to be along the  $+z$  axis. Performing this transformation at the start greatly reduces the number of operations that must be performed for intersection tests.

For the Curve shape, we'll need an explicit representation of the transformation, so the LookAt() function is used to generate it here. The origin is the ray's origin, the “look at” point is a point offset from the origin along the ray's direction, and the “up” direction is an arbitrary direction orthogonal to the ray direction.

```
CoordinateSystem() 67
Curve::recursiveIntersect() 174
Curve::uMax 170
Curve::uMin 170
Float 1062
Inverse() 1081
LookAt() 92
Point3f 68
Ray 73
Ray::d 73
Ray::o 73
SurfaceInteraction 116
Transform 83
Vector3f 60
```

*<Project curve control points to plane perpendicular to ray> ≡*

```
    Vector3f dx, dy;
    CoordinateSystem(ray.d, &dx, &dy);
    Transform objectToRay = LookAt(ray.o, ray.o + ray.d, dx);
    Point3f cp[4] = { objectToRay(cpObj[0]), objectToRay(cpObj[1]),
        objectToRay(cpObj[2]), objectToRay(cpObj[3]) };
```

173

The maximum number of times to subdivide the curve is computed so that the maximum distance from the eventual linearized curve at the finest refinement level is bounded to be less than a small fixed distance. We won't go into the details of this computation, which is implemented in the fragment *(Compute refinement depth for curve, maxDepth)*.

The recursiveIntersect() method then tests whether the given ray intersects the given curve segment over the given parametric range  $[u_0, u_1]$ .

```
(Curve Method Definitions) +≡
bool Curve::recursiveIntersect(const Ray &ray, Float *tHit,
    SurfaceInteraction *isect, const Point3f cp[4],
    const Transform &rayToObject, Float u0, Float u1,
    int depth) const {
    (Try to cull curve segment versus ray 174)
    if (depth > 0) {
        (Split curve segment into sub-segments and test for intersection 175)
    } else {
        (Intersect ray with curve segment 176)
    }
}
```

The method starts by checking to see if the ray intersects the curve segment's bounding box; if it doesn't, no intersection is possible and it can return immediately.

*(Try to cull curve segment versus ray) ≡* 174

*(Compute bounding box of curve segment, curveBounds 174)*  
*(Compute bounding box of ray, rayBounds 174)*  
if (Overlaps(curveBounds, rayBounds) == false)  
 return false;

Along the lines of the implementation in `Curve::ObjectBound()`, a conservative bounding box for the segment can be found by taking the bounds of the curve's control points and expanding by half of the maximum width of the curve over the  $u$  range being considered.

*(Compute bounding box of curve segment, curveBounds) ≡* 174

```
Bounds3f curveBounds =
    Union(Bounds3f(cp[0], cp[1]), Bounds3f(cp[2], cp[3]));
Float maxWidth = std::max(Lerp(u0, common->width[0], common->width[1]),
                           Lerp(u1, common->width[0], common->width[1]));
curveBounds = Expand(curveBounds, 0.5 * maxWidth);
```

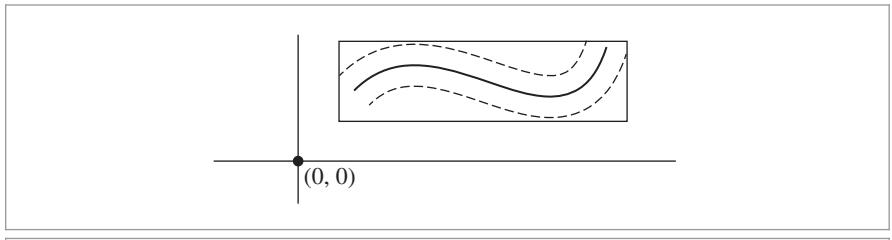
Because the ray's origin is at  $(0, 0, 0)$  and its direction is aligned with the  $+z$  axis in the intersection space, its bounding box only includes the origin in  $x$  and  $y$  (Figure 3.20); its  $z$  extent is given by the  $z$  range that its parametric extent covers.

*(Compute bounding box of ray, rayBounds) ≡* 174

```
Float rayLength = ray.d.Length();
Float zMax = rayLength * ray.tMax;
Bounds3f rayBounds(Point3f(0, 0, 0), Point3f(0, 0, zMax));
```

If the ray does intersect the curve's bounding box and the recursive splitting hasn't bottomed out, then the curve is split in half along the parametric  $u$  range. `SubdivideBezier()` computes seven control points: the first four correspond to the control points for the first half of the split curve, and the last four (starting with the last control point of the first half) correspond to the control points for the second half. Two calls to `recursiveIntersect()` test the two sub-segments.

Bounds3::Expand() 79  
Bounds3::Overlaps() 79  
Bounds3::Union() 78  
Bounds3f 76  
Curve::ObjectBound() 171  
Float 1062  
Lerp() 1079  
Point3f 68  
Ray 73  
Ray::tMax 73  
SurfaceInteraction 116  
Transform 83  
Vector3::Length() 65



**Figure 3.20: Ray-Curve Bounds Test.** In the ray coordinate system, the ray's origin is at  $(0, 0, 0)$  and its direction is aligned with the  $+z$  axis. Therefore, if the 2D point  $(x, y) = (0, 0)$  is outside the  $xy$  bounding box of the curve segment, then it's impossible that the ray intersects the curve.

*{Split curve segment into sub-segments and test for intersection}  $\equiv$*

174

```
Float uMid = 0.5f * (u0 + u1);
Point3f cpSplit[7];
SubdivideBezier(cp, cpSplit);
return (recursiveIntersect(ray, tHit, isect, &cpSplit[0], rayToObject,
                           u0, uMid, depth - 1) ||
        recursiveIntersect(ray, tHit, isect, &cpSplit[3], rayToObject,
                           uMid, u1, depth - 1));
```

While we could use the `BlossomBezier()` function to compute the control points of the subdivided curves, they can be more efficiently computed by taking advantage of the fact that we're always splitting the curve exactly in the middle of its parametric extent. This computation is implemented in the `SubdivideBezier()` function; the seven control points it computes correspond to using  $(0, 0, 0)$ ,  $(0, 0, 1/2)$ ,  $(0, 1/2, 1/2)$ ,  $(1/2, 1/2, 1/2)$ ,  $(1/2, 1/2, 1)$ ,  $(1/2, 1, 1)$ , and  $(1, 1, 1)$  as blossoms in Equation (3.4).

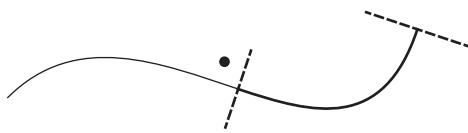
*{Curve Utility Functions}  $+\equiv$*

```
inline void SubdivideBezier(const Point3f cp[4], Point3f cpSplit[7]) {
    cpSplit[0] = cp[0];
    cpSplit[1] = (cp[0] + cp[1]) / 2;
    cpSplit[2] = (cp[0] + 2 * cp[1] + cp[2]) / 4;
    cpSplit[3] = (cp[0] + 3 * cp[1] + 3 * cp[2] + cp[3]) / 8;
    cpSplit[4] = (cp[1] + 2 * cp[2] + cp[3]) / 4;
    cpSplit[5] = (cp[2] + cp[3]) / 2;
    cpSplit[6] = cp[3];
```

BlossomBezier() 172  
Curve::recursiveIntersect()  
174

```
Float 1062
Point3f 68
SubdivideBezier() 175
```

After a number of subdivisions, an intersection test is performed. Parts of this test are made more efficient by using a linear approximation of the curve; the variation diminishing property allows us to make this approximation without introducing too much error.



**Figure 3.21: Curve Segment Boundaries.** The intersection test for a segment of a larger curve computes edge functions for the lines that are perpendicular to the segment endpoints (dashed lines). If a potential intersection point (solid dot) is on the other side of edge than the segment, it is rejected; another curve segment (if present on that side) should account for this intersection instead.

*(Intersect ray with curve segment) ≡*

174

*(Test ray against segment endpoint boundaries 176)*

*(Compute line w that gives minimum distance to sample point 178)*

*(Compute u coordinate of curve intersection point and hitWidth 179)*

*(Test intersection point against curve width 179)*

*(Compute v coordinate of curve intersection point 180)*

*(Compute hit t and partial derivatives for curve intersection 180)*

*return true;*

It's important that the intersection test only accept intersections that are on the Curve's surface for the  $u$  segment currently under consideration. Therefore, the first step of the intersection test is to compute edge functions for lines perpendicular to the curve starting point and ending point and to classify the potential intersection point against them (Figure 3.21).

*(Test ray against segment endpoint boundaries) ≡*

176

*(Test sample point against tangent perpendicular at curve start 177)*

*(Test sample point against tangent perpendicular at curve end)*

Projecting the curve control points into the ray coordinate system makes this test more efficient for two reasons. First, because the ray's direction is oriented with the  $+z$  axis, the problem is reduced to a 2D test in  $x$  and  $y$ . Second, because the ray origin is at the origin of the coordinate system, the point we need to classify is  $(0, 0)$ , which simplifies evaluating the edge function, just like the ray-triangle intersection test.

Edge functions were introduced for ray-triangle intersection tests in Equation (3.1); see also Figure 3.14. To define the edge function, we need any two points on the line perpendicular to the curve going through starting point. The first control point,  $p_0$ , is a fine choice for the first point. For the second one, we'll compute the vector perpendicular to the curve's tangent and add that offset to the control point.

Differentiation of Equation (3.3) shows that the tangent to the curve at the first control point  $p_0$  is  $3(p_1 - p_0)$ . The scaling factor doesn't matter here, so we'll use  $t = p_1 - p_0$  here. Computing the vector perpendicular to the tangent is easy in 2D: it's just necessary to swap the  $x$  and  $y$  coordinates and negate one of them. (To see why this works, consider the dot product  $(x, y) \cdot (y, -x) = xy + -yx = 0$ . Because the cosine of the angle between the two vectors is zero, they must be perpendicular.) Thus, the second point on

the edge is

$$p_0 + (p_{1y} - p_{0y}, -(p_{1x} - p_{0x})) = p_0 + (p_{1y} - p_{0y}, p_{0x} - p_{1x}).$$

Substituting these two points into the definition of the edge function, Equation (3.1), and simplifying gives

$$e(p) = (p_{1y} - p_{0y})(p_y - p_{0y}) - (p_x - p_{0x})(p_{0x} - p_{1x}).$$

Finally, substituting  $p = (0, 0)$  gives the final expression to test:

$$e((0, 0)) = (p_{1y} - p_{0y})(-p_{0y}) + p_{0x}(p_{0x} - p_{1x}).$$

*(Test sample point against tangent perpendicular at curve start) ≡*

```
Float edge = (cp[1].y - cp[0].y) * -cp[0].y +
            cp[0].x * (cp[0].x - cp[1].x);
if (edge < 0)
    return false;
```

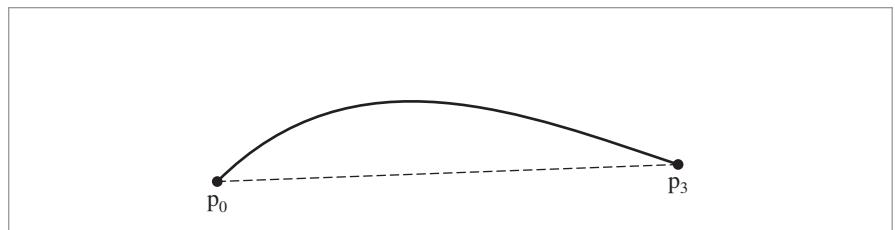
176

The *(Test sample point against tangent perpendicular at curve end)* fragment, not included here, does the corresponding test at the end of the curve.

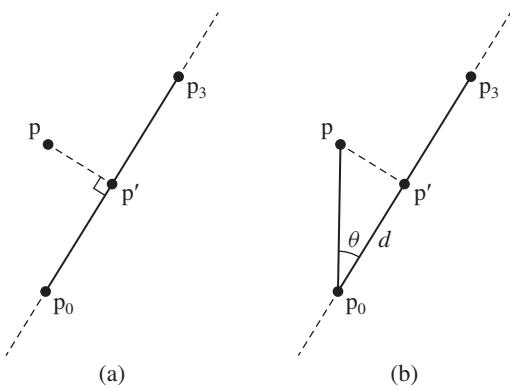
The next part of the test is to determine the  $u$  value along the curve segment where the point  $(0, 0)$  is closest to the curve. This will be the intersection point, if it's no farther than the curve's width away from the center at that point. Determining this distance for a cubic Bézier curve is not efficient, so instead this intersection approach approximates the curve with a linear segment to compute this  $u$  value.

We'll linearly approximate the Bézier curve with a line segment from its starting point  $p_0$  to its end point  $p_3$  that is parameterized by  $w$ . In this case, the position is  $p_0$  at  $w = 0$  and  $p_3$  at  $w = 1$  (Figure 3.22). Our task now is to compute the value of  $w$  along the line corresponding to the point on the line  $p'$  that is closest to the point  $p$ . The key insight to apply is that at  $p'$ , the vector from the corresponding point on the line to  $p$  will be perpendicular to the line (Figure 3.23 (a)).

Equation (2.1) gives us a relationship between the dot product of two vectors, their lengths, and the cosine of the angle between them. In particular, it shows us how to



**Figure 3.22: Approximation of a Cubic Bézier Curve with a Linear Segment.** For part of the ray–curve intersection test, we approximate the Bézier with a linear segment (dashed line) passing through its starting and ending points. (In practice, after being subdivided, the curve will be already nearly linear, so the error is less than this figure suggests.)



**Figure 3.23:** (a) Given an infinite line and a point  $p$ , then the vector from the point to the closest point on the line,  $p'$ , is perpendicular to the line. (b) Because this vector is perpendicular, we can compute the distance from the first point of the line to the point of closest approach,  $p'$  as  $d = \|p - p_0\| \cos \theta$ .

compute the cosine of the angle between the vector from  $p_0$  to  $p$  and the vector from  $p_0$  to  $p_3$ :

$$\cos \theta = \frac{(p - p_0) \cdot (p_3 - p_0)}{\|p - p_0\| \|p_3 - p_0\|}.$$

Because the vector from  $p'$  to  $p$  is perpendicular to the line (Figure 3.23(b)), then we can compute the distance along the line from  $p_0$  to  $p'$  as

$$d = \|p - p_0\| \cos \theta = \frac{(p - p_0) \cdot (p_3 - p_0)}{\|p_3 - p_0\|}.$$

Finally, the parametric offset  $w$  along the line is the ratio of  $d$  to the line's length,

$$w = \frac{d}{\|p_3 - p_0\|} = \frac{(p - p_0) \cdot (p_3 - p_0)}{\|p_3 - p_0\|^2}.$$

The computation of the value of  $w$  is in turn slightly simplified from the fact that  $p = (0, 0)$  in the intersection coordinate system.

*(Compute line  $w$  that gives minimum distance to sample point)  $\equiv$*

```

Vector2f segmentDirection = Point2f(cp[3]) - Point2f(cp[0]);
Float denom = segmentDirection.LengthSquared();
if (denom == 0)
    return false;
Float w = Dot(-Vector2f(cp[0]), segmentDirection) / denom;
```

176

Dot() 63

Float 1062

Point2f 68

Vector2f 60

The parametric  $u$  coordinate of the (presumed) closest point on the Bézier curve to the candidate intersection point is computed by linearly interpolating along the  $u$  range of the segment. Given this  $u$  value, the width of the curve at that point can be computed.

*(Compute u coordinate of curve intersection point and hitWidth) ≡*

```
Float u = Clamp(Lerp(w, u0, u1), u0, u1);
Float hitWidth = Lerp(u, common->width[0], common->width[1]);
Normal3f nHit;
if (common->type == CurveType::Ribbon) {
    (Scale hitWidth based on ribbon orientation 179)
}
```

176

For Ribbon curves, the curve is not always oriented to face the ray. Rather, its orientation is interpolated between two surface normals given at each endpoint. Here, spherical linear interpolation is used to interpolate the normal at  $u$  (recall Section 2.9.2). The curve's width is then scaled by the cosine of the angle between the normalized ray direction and the ribbon's orientation so that it reflects the visible width of the curve from the given direction.

*(Scale hitWidth based on ribbon orientation) ≡*

```
Float sin0 = std::sin((1 - u) * common->normalAngle) *
    common->invSinNormalAngle;
Float sin1 = std::sin(u * common->normalAngle) *
    common->invSinNormalAngle;
nHit = sin0 * common->n[0] + sin1 * common->n[1];
hitWidth *= AbsDot(nHit, ray.d) / rayLength;
```

179

To finally classify the potential intersection as a hit or miss, the Bézier curve must still be evaluated at  $u$  using the `EvalBezier()` function. (Because the control points `cp` represent the curve segment currently under consideration, it's important to use  $w$  rather than  $u$  in the function call, however, since  $w$  is in the range  $[0, 1]$ .) The derivative of the curve at this point will be useful shortly, so it's recorded now.

We'd like to test whether the distance from  $p$  to this point on the curve  $pc$  is less than half the curve's width. Because  $p = (0, 0)$ , we can equivalently test whether the distance from  $pc$  to the origin is less than half the width or, equivalently, whether the squared distance is less than one quarter the width squared. If this test passes, the last thing to check is if the intersection point is in the ray's parametric  $t$  range.

AbsDot() 64  
 Clamp() 1062  
 Curve::common 170  
 CurveCommon:::  
 invSinNormalAngle  
 171  
 CurveCommon::normalAngle 171  
 CurveCommon::type 171  
 CurveCommon::width 171  
 CurveType::Ribbon 169  
 EvalBezier() 180  
 Float 1062  
 Lerp() 1079  
 Normal3f 71  
 Point3f 68  
 Vector3f 60

*(Test intersection point against curve width) ≡*

```
Vector3f dpcdw;
Point3f pc = EvalBezier(cp, Clamp(w, 0, 1), &dpcdw);
Float ptCurveDist2 = pc.x * pc.x + pc.y * pc.y;
if (ptCurveDist2 > hitWidth * hitWidth * .25)
    return false;
if (pc.z < 0 || pc.z > zMax)
    return false;
```

176

`EvalBezier()` computes the blossom  $p(u, u, u)$  to evaluate a point on a Bézier spline. It optionally also returns the derivative of the curve at the point.

```
(Curve Utility Functions) +≡
static Point3f EvalBezier(const Point3f cp[4], Float u,
    Vector3f *deriv = nullptr) {
    Point3f cp1[3] = { Lerp(u, cp[0], cp[1]), Lerp(u, cp[1], cp[2]),
        Lerp(u, cp[2], cp[3]) };
    Point3f cp2[2] = { Lerp(u, cp1[0], cp1[1]), Lerp(u, cp1[1], cp1[2]) };
    if (deriv)
        *deriv = (Float)3 * (cp2[1] - cp2[0]);
    return Lerp(u, cp2[0], cp2[1]);
}
```

If the earlier tests have all passed, we have found a valid intersection, and the  $v$  coordinate of the intersection point can now be computed. The curve's  $v$  coordinate ranges from 0 to 1, taking on the value 0.5 at the center of the curve; here, we classify the intersection point,  $(0, 0)$ , with respect to an edge function going through the point on the curve  $pc$  and a point along its derivative to determine which side of the center the intersection point is on and in turn how to compute  $v$ .

(Compute  $v$  coordinate of curve intersection point) ≡ 176

```
Float ptCurveDist = std::sqrt(ptCurveDist2);
Float edgeFunc = dpcdw.x * -pc.y + pc.x * dpcdw.y;
Float v = (edgeFunc > 0) ? 0.5f + ptCurveDist / hitWidth :
    0.5f - ptCurveDist / hitWidth;
```

Finally, the partial derivatives are computed, and the `SurfaceInteraction` for the intersection can be initialized.

(Compute hit  $t$  and partial derivatives for curve intersection) ≡ 176

```
if (tHit != nullptr) {
    *tHit = pc.z / rayLength;
    (Compute error bounds for curve intersection 227)
    (Compute  $\partial p/\partial u$  and  $\partial p/\partial v$  for curve intersection 180)
    *isect = (*ObjectToWorld)(SurfaceInteraction(
        ray(pc.z), pError, Point2f(u, v), -ray.d, dpdu, dpdv,
        Normal3f(0, 0, 0), Normal3f(0, 0, 0), ray.time, this));
}
```

The partial derivative  $\partial p/\partial u$  comes directly from the derivative of the underlying Bézier curve. The second partial derivative,  $\partial p/\partial v$ , is computed in different ways based on the type of the curve. For ribbons, we have  $\partial p/\partial u$  and the surface normal, and so  $\partial p/\partial v$  must be the vector such that  $\partial p/\partial u \times \partial p/\partial v = n$  and has length equal to the curve's width.

(Compute  $\partial p/\partial u$  and  $\partial p/\partial v$  for curve intersection) ≡ 180

```
Vector3f dpdu, dpdv;
EvalBezier(common->cpObj, u, &dpdu);
EvalBezier(common->cpObj, v, &dpdv);
```

Cross() 65  
 Curve::common 170  
 CurveCommon::type 171  
 CurveType::Ribbon 169  
 EvalBezier() 180  
 Float 1062  
 Lerp() 1079  
 Normal3f 71  
 Point2f 68  
 Point3f 68  
 SurfaceInteraction 116  
 Vector3f::Normalize() 66  
 Vector3f 60

```

if (common->type == CurveType::Ribbon)
    dpdv = Normalize(Cross(nHit, dpdu)) * hitWidth;
else {
    <Compute curve ∂p/∂v for flat and cylinder curves 181>
}

```

For flat and cylinder curves, we transform  $\partial p/\partial u$  to the intersection coordinate system. For flat curves, we know that  $\partial p/\partial v$  lies in the  $xy$  plane, is perpendicular to  $\partial p/\partial u$ , and has length equal to `hitWidth`. We can find the 2D perpendicular vector using the same approach as was used earlier for the perpendicular curve segment boundary edges.

```

<Compute curve ∂p/∂v for flat and cylinder curves> ≡ 180
Vector3f dpduPlane = (Inverse(rayToObject))(dpdu);
Vector3f dpdvPlane = Normalize(Vector3f(-dpduPlane.y, dpduPlane.x, 0)) *
    hitWidth;
if (common->type == CurveType::Cylinder) {
    <Rotate dpdvPlane to give cylindrical appearance 181>
}
dpdv = rayToObject(dpdvPlane);

```

The  $\partial p/\partial v$  vector for cylinder curves is rotated around the `dpduPlane` axis so that its appearance resembles a cylindrical cross-section.

```

<Rotate dpdvPlane to give cylindrical appearance> ≡ 181
Float theta = Lerp(v, -90., 90.);
Transform rot = Rotate(-theta, dpduPlane);
dpdvPlane = rot(dpdvPlane);

```

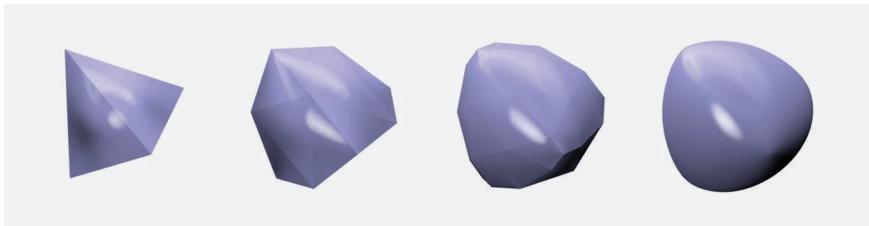
The `Curve::Area()` method, not included here, first approximates the curve length by the length of its control hull. It then multiplies this length by the average curve width over its extent to approximate the overall surface area.

## \*3.8 SUBDIVISION SURFACES

**Curve::common** 170  
**CurveCommon::type** 171  
**CurveType::Cylinder** 169  
**Cylinder** 142  
**Float** 1062  
**Inverse()** 1081  
**Lerp()** 1079  
**Rotate()** 91  
**Transform** 83  
**Vector3::Normalize()** 66  
**Vector3f** 60

The last shape representation that we'll define in this chapter implements *subdivision surfaces*, a representation that is particularly well suited to describing complex smooth shapes. The subdivision surface for a particular mesh is defined by repeatedly subdividing the faces of the mesh into smaller faces and then finding the new vertex locations using weighted combinations of the old vertex positions.

For appropriately chosen subdivision rules, this process converges to give a smooth *limit surface* as the number of subdivision steps goes to infinity. In practice, just a few levels of subdivision typically suffice to give a good approximation of the limit surface. Figure 3.24 shows a simple example of a subdivision, where a tetrahedron has been subdivided zero, one, two, and six times. Figure 3.25 shows the effect of applying subdivision to the Killeroo model; on the top is the original control mesh, and below is the subdivision surface that the control mesh represents.



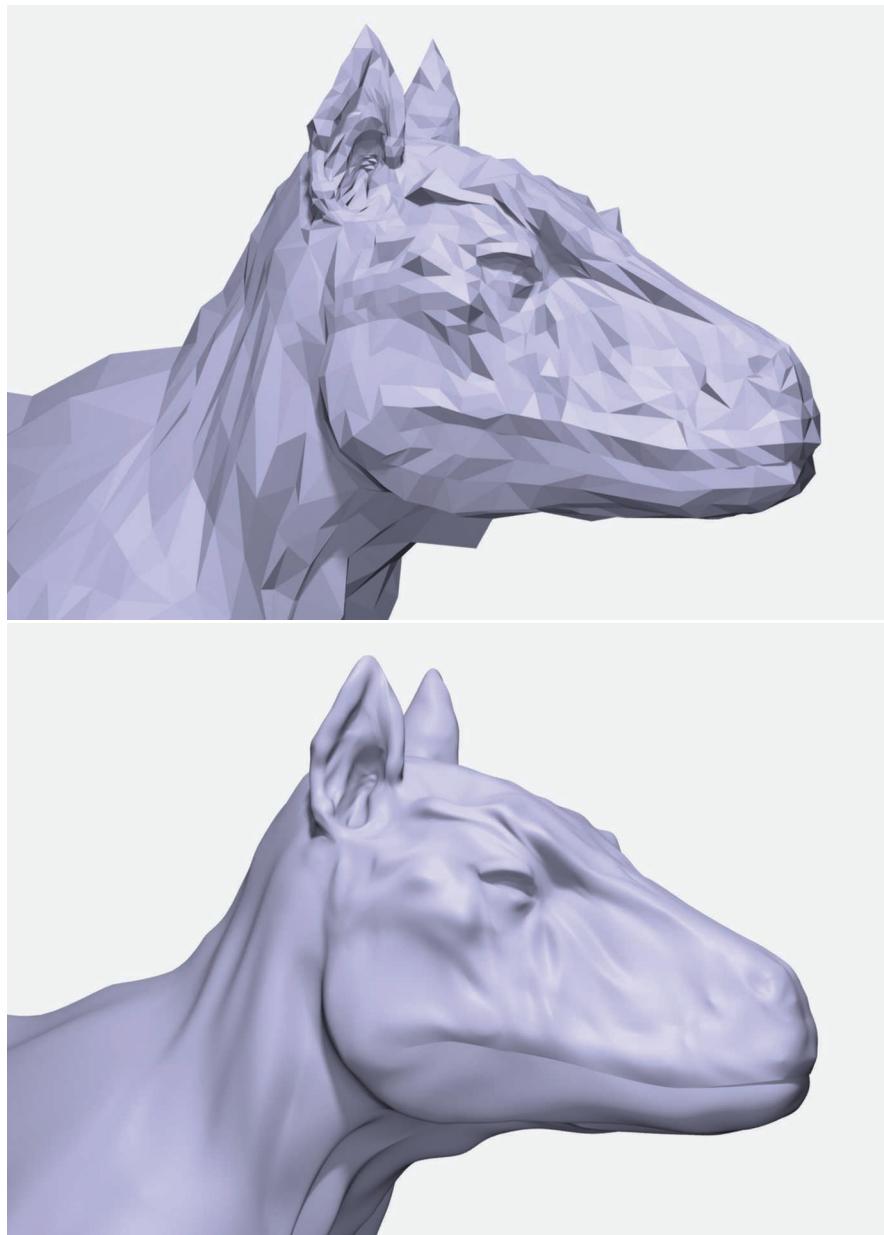
**Figure 3.24: Subdivision of a Tetrahedron.** From left to right, zero, one, two, and six subdivision steps have been used. (At zero levels, the vertices are just moved to lie on the limit surface.) As more subdivision is done, the mesh approaches the limit surface, the smooth surface described by the original mesh. Notice how the specular highlights become progressively more accurate and the silhouette edges appear smoother as more levels of subdivision are performed.

Although originally developed in the 1970s, subdivision surfaces have seen widespread use in recent years thanks to some important advantages over polygonal and spline-based representations of surfaces. The advantages of subdivision include the following:

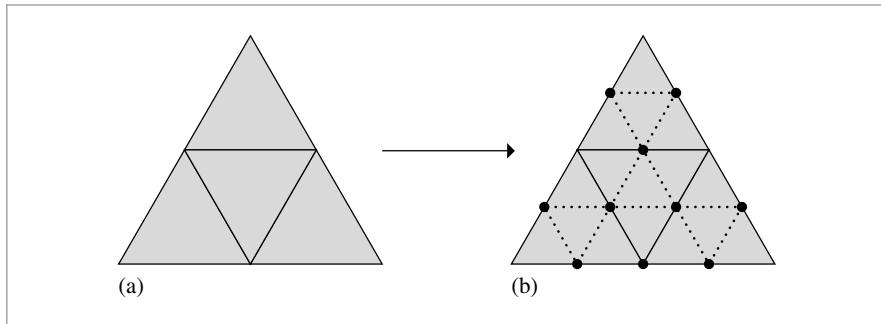
- Subdivision surfaces are smooth, as opposed to polygon meshes, which appear faceted when viewed close up, regardless of how finely they are modeled.
- Much of the existing infrastructure in modeling systems can be retargeted to subdivision. The classic toolbox of techniques for modeling polygon meshes can be applied to modeling subdivision control meshes.
- Subdivision surfaces are well suited to describing objects with complex topology, since they start with a control mesh of arbitrary (manifold) topology. Parametric surface models generally don't handle complex topology well.
- Subdivision methods are often generalizations of spline-based surface representations, so spline surfaces can often just be run through general subdivision surface renderers.
- It is easy to add detail to a localized region of a subdivision surface simply by adding faces to appropriate parts of the control mesh. This is much harder with spline representations.

Here, we will describe an implementation of *Loop subdivision surfaces*.<sup>7</sup> The Loop subdivision rules are based on triangular faces in the control mesh; faces with more than three vertices are triangulated at the start. At each subdivision step, all faces split into four child faces (Figure 3.26). New vertices are added along all of the edges of the original mesh, with positions computed using weighted averages of nearby vertices. Furthermore, the position of each original vertex is updated with a weighted average of its position and its new neighbors' positions. The implementation here uses weights based on improvements to Loop's method developed by Hoppe et al. (1994). We will not include discussion here about how these weights are derived. They must be chosen carefully to ensure that the limit surface actually has particular desirable smoothness properties, although subtle mathematics is necessary to prove that they indeed do this.

<sup>7</sup> Named after the inventor of the subdivision rules used, Charles Loop.



**Figure 3.25: Subdivision Applied to the Killeroo Model.** The control mesh (top) describes the subdivision surface shown below it. Subdivision is well suited to modeling shapes like this one, since it's easy to add detail locally by refining the control mesh, and there are no limitations on the topology of the final surface. (*Model courtesy of headus/Rezard.*)



**Figure 3.26: Basic Refinement Process for Loop Subdivision.** (a) The control mesh before subdivision. (b) The new mesh after one subdivision step. Each triangular face of the mesh has been subdivided into four new faces by splitting each of the edges and connecting the new vertices with new edges.

Rather than being implemented as a Shape in pbrt, subdivision surfaces are generated by a function, `LoopSubdivide()`, that applies the subdivision rules to a mesh represented by a collection of vertices and vertex indices and returns a vector of `Triangles` that represent the final subdivided mesh.

```
(LoopSubdiv Function Definitions) ≡
    std::vector<std::shared_ptr<Shape>> LoopSubdivide(
        const Transform *ObjectToWorld, const Transform *WorldToObject,
        bool reverseOrientation, int nLevels, int nIndices,
        const int *vertexIndices, int nVertices, const Point3f *p) {
    std::vector<SDVertex *> vertices;
    std::vector<SDFace *> faces;
    ⟨Allocate LoopSubdiv vertices and faces 185⟩
    ⟨Set face to vertex pointers 187⟩
    ⟨Set neighbor pointers in faces 188⟩
    ⟨Finish vertex initialization 190⟩
    ⟨Refine subdivision mesh into triangles 193⟩
}
```

### 3.8.1 MESH REPRESENTATION

The parameters to `LoopSubdivide()` specify a triangle mesh in exactly the same format used in the `TriangleMesh` constructor (Section 3.6): each face is described by three integer vertex indices, giving offsets into the vertex array `p` for the face's three vertices. We will need to process this data to determine which faces are adjacent to each other, which faces are adjacent to which vertices, and so on, in order to implement the subdivision algorithm.

We will shortly define `SDVertex` and `SDFace` structures, which hold data for vertices and faces in the subdivision mesh. `LoopSubdivide()` starts by allocating one instance of the `SDVertex` class for each vertex in the mesh and an `SDFace` for each face. For now, these are mostly uninitialized.

Point3f 68  
 SDFace 186  
 SDVertex 185  
 Shape 123  
 Transform 83  
 Triangle 156  
 TriangleMesh 154

*(Allocate LoopSubdiv vertices and faces) ≡*

```

    std::unique_ptr<SDVertex[]> verts(new SDVertex[nVertices]);
    for (int i = 0; i < nVertices; ++i)
        verts[i] = SDVertex(p[i]);
        vertices.push_back(&verts[i]);
    }
    int nFaces = nIndices / 3;
    std::unique_ptr<SDFace[]> fs(new SDFace[nFaces]);
    for (int i = 0; i < nFaces; ++i)
        faces.push_back(&fs[i]);

```

184

The Loop subdivision scheme, like most other subdivision schemes, assumes that the control mesh is *manifold*—no more than two faces share any given edge. Such a mesh may be closed or open: a *closed mesh* has no boundary, and all faces have adjacent faces across each of their edges. An *open mesh* has some faces that do not have all three neighbors. The implementation here supports both closed and open meshes.

In the interior of a triangle mesh, most vertices are adjacent to six faces and have six neighbor vertices directly connected to them with edges. On the boundaries of an open mesh, most vertices are adjacent to three faces and four vertices. The number of vertices directly adjacent to a vertex is called the vertex’s *valence*. Interior vertices with valence other than six, or boundary vertices with valence other than four, are called *extraordinary vertices*; otherwise, they are called *regular*.<sup>8</sup> Loop subdivision surfaces are smooth everywhere except at their extraordinary vertices.

Each *SDVertex* stores its position *p*, a Boolean that indicates whether it is a regular or extraordinary vertex, and a Boolean that records if it lies on the boundary of the mesh. It also holds a pointer to an arbitrary face adjacent to it; this pointer gives a starting point for finding all of the adjacent faces. Finally, there is a pointer to store the corresponding *SDVertex* for the next level of subdivision, if any.

*(LoopSubdiv Local Structures) ≡*

```

struct SDVertex {
    (SDVertex Constructor 185)
    (SDVertex Methods)
    Point3f p;
    SDFace *startFace = nullptr;
    SDVertex *child = nullptr;
    bool regular = false, boundary = false;
};

```

185

*(SDVertex Constructor) ≡*

```

SDVertex(const Point3f &p = Point3f(0, 0, 0)) : p(p) { }

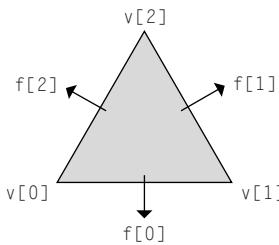
```

LoopSubdiv::faces 184  
 LoopSubdiv::vertices 184  
 Point3f 68  
 SDFace 186  
 SDVertex 185

The *SDFace* structure is where most of the topological information about the mesh is maintained. Because all faces are triangular, faces always store pointers to their three

---

<sup>8</sup> These terms are commonly used in the modeling literature, although *irregular* versus *regular* or *extraordinary* versus *ordinary* might be more intuitive.



**Figure 3.27:** Each triangular face stores three pointers to SDVertex objects  $v[i]$  and three pointers to neighboring faces  $f[i]$ . Neighboring faces are indexed using the convention that the  $i$ th edge is the edge from  $v[i]$  to  $v[(i+1)\%3]$ , and the neighbor across the  $i$ th edge is in  $f[i]$ .

vertices and pointers to the adjacent faces across its three edges. The corresponding face neighbor pointers will be `nullptr` if the face is on the boundary of an open mesh.

The face neighbor pointers are indexed such that if we label the edge from  $v[i]$  to  $v[(i+1)\%3]$  as the  $i$ th edge, then the neighbor face across that edge is stored in  $f[i]$  (Figure 3.27). This labeling convention is important to keep in mind. Later when we are updating the topology of a newly subdivided mesh, we will make extensive use of it to navigate around the mesh. Similarly to the SDVertex class, the SDFace also stores pointers to child faces at the next level of subdivision.

```
(LoopSubdiv Local Structures) +≡
struct SDFace {
    (SDFace Constructor)
    (SDFace Methods 191)
    SDVertex *v[3];
    SDFace *f[3];
    SDFace *children[4];
};
```

The SDFace constructor is straightforward—it simply sets these various pointers to `nullptr`—so it is not shown here.

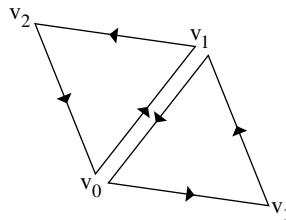
In order to simplify navigation of the SDFace data structure, we'll provide macros that make it easy to determine the vertex and face indices before or after a particular index. These macros add appropriate offsets and compute the result modulus three to handle cycling around.

```
(LoopSubdiv Macros) ≡
#define NEXT(i) (((i) + 1) % 3)
#define PREV(i) (((i) + 2) % 3)
```

In addition to requiring a manifold mesh, the subdivision code expects that the control mesh specified by the user will be *consistently ordered*—each *directed edge* in the mesh can be present only once. An edge that is shared by two faces should be specified in a different direction by each face. Consider two vertices,  $v_0$  and  $v_1$ , with an edge between them. We

SDFace 186

SDVertex 185



**Figure 3.28:** All of the faces in the input mesh must be specified so that each shared edge is given no more than once in each direction. Here, the edge from  $v_0$  to  $v_1$  is traversed from  $v_0$  to  $v_1$  by one face and from  $v_1$  to  $v_0$  by the other. Another way to think of this is in terms of face orientation: all faces' vertices should be given consistently in either clockwise or counterclockwise order, as seen from outside the mesh.

expect that one of the triangular faces that has this edge will specify its three vertices so that  $v_0$  is before  $v_1$ , and that the other face will specify its vertices so that  $v_1$  is before  $v_0$  (Figure 3.28). A Möbius strip is one example of a surface that cannot be consistently ordered, but such surfaces come up rarely in rendering so in practice this restriction is not a problem. Poorly formed mesh data from other programs that don't create consistently ordered meshes can be troublesome, however.

Given this assumption about the input data, the `LoopSubdivide()` can now initialize the mesh's topological data structures. It first loops over all of the faces and sets their `v` pointers to point to their three vertices. It also sets each vertex's `SDVertex::startFace` pointer to point to one of the vertex's neighboring faces. It doesn't matter which of its adjacent faces is used, so the implementation just keeps resetting it each time it comes across another face that the vertex is incident to, thus ensuring that all vertices have a non-nullptr face pointer by the time the loop is complete.

```
{Set face to vertex pointers} ≡ 184
    const int *vp = vertexIndices;
    for (int i = 0; i < nFaces; ++i, vp += 3) {
        SDFace *f = faces[i];
        for (int j = 0; j < 3; ++j) {
            SDVertex *v = vertices[vp[j]];
            f->v[j] = v;
            v->startFace = f;
        }
    }
```

`SDFace` 186  
`SDFace::v` 186  
`SDVertex` 185  
`SDVertex::startFace` 185

Now it is necessary to set each face's `f` pointer to point to its neighboring faces. This is a bit trickier, since face adjacency information isn't directly specified in the data passed to `LoopSubdivide()`. The implementation here loops over the faces and creates an `SDEdge` object for each of their three edges. When it comes to another face that shares the same edge, it can update both faces' neighbor pointers.

```
(LoopSubdiv Local Structures) +≡
  struct SDEdge {
    (SDEdge Constructor 188)
    (SDEdge Comparison Function 188)
    SDVertex *v[2];
    SDFace *f[2];
    int f0edgeNum;
  };

```

The SDEdge constructor takes pointers to the two vertices at each end of the edge. It orders them so that v[0] holds the one that is first in memory. This code may seem strange, but it is simply relying on the fact that pointers in C++ are effectively numbers that can be manipulated like integers<sup>9</sup> and that the ordering of vertices on an edge is arbitrary. Sorting the two vertices based on the addresses of their pointers guarantees that the edge (v<sub>a</sub>, v<sub>b</sub>) is correctly recognized as the same as the edge (v<sub>b</sub>, v<sub>a</sub>), regardless of what order the vertices are provided in.

```
(SDEdge Constructor) ≡ 188
  SDEdge(SDVertex *v0 = nullptr, SDVertex *v1 = nullptr) {
    v[0] = std::min(v0, v1);
    v[1] = std::max(v0, v1);
    f[0] = f[1] = nullptr;
    f0edgeNum = -1;
  }

```

The class also defines an ordering operation for SDEdge objects so that they can be stored in other data structures that rely on ordering being well defined.

```
(SDEdge Comparison Function) ≡ 188
  bool operator<(const SDEdge &e2) const {
    if (v[0] == e2.v[0]) return v[1] < e2.v[1];
    return v[0] < e2.v[0];
  }

```

Now the LoopSubdivide() function can get to work, looping over the edges in all of the faces and updating the neighbor pointers as it goes. It uses a set to store the edges that have only one adjacent face so far. The set makes it possible to search for a particular edge in  $O(\log n)$  time.

```
(Set neighbor pointers in faces) ≡ 184
  std::set<SDEdge> edges;
  for (int i = 0; i < nFaces; ++i) {
    SDFace *f = faces[i];
    for (int edgeNum = 0; edgeNum < 3; ++edgeNum) {
      (Update neighbor pointer for edgeNum 189)
    }
  }

```

SDEdge 188  
SDEdge::v 188  
SDFace 186  
SDVertex 185

---

<sup>9</sup> Segmented architectures notwithstanding.

For each edge in each face, the loop body creates an edge object and sees if the same edge has been seen previously. If so, it initializes both faces' neighbor pointers across the edge. If not, it adds the edge to the set of edges. The indices of the two vertices at the ends of the edge,  $v_0$  and  $v_1$ , are equal to the edge index and the edge index plus one.

```
<Update neighbor pointer for edgeNum> ≡ 188
    int v0 = edgeNum, v1 = NEXT(edgeNum);
    SDEdge e(f->v[v0], f->v[v1]);
    if (edges.find(e) == edges.end()) {
        <Handle new edge 189>
    } else {
        <Handle previously seen edge 189>
    }
```

Given an edge that hasn't been encountered before, the current face's pointer is stored in the edge object's  $f[0]$  member. Because the input mesh is assumed to be manifold, there can be at most one other face that shares this edge. When such a face is discovered, it can be used to initialize the neighboring face field. Storing the edge number of this edge in the current face allows the neighboring face to initialize its corresponding edge neighbor pointer.

```
<Handle new edge> ≡ 189
    e.f[0] = f;
    e.f0edgeNum = edgeNum;
    edges.insert(e);
```

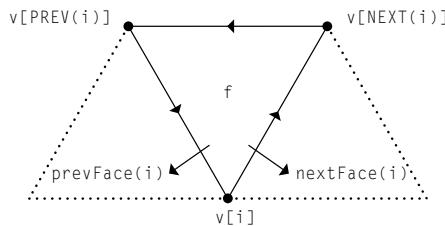
When the second face on an edge is found, the neighbor pointers for each of the two faces are set. The edge is then removed from the edge set, since no edge can be shared by more than two faces.

```
<Handle previously seen edge> ≡ 189
    e = *edges.find(e);
    e.f[0]->f[e.f0edgeNum] = f;
    f->f[edgeNum] = e.f[0];
    edges.erase(e);
```

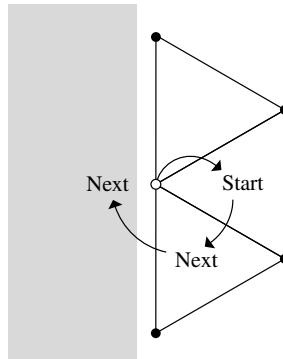
Now that all faces have proper neighbor pointers, the boundary and regular flags in each of the vertices can be set. In order to determine if a vertex is a boundary vertex, we'll define an ordering of faces around a vertex (Figure 3.29). For a vertex  $v[i]$  on a face  $f$ , we define the vertex's *next face* as the face across the edge from  $v[i]$  to  $v[NEXT(i)]$  and the *previous face* as the face across the edge from  $v[PREV(i)]$  to  $v[i]$ .

NEXT() 186  
 SDEdge 188  
 SDEdge::f 188  
 SDEdge::f0edgeNum 188  
 SDFace::f 186

By successively going to the next face around  $v$ , we can iterate over the faces adjacent to it. If we eventually return to the face we started at, then we are at an interior vertex; if we come to an edge with a `nullptr` neighbor pointer, then we're at a boundary vertex (Figure 3.30). Once the initialization routine has determined if this is a boundary vertex, it computes the valence of the vertex and sets the `regular` flag if the valence is 6 for an interior vertex or 4 for a boundary vertex; otherwise, it is an extraordinary vertex.



**Figure 3.29:** Given a vertex  $v[i]$  and a face that it is incident to,  $f$ , we define the *next face* as the face adjacent to  $f$  across the edge from  $v[i]$  to  $v[NEXT(i)]$ . The previous face is defined analogously.



**Figure 3.30:** We can determine if a vertex is a boundary vertex by starting from the adjacent face  $startFace$  and following next face pointers around the vertex. If we come to a face that has no next neighbor face, then the vertex is on a boundary. If we return to  $startFace$ , it's an interior vertex.

*(Finish vertex initialization)  $\equiv$*

184

```

for (int i = 0; i < nVertices; ++i) {
    SDVertex *v = vertices[i];
    SDFace *f = v->startFace;
    do {
        f = f->nextFace(v);
    } while (f && f != v->startFace);
    v->boundary = (f == nullptr);
    if (!v->boundary && v->valence() == 6)
        v->regular = true;
    else if (v->boundary && v->valence() == 4)
        v->regular = true;
    else
        v->regular = false;
}

```

SDFace 186

SDFace::nextFace() 192

SDVertex 185

SDVertex::boundary 185

SDVertex::regular 185

SDVertex::startFace 185

Because the valence of a vertex is frequently needed, we provide the method `SDVertex::valence()`.

```
(LoopSubdiv Inline Functions) ≡
    inline int SDVertex::valence() {
        SDFace *f = startFace;
        if (!boundary) {
            (Compute valence of interior vertex 191)
        } else {
            (Compute valence of boundary vertex 191)
        }
    }
```

To compute the valence of a nonboundary vertex, this method counts the number of the adjacent faces starting by following each face's neighbor pointers around the vertex until it reaches the starting face. The valence is equal to the number of faces visited.

```
(Compute valence of interior vertex) ≡ 191
    int nf = 1;
    while ((f = f->nextFace(this)) != startFace)
        ++nf;
    return nf;
```

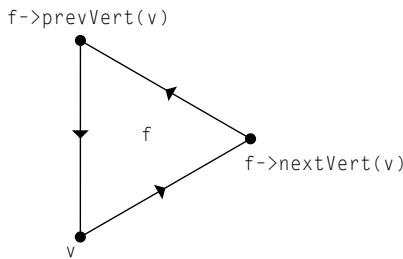
For boundary vertices we can use the same approach, although in this case, the valence is one more than the number of adjacent faces. The loop over adjacent faces is slightly more complicated here: it follows pointers to the next face around the vertex until it reaches the boundary, counting the number of faces seen. It then starts again at `startFace` and follows previous face pointers until it encounters the boundary in the other direction.

```
(Compute valence of boundary vertex) ≡ 191
    int nf = 1;
    while ((f = f->nextFace(this)) != nullptr)
        ++nf;
    f = startFace;
    while ((f = f->prevFace(this)) != nullptr)
        ++nf;
    return nf + 1;
```

`SDFace` 186  
`SDFace::nextFace()` 192  
`SDFace::prevFace()` 192  
`SDFace::v` 186  
`SDFace::vnum()` 191  
`SDVertex` 185  
`SDVertex::boundary` 185  
`SDVertex::startFace` 185  
`SDVertex::valence()` 191  
`Severe()` 1068

`SDFace::vnum()` is a utility function that finds the index of a given vertex pointer. It is a fatal error to pass a pointer to a vertex that isn't part of the current face—this case would represent a bug elsewhere in the subdivision code.

```
(SDFace Methods) ≡ 186
    int vnum(SDVertex *vert) const {
        for (int i = 0; i < 3; ++i)
            if (v[i] == vert) return i;
        Severe("Basic logic error in SDFace::vnum()");
        return -1;
    }
```



**Figure 3.31:** Given a vertex  $v$  on a face  $f$ , the method  $f->\text{prevVert}(v)$  returns the previous vertex around the face from  $v$ , and  $f->\text{nextVert}(v)$  returns the next vertex, where “next” and “previous” are defined by the original ordering of vertices when this face was defined.

Since the next face for a vertex  $v[i]$  on a face  $f$  is over the  $i$ th edge (recall the mapping of edge neighbor pointers from Figure 3.27), we can find the appropriate face neighbor pointer easily given the index  $i$  for the vertex, which the  $\text{vnum}()$  utility function provides. The previous face is across the edge from  $\text{PREV}(i)$  to  $i$ , so the method returns  $f[\text{PREV}(i)]$  for the previous face.

$\langle \text{SDFace Methods} \rangle + \equiv$

186

```
SDFace *nextFace(SDVertex *vert) {
    return f[vnum(vert)];
}
```

$\langle \text{SDFace Methods} \rangle + \equiv$

186

```
SDFace *prevFace(SDVertex *vert) {
    return f[PREV(vnum(vert))];
}
```

It is also useful to be able to get the next and previous vertices around a face starting at any vertex. The `SDFace::nextVert()` and `SDFace::prevVert()` methods do just that (Figure 3.31).

$\langle \text{SDFace Methods} \rangle + \equiv$

186

```
SDVertex *nextVert(SDVertex *vert) {
    return v[NEXT(vnum(vert))];
}
```

NEXT() 186

PREV() 186

SDFace 186

SDFace::f 186

SDFace::nextVert() 192

SDFace::prevVert() 192

SDFace::v 186

SDFace::vnum() 191

SDVertex 185

$\langle \text{SDFace Methods} \rangle + \equiv$

186

```
SDVertex *prevVert(SDVertex *vert) {
    return v[PREV(vnum(vert))];
}
```

### 3.8.2 SUBDIVISION

Now we can show how subdivision proceeds with the modified Loop rules. The implementation here applies subdivision a fixed number of times to generate a triangle

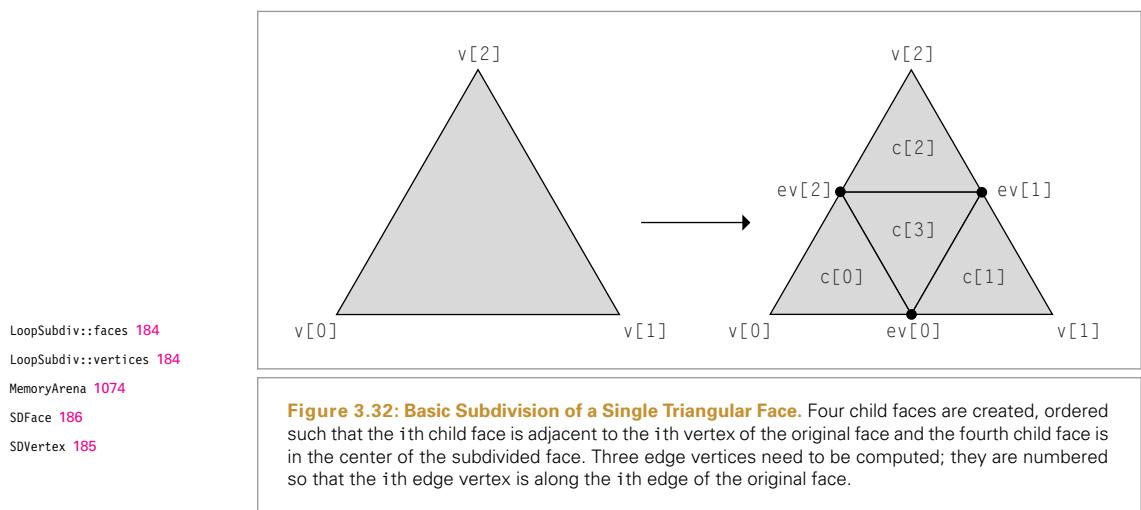
mesh for rendering; Exercise 3.11 at the end of the chapter discusses adaptive subdivision, where each original face is subdivided enough times so that the result looks smooth from a particular viewpoint rather than just using a fixed number of levels of subdivision, which may over-subdivide some areas while simultaneously under-subdividing others.

The *(Refine subdivision mesh into triangles)* fragment repeatedly applies the subdivision rules to the mesh, each time generating a new mesh to be used as the input to the next step. After each subdivision step, the *f* and *v* arrays are updated to point to the faces and vertices from the level of subdivision just computed. When it's done subdividing, a triangle mesh representation of the surface is returned.

An instance of the *MemoryArena* class is used to allocate temporary storage through this process. This class, defined in Section A.4.3, provides a custom memory allocation method that quickly allocates memory, automatically freeing the memory when it goes out of scope.

```
<Refine subdivision mesh into triangles> ≡ 184
    std::vector<SDFace *> f = faces;
    std::vector<SDVertex *> v = vertices;
    MemoryArena arena;
    for (int i = 0; i < nLevels; ++i) {
        (Update f and v for next level of subdivision 194)
    }
    (Push vertices to limit surface 203)
    (Compute vertex tangents on limit surface 203)
    (Create triangle mesh from subdivision mesh)
```

The main loop of a subdivision step proceeds as follows: it creates vectors to store the vertices and faces at the current level of subdivision and then proceeds to compute new vertex positions and update the topological representation for the refined mesh. Figure 3.32 shows the basic refinement rules for faces in the mesh. Each face is split into



four child faces, such that the  $i$ th child face is next to the  $i$ th vertex of the input face and the final face is in the center. Three new vertices are then computed along the split edges of the original face.

*(Update f and v for next level of subdivision) ≡* 193

```
std::vector<SDFace *> newFaces;
std::vector<SDVertex *> newVertices;
(Allocate next level of children in mesh tree 194)
(Update vertex positions and create new edge vertices 194)
(Update new mesh topology 201)
(Prepare for next level of subdivision 203)
```

First, storage is allocated for the updated values of the vertices already present in the input mesh. The method also allocates storage for the child faces. It doesn't yet do any initialization of the new vertices and faces other than setting the regular and boundary flags for the vertices since subdivision leaves boundary vertices on the boundary and interior vertices in the interior and it doesn't change the valence of vertices in the mesh.

*(Allocate next level of children in mesh tree) ≡* 194

```
for (SDVertex *vertex : v) {
    vertex->child = arena.Alloc<SDVertex>();
    vertex->child->regular = vertex->regular;
    vertex->child->boundary = vertex->boundary;
    newVertices.push_back(vertex->child);
}
for (SDFace *face : f) {
    for (int k = 0; k < 4; ++k) {
        face->children[k] = arena.Alloc<SDFace>();
        newFaces.push_back(face->children[k]);
    }
}
```

## Computing New Vertex Positions

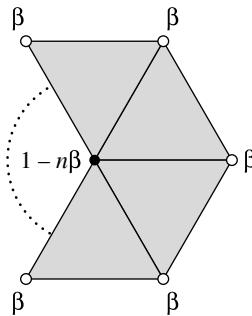
Before worrying about initializing the topology of the subdivided mesh, the refinement method computes positions for all of the vertices in the mesh. First, it considers the problem of computing updated positions for all of the vertices that were already present in the mesh; these vertices are called *even vertices*. It then computes the new vertices on the split edges. These are called *odd vertices*.

*(Update vertex positions and create new edge vertices) ≡* 194

*(Update vertex positions for even vertices 195)*  
*(Compute new odd edge vertices 198)*

Different techniques are used to compute the updated positions for each of the different types of even vertices—regular and extraordinary, boundary and interior. This gives four cases to handle.

MemoryArena::Alloc() [1074](#)  
 SDFace [186](#)  
 SDFace::children [186](#)  
 SDVertex [185](#)  
 SDVertex::boundary [185](#)  
 SDVertex::regular [185](#)



**Figure 3.33:** The new position  $v'$  for a vertex  $v$  is computed by weighting the adjacent vertices  $v_i$  by a weight  $\beta$  and weighting  $v$  by  $(1 - n\beta)$ , where  $n$  is the valence of  $v$ . The adjacent vertices  $v_i$  are collectively referred to as the *one-ring* around  $v$ .

```

⟨Update vertex positions for even vertices⟩ ≡
    for (SDVertex *vertex : v) {
        if (!vertex->boundary) {
            ⟨Apply one-ring rule for even vertex 196⟩
        } else {
            ⟨Apply boundary rule for even vertex 197⟩
        }
    }

```

194

For both types of interior vertices, we take the set of vertices adjacent to each vertex (called the *one-ring* around it, reflecting the fact that it's a ring of neighbors) and weight each of the neighbor vertices by a weight  $\beta$  (Figure 3.33). The vertex we are updating, in the center, is weighted by  $1 - n\beta$ , where  $n$  is the valence of the vertex. Thus, the new position  $v'$  for a vertex  $v$  is

$$v' = (1 - n\beta)v + \sum_{i=1}^N \beta v_i.$$

This formulation ensures that the sum of weights is one, which guarantees the convex hull property of Loop subdivision surfaces, which ensures that the final mesh is in the convex hull of the control mesh. The position of the vertex being updated is only affected by vertices that are nearby; this is known as *local support*. Loop subdivision is particularly efficient because its subdivision rules all have this property.

The specific weight  $\beta$  used for this step is a key component of the subdivision method and must be chosen carefully in order to ensure smoothness of the limit surface, among other desirable properties.<sup>10</sup> The `beta()` function that follows computes a  $\beta$  value based

SDVertex 185

SDVertex::boundary 185

10 Again, see the papers cited at the start of this section and in the “Further Reading” section for information about how values like  $\beta$  are derived.

on the vertex's valence that ensures smoothness. For regular interior vertices, `beta()` returns 1/16. Since this is a common case, the implementation uses 1/16 directly instead of calling `beta()` every time.

```
(Apply one-ring rule for even vertex) ≡ 195
if (vertex->regular)
    vertex->child->p = weightOneRing(vertex, 1.f / 16.f);
else
    vertex->child->p = weightOneRing(vertex, beta(vertex->valence()));
```

```
(LoopSubdiv Inline Functions) +≡
inline Float beta(int valence) {
    if (valence == 3) return 3.f / 16.f;
    else return 3.f / (8.f * valence);
}
```

The `weightOneRing()` function loops over the one-ring of adjacent vertices and applies the given weight to compute a new vertex position. It uses the `SDVertex::oneRing()` method, defined in the following, which returns the positions of the vertices around the vertex `vert`.

```
(LoopSubdiv Function Definitions) +≡
static Point3f weightOneRing(SDVertex *vert, Float beta) {
    (Put vert one-ring in pRing 196)
    Point3f p = (1 - valence * beta) * vert->p;
    for (int i = 0; i < valence; ++i)
        p += beta * pRing[i];
    return p;
}
```

Because a variable number of vertices are in the one-rings, we use the `ALLOCA()` macro to efficiently allocate space to store their positions.

```
(Put vert one-ring in pRing) ≡ 196, 198
int valence = vert->valence();
Point3f *pRing = ALLOCA(Point3f, valence);
vert->oneRing(pRing);
```

`ALLOCA()` 1071

`Float` 1062

`LoopSubdiv::weightOneRing()` 196

`Point3f` 68

`SDVertex` 185

`SDVertex::boundary` 185

`SDVertex::child` 185

`SDVertex::oneRing()` 196

`SDVertex::p` 185

`SDVertex::regular` 185

`SDVertex::valence()` 191

The `oneRing()` method assumes that the pointer passed in points to an area of memory large enough to hold the one-ring around the vertex.

```
(LoopSubdiv Function Definitions) +≡
void SDVertex::oneRing(Point3f *p) {
    if (!boundary) {
        (Get one-ring vertices for interior vertex 197)
    } else {
        (Get one-ring vertices for boundary vertex 197)
    }
}
```

It's relatively easy to get the one-ring around an interior vertex by looping over the faces adjacent to the vertex and for each face retaining the vertex after the center vertex. (Brief sketching with pencil and paper should convince you that this process returns all of the vertices in the one-ring.)

*(Get one-ring vertices for interior vertex) ≡*

```
SDFace *face = startFace;
do {
    *p++ = face->nextVert(this)->p;
    face = face->nextFace(this);
} while (face != startFace);
```

196

The one-ring around a boundary vertex is a bit trickier. The implementation here carefully stores the one-ring in the given `Point3f` array so that the first and last entries in the array are the two adjacent vertices along the boundary. This ordering is important because the adjacent boundary vertices will often be weighted differently from the adjacent vertices that are in the interior of the mesh. Doing so requires that we first loop around neighbor faces until we reach a face on the boundary and then loop around the other way, storing vertices one by one.

*(Get one-ring vertices for boundary vertex) ≡*

```
SDFace *face = startFace, *f2;
while ((f2 = face->nextFace(this)) != nullptr)
    face = f2;
*p++ = face->nextVert(this)->p;
do {
    *p++ = face->prevVert(this)->p;
    face = face->prevFace(this);
} while (face != nullptr);
```

196

For vertices on the boundary, the new vertex's position is based only on the two neighboring boundary vertices (Figure 3.34). Not depending on interior vertices ensures that two abutting surfaces that share the same vertices on the boundary will have abutting limit surfaces. The `weightBoundary()` utility function applies the given weighting on the two neighbor vertices  $v_1$  and  $v_2$  to compute the new position  $v'$  as

$$v' = (1 - 2\beta)v + \beta v_1 + \beta v_2.$$

LoopSubdiv::weightBoundary()  
198  
SDFace 186  
SDFace::nextFace() 192  
SDFace::nextVert() 192  
SDFace::prevFace() 192  
SDFace::prevVert() 192  
SDVertex::child 185  
SDVertex::oneRing() 196  
SDVertex::p 185  
SDVertex::startFace 195

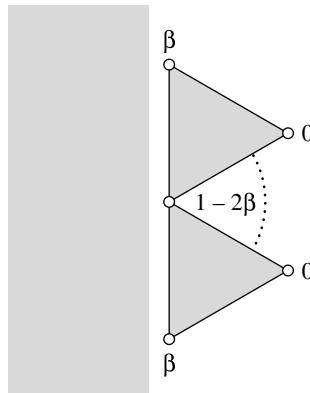
The same weight of 1/8 is used for both regular and extraordinary vertices.

*(Apply boundary rule for even vertex) ≡*

```
vertex->child->p = weightBoundary(vertex, 1.f / 8.f);
```

195

The `weightBoundary()` utility function applies the given weights at a boundary vertex. Because the `SDVertex::oneRing()` function orders the boundary vertex's one-ring such that the first and last entries are the boundary neighbors, the implementation here is particularly straightforward.



**Figure 3.34: Subdivision on a Boundary Edge.** The new position for the vertex in the center is computed by weighting it and its two neighbor vertices by the weights shown.

*(LoopSubdiv Function Definitions)* +≡

```
static Point3f weightBoundary(SDVertex *vert, Float beta) {
    {Put vert one-ring in pRing 196}
    Point3f p = (1 - 2 * beta) * vert->p;
    p += beta * pRing[0];
    p += beta * pRing[valence - 1];
    return p;
}
```

Now the refinement method computes the positions of the odd vertices—the new vertices along the split edges of the mesh. It loops over each edge of each face in the mesh, computing the new vertex that splits the edge (Figure 3.35). For interior edges, the new vertex is found by weighting the two vertices at the ends of the edge and the two vertices across from the edge on the adjacent faces. It loops through all three edges of each face, and each time it comes to an edge that hasn't been seen before it computes and stores the new odd vertex for the edge in the edgeVerts associative array.

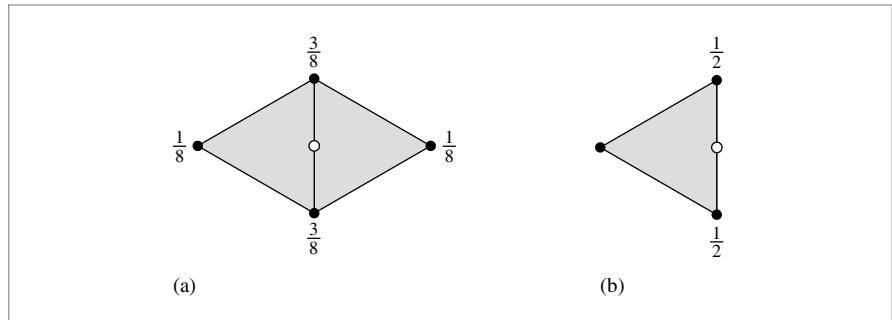
*(Compute new odd edge vertices)* ≡

194

```
std::map<SDEdge, SDVertex *> edgeVerts;
for (SDFace *face : f) {
    for (int k = 0; k < 3; ++k)
        {Compute odd vertex on kth edge 199}
}
```

Float 1062  
Point3f 68  
SDEdge 188  
SDFace 186  
SDVertex 185  
SDVertex::p 185

As was done when setting the face neighbor pointers in the original mesh, an SDEdge object is created for the edge and checked to see if it is in the set of edges that have already been visited. If it isn't, the new vertex on this edge is computed and added to the map, which is an associative array structure that performs efficient lookups.



**Figure 3.35: Subdivision Rule for Edge Split.** The position of the new odd vertex, marked with an open circle, is found by weighting the two vertices at the ends of the edge and the two vertices opposite it on the adjacent triangles. (a) The weights for an interior vertex; (b) the weights for a boundary vertex.

```

⟨Compute odd vertex on kth edge⟩ ≡
    SDEdge edge(face->v[k], face->v[NEXT(k)]);
    SDVertex *vert = edgeVerts[edge];
    if (!vert) {
        ⟨Create and initialize new odd vertex 199⟩
        ⟨Apply edge rules to compute new vertex position 200⟩
        edgeVerts[edge] = vert;
    }
}

```

198

In Loop subdivision, the new vertices added by subdivision are always regular. (This means that the proportion of extraordinary vertices with respect to regular vertices will decrease with each level of subdivision.) Therefore, the regular member of the new vertex can immediately be set to true. The boundary member can also be easily initialized, by checking to see if there is a neighbor face across the edge that is being split. Finally, the new vertex's startFace pointer can also be set here. For all odd vertices on the edges of a face, the center child (child face number three) is guaranteed to be adjacent to the new vertex.

```

MemoryArena::Alloc() 1074
NEXT() 186
SDEdge 188
SDFace::children 186
SDFace::f 186
SDFace::otherVert() 200
SDFace::v 186
SDVertex 185
SDVertex::boundary 185
SDVertex::regular 185
SDVertex::startFace 185

```

```

⟨Create and initialize new odd vertex⟩ ≡
    vert = arena.Alloc<SDVertex>();
    newVertices.push_back(vert);
    vert->regular = true;
    vert->boundary = (face->f[k] == nullptr);
    vert->startFace = face->children[3];
}

```

199

For odd boundary vertices, the new vertex is just the average of the two adjacent vertices. For odd interior vertices, the two vertices at the ends of the edge are given weight 1/8, and the two vertices opposite the edge are given weight 3/8 (Figure 3.35). These last two vertices can be found using the SDFace::otherVert() utility function, which returns the vertex opposite a given edge of a face.

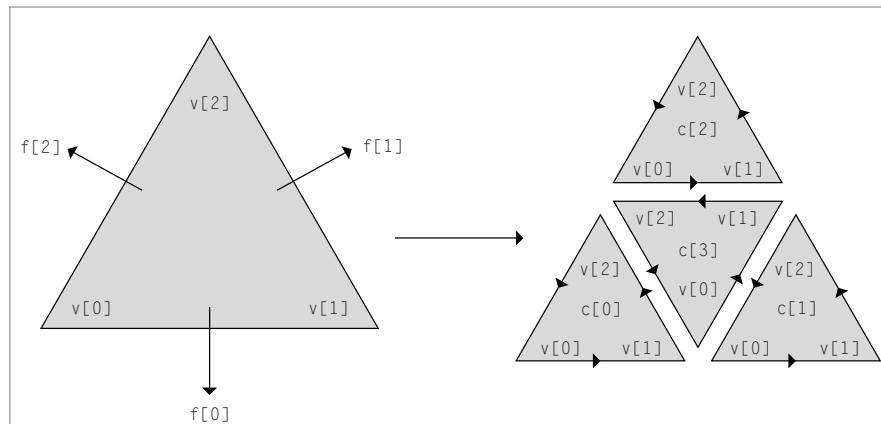
```
(Apply edge rules to compute new vertex position) ≡ 199
if (vert->boundary) {
    vert->p = 0.5f * edge.v[0]->p;
    vert->p += 0.5f * edge.v[1]->p;
} else {
    vert->p = 3.f/8.f * edge.v[0]->p;
    vert->p += 3.f/8.f * edge.v[1]->p;
    vert->p += 1.f/8.f * face->otherVert(edge.v[0], edge.v[1])->p;
    vert->p += 1.f/8.f *
        face->f[k]->otherVert(edge.v[0], edge.v[1])->p;
}
```

The `SDFace::otherVert()` method is self-explanatory:

```
(SDFace Methods) +≡ 186
SDVertex *otherVert(SDVertex *v0, SDVertex *v1) {
    for (int i = 0; i < 3; ++i)
        if (v[i] != v0 && v[i] != v1)
            return v[i];
    Severe("Basic logic error in SDVertex::otherVert()");
    return nullptr;
}
```

## Updating Mesh Topology

In order to keep the details of the topology update as straightforward as possible, the numbering scheme for the subdivided faces and their vertices has been chosen carefully (Figure 3.36). Review the figure carefully; the conventions shown are key to the next few pages.



**Figure 3.36:** Each face is split into four child faces, such that the  $i$ th child is adjacent to the  $i$ th vertex of the original face, and such that the  $i$ th child face's  $j$ th vertex is the child of the  $i$ th vertex of the original face. The vertices of the center child are oriented such that the  $i$ th vertex is the odd vertex along the  $i$ th edge of the parent face.

`SDEdge::v` 188  
`SDFace::otherVert()` 200  
`SDFace::v` 186  
`SDVertex` 185  
`SDVertex::boundary` 185  
`SDVertex::p` 185  
`Severe()` 1068

There are four main tasks required to update the topological pointers of the refined mesh:

1. The odd vertices' `SDVertex::startFace` pointers need to store a pointer to one of their adjacent faces.
2. Similarly, the even vertices' `SDVertex::startFace` pointers must be set.
3. The new faces' neighbor `f[i]` pointers need to be set to point to the neighboring faces.
4. The new faces' `v[i]` pointers need to point to the appropriate vertices.

The `startFace` pointers of the odd vertices were already initialized when they were first created. We'll handle the other three tasks in order here.

*(Update new mesh topology) ≡*

194

*(Update even vertex face pointers 201)*

*(Update face neighbor pointers 201)*

*(Update face vertex pointers 202)*

If a vertex is the  $i$ th vertex of its `startFace`, then it is guaranteed that it will be adjacent to the  $i$ th child face of `startFace`. Therefore, it is just necessary to loop through all the parent vertices in the mesh, and for each one find its vertex index in its `startFace`. This index can then be used to find the child face adjacent to the new even vertex.

*(Update even vertex face pointers) ≡*

201

```
for (SDVertex *vertex : v) {
    int vertNum = vertex->startFace->vnum(vertex);
    vertex->child->startFace =
        vertex->startFace->children[vertNum];
}
```

Next, the face neighbor pointers for the newly created faces are updated. We break this into two steps: one to update neighbors among children of the same parent, and one to do neighbors across children of different parents. This involves some tricky pointer manipulation.

*(Update face neighbor pointers) ≡*

201

```
for (SDFace *face : f) {
    for (int j = 0; j < 3; ++j) {
        (Update children f pointers for siblings 201)
        (Update children f pointers for neighbor children 202)
    }
}
```

For the first step, recall that the interior child face is always stored in `children[3]`. Furthermore, the  $k + 1$ st child face (for  $k = 0, 1, 2$ ) is across the  $k$ th edge of the interior face, and the interior face is across the  $k + 1$ st edge of the  $k$ th face.

*(Update children f pointers for siblings) ≡*

201

```
face->children[3]->f[j] = face->children[NEXT(j)];
face->children[j]->f[NEXT(j)] = face->children[3];
```

We'll now update the children's face neighbor pointers that point to children of other parents. Only the first three children need to be addressed here; the interior child's

`NEXT()` 186  
`SDFace` 186  
`SDFace::children` 186  
`SDFace::f` 186  
`SDFace::vnum()` 191  
`SDVertex` 185  
`SDVertex::startFace` 185

neighbor pointers have already been fully initialized. Inspection of Figure 3.36 reveals that the  $k$ th and  $\text{PREV}(k)$ th edges of the  $i$ th child need to be set. To set the  $k$ th edge of the  $k$ th child, we first find the  $k$ th edge of the parent face, then the neighbor parent  $f_2$  across that edge. If  $f_2$  exists (meaning we aren't on a boundary), the neighbor parent index for the vertex  $v[k]$  is found. That index is equal to the index of the neighbor child we are searching for. This process is then repeated to find the child across the  $\text{PREV}(k)$ th edge.

*(Update children f pointers for neighbor children) ≡*

201

```
SDFace *f2 = face->f[j];
face->children[j]->f[j] =
    f2 ? f2->children[f2->vnum(face->v[j])] : nullptr;
f2 = face->f[PREV(j)];
face->children[j]->f[PREV(j)] =
    f2 ? f2->children[f2->vnum(face->v[j])] : nullptr;
```

Finally, we handle the fourth step in the topological updates: setting the children faces' vertex pointers.

*(Update face vertex pointers) ≡*

201

```
for (SDFace *face : f) {
    for (int j = 0; j < 3; ++j) {
        (Update child vertex pointer to new even vertex 202)
        (Update child vertex pointer to new odd vertex 202)
    }
}
```

For the  $k$ th child face (for  $k = 0, 1, 2$ ), the  $k$ th vertex corresponds to the even vertex that is adjacent to the child face. For the noninterior child faces, there is one even vertex and two odd vertices; for the interior child face, there are three odd vertices. This vertex can be found by following the child pointer of the parent vertex, available from the parent face.

*(Update child vertex pointer to new even vertex) ≡*

202

```
face->children[j]->v[j] = face->v[j]->child;
```

To update the rest of the vertex pointers, the `edgeVerts` associative array is reused to find the odd vertex for each split edge of the parent face. Three child faces have that vertex as an incident vertex. The vertex indices for the three faces are easily found, again based on the numbering scheme established in Figure 3.36.

*(Update child vertex pointer to new odd vertex) ≡*

202

```
SDVertex *vert = edgeVerts[SDEdge(face->v[j], face->v[NEXT(j)])];
face->children[j]->v[NEXT(j)] = vert;
face->children[NEXT(j)]->v[j] = vert;
face->children[3]->v[j] = vert;
```

After the geometric and topological work has been done for a subdivision step, the newly created vertices and faces are moved into the `v` and `f` arrays:

NEXT() 186  
 PREV() 186  
 SDEdge 188  
 SDFace 186  
 SDFace::children 186  
 SDFace::f 186  
 SDFace::v 186  
 SDFace::vnum() 191  
 SDVertex 185  
 SDVertex::child 185

*(Prepare for next level of subdivision) ≡*

```
f = newFaces;
v = newVertices;
```

194

## To the Limit Surface and Output

One of the remarkable properties of subdivision surfaces is that there are special subdivision rules that give the positions that the vertices of the mesh would have if we continued subdividing forever. We apply these rules here to initialize an array of limit surface positions, `pLimit`. Note that it's important to temporarily store the limit surface positions somewhere other than in the vertices while the computation is taking place. Because the limit surface position of each vertex depends on the original positions of its surrounding vertices, the original positions of all vertices must remain unchanged until the computation is complete.

The limit rule for a boundary vertex weights the two neighbor vertices by 1/5 and the center vertex by 3/5. The rule for interior vertices is based on a function `loopGamma()`, which computes appropriate vertex weights based on the valence of the vertex.

*(Push vertices to limit surface) ≡*

193

```
std::unique_ptr<Point3f[]> pLimit(new Point3f[v.size()]);
for (size_t i = 0; i < v.size(); ++i) {
    if (v[i]→boundary)
        pLimit[i] = weightBoundary(v[i], 1.f / 5.f);
    else
        pLimit[i] = weightOneRing(v[i], loopGamma(v[i]→valence()));
}
for (size_t i = 0; i < v.size(); ++i)
    v[i]→p = pLimit[i];
```

*(LoopSubdiv Inline Functions) +≡*

```
inline Float loopGamma(int valence) {
    return 1.f / (valence + 3.f / (8.f * beta(valence)));
}
```

Cross() 65  
 Float 1062  
`LoopSubdiv::beta()` 196  
`LoopSubdiv::loopGamma()` 203  
`LoopSubdiv::weightBoundary()` 198  
`LoopSubdiv::weightOneRing()` 196  
 Normal3f 71  
 Point3f 68  
 SDVertex 185  
`SDVertex::boundary` 185  
`SDVertex::oneRing()` 196  
`SDVertex::valence()` 191  
 Vector3f 60

In order to generate a smooth-looking triangle mesh with per-vertex surface normals, a pair of nonparallel tangent vectors to the limit surface is computed at each vertex. As with the limit rule for positions, this is an analytic computation that gives the precise tangents on the actual limit surface.

*(Compute vertex tangents on limit surface) ≡*

193

```
std::vector<Normal3f> Ns;
Ns.reserve(v.size());
std::vector<Point3f> pRing(16, Point3f());
```

```

for (SDVertex *vertex : v) {
    Vector3f S(0,0,0), T(0,0,0);
    int valence = vertex->valence();
    if (valence > (int)pRing.size())
        pRing.resize(valence);
    vertex->oneRing(&pRing[0]);
    if (!vertex->boundary) {
        <Compute tangents of interior face 204>
    } else {
        <Compute tangents of boundary face 206>
    }
    Ns.push_back(Normal3f(Cross(S, T)));
}

```

Figure 3.37 shows the setting for computing tangents in the mesh interior. The center vertex is given a weight of zero, and the neighbors are given weights  $w_i$ . To compute the first tangent vector  $s$ , the weights are

$$w_i = \cos\left(\frac{2\pi i}{n}\right),$$

where  $n$  is the valence of the vertex. The second tangent  $t$  is computed with weights

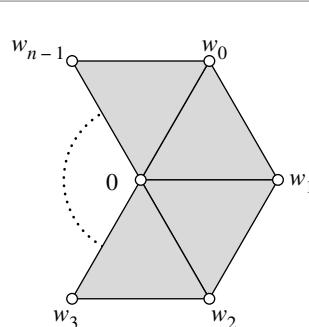
$$w_i = \sin\left(\frac{2\pi i}{n}\right).$$

*<Compute tangents of interior face>* 203

```

for (int j = 0; j < valence; ++j) {
    S += std::cos(2 * Pi * j / valence) * Vector3f(pRing[j]);
    T += std::sin(2 * Pi * j / valence) * Vector3f(pRing[j]);
}

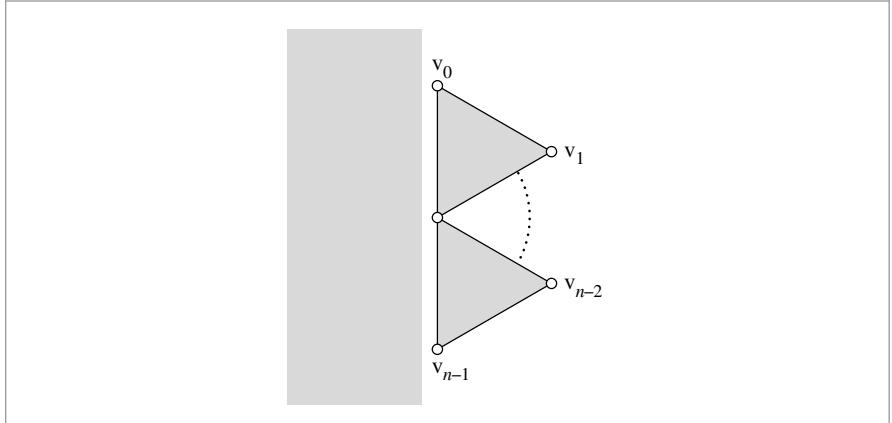
```



**Figure 3.37:** To compute tangents for interior vertices, the one-ring vertices are weighted with weights  $w_i$ . The center vertex, where the tangent is being computed, always has a weight of 0.

Pi 1063

Vector3f 60



**Figure 3.38:** Tangents at boundary vertices are also computed as weighted averages of the adjacent vertices. However, some of the boundary tangent rules incorporate the value of the center vertex.

Tangents on boundary vertices are a bit trickier. Figure 3.38 shows the ordering of vertices in the one-ring expected in the following discussion.

The first tangent, known as the *across tangent*, is given by the vector between the two neighboring boundary vertices:

$$\mathbf{s} = \mathbf{v}_{n-1} - \mathbf{v}_0.$$

The second tangent, known as the *transverse tangent*, is computed based on the vertex's valence. The center vertex is given a weight  $w_c$  and the one-ring vertices are given weights specified by a vector  $(w_0, w_1, \dots, w_{n-1})$ . The transverse tangent rules we will use are

Valence	$w_c$	$w_i$
2	-2	(1, 1)
3	-1	(0, 1, 0)
4 (regular)	-2	(-1, 2, 2, -1)

For valences of 5 and higher,  $w_c = 0$  and

$$w_0 = w_{n-1} = \sin \theta$$

$$w_i = (2 \cos \theta - 2) \sin(\theta i),$$

where

$$\theta = \frac{\pi}{n-1}.$$

Although we will not prove it here, these weights sum to zero for all values of  $i$ . This guarantees that the weighted sum is in fact a tangent vector.

```

⟨Compute tangents of boundary face⟩ ≡ 203
S = pRing[valence - 1] - pRing[0];
if (valence == 2)
    T = Vector3f(pRing[0] + pRing[1] - 2 * vertex->p);
else if (valence == 3)
    T = pRing[1] - vertex->p;
else if (valence == 4) // regular
    T = Vector3f(-1 * pRing[0] + 2 * pRing[1] + 2 * pRing[2] +
                  -1 * pRing[3] + -2 * vertex->p);
else {
    Float theta = Pi / float(valence - 1);
    T = Vector3f(std::sin(theta) * (pRing[0] + pRing[valence - 1]));
    for (int k = 1; k < valence - 1; ++k) {
        Float wt = (2 * std::cos(theta) - 2) * std::sin((k) * theta);
        T += Vector3f(wt * pRing[k]);
    }
    T = -T;
}

```

Finally, the fragment *(Create triangle mesh from subdivision mesh)* initializes a vector of `Triangles` corresponding to the triangulation of the limit surface. We won't include it here, since it's just a straightforward transformation of the subdivided mesh into an indexed triangle mesh.

## \* 3.9 MANAGING ROUNDING ERROR

Thus far, we've been discussing ray–shape intersection algorithms purely with respect to idealized arithmetic operations based on the real numbers. This approach has gotten us far, although the fact that computers can only represent finite quantities and therefore can't actually represent all of the real numbers is important. In place of real numbers, computers use floating-point numbers, which have fixed storage requirements. However, error may be introduced each time a floating-point operation is performed, since the result may not be representable in the designated amount of memory.

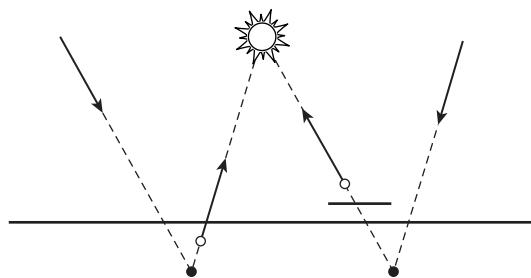
The accumulation of this error has a few implications for the accuracy of intersection tests. First, it's possible that it will cause valid intersections to be missed completely—for example, if a computed intersection's  $t$  value is negative even though the precise value is positive. Furthermore, computed ray–shape intersection points may be above or below the actual surface of the shape. This leads to a problem: when new rays are traced starting from computed intersection points for shadow rays and reflection rays, if the ray origin is below the actual surface, we may find an incorrect re-intersection with the surface. Conversely, if the origin is too far above the surface, shadows and reflections may appear detached. (See Figure 3.39.)

Typical practice to address this issue in ray tracing is to offset spawned rays by a fixed “ray epsilon” value, ignoring any intersections along the ray  $p + t\mathbf{d}$  closer than some  $t_{\min}$  value. Figure 3.40 shows why this approach requires fairly high  $t_{\min}$  values to work

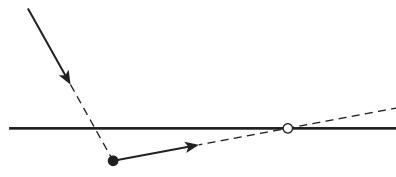
Float 1062

Pi 1063

Vector3f 60



**Figure 3.39: Geometric Settings for Rounding-Error Issues That Can Cause Visible Errors in Images.** The incident ray on the left intersects the surface. On the left, the computed intersection point (black circle) is slightly below the surface and a too-low “epsilon” offsetting the origin of the shadow ray leads to an incorrect self-intersection, as the shadow ray origin (white circle) is still below the surface; thus the light is incorrectly determined to be occluded. On the right a too-high “epsilon” causes a valid intersection to be missed as the ray’s origin is past the occluding surface.



**Figure 3.40:** If the computed intersection point (filled circle) is below the surface and the spawned ray is oblique, incorrect re-intersections may occur some distance from the ray origin (open circle). If a minimum  $t$  value along the ray is used to discard nearby intersections, a relatively large  $t_{\min}$  is needed to handle oblique rays well.

effectively: if the spawned ray is fairly oblique to the surface, incorrect ray intersections may occur quite some distance from the ray origin. Unfortunately, large  $t_{\min}$  values cause ray origins to be relatively far from the original intersection points, which in turn can cause valid nearby intersections to be missed, leading to loss of fine detail in shadows and reflections.

In this section, we’ll introduce the ideas underlying floating-point arithmetic and describe techniques for analyzing the error in floating-point computations. We’ll then apply these methods to the ray–shape algorithms introduced earlier in this chapter and show how to compute ray intersection points with bounded error. This will allow us to conservatively position ray origins so that incorrect self-intersections are never found, while keeping ray origins extremely close to the actual intersection point so that incorrect misses are minimized. In turn, no additional “ray epsilon” values are needed.

### 3.9.1 FLOATING-POINT ARITHMETIC

Computation must be performed on a finite representation of numbers that fits in a finite amount of memory; the infinite set of real numbers just can't be represented on a computer. One such finite representation is fixed point, where given a 16-bit integer, for example, one might map it to positive real numbers by dividing by 256. This would allow us to represent the range  $[0, 65535/256] = [0, 255 + 255/256]$  with equal spacing of  $1/256$  between values. Fixed-point numbers can be implemented efficiently using integer arithmetic operations (a property that made them popular on early PCs that didn't support floating-point computation), but they suffer from a number of shortcomings; among them, the maximum number they can represent is limited, and they aren't able to accurately represent very small numbers near zero.

An alternative representation for real numbers on computers is floating-point numbers. These are based on representing numbers with a sign, a significand,<sup>11</sup> and an exponent: essentially, the same representation as scientific notation but with a fixed number of digits devoted to significand and exponent. (In the following, we will assume base-2 digits exclusively.) This representation makes it possible to represent and perform computations on numbers with a wide range of magnitudes while using a fixed amount of storage.

Programmers using floating-point arithmetic are generally aware that floating point is imprecise; this understanding sometimes leads to a belief that floating-point arithmetic is unpredictable. In this section we'll see that floating-point arithmetic has a carefully designed foundation that in turn makes it possible to compute conservative bounds on the error introduced in a particular computation. For ray-tracing calculations, this error is often surprisingly small.

Modern CPUs and GPUs nearly ubiquitously implement a model of floating-point arithmetic based on a standard promulgated by the Institute of Electrical and Electronics Engineers (1985, 2008). (Henceforth when we refer to floats, we will specifically be referring to 32-bit floating-point numbers as specified by IEEE 754.) The IEEE 754 technical standard specifies the format of floating-point numbers in memory as well as specific rules for precision and rounding of floating-point computations; it is these rules that make it possible to reason rigorously about the error present in a given floating-point value.

#### Floating-Point Representation

The IEEE standard specifies that 32-bit floats are represented with a sign bit, 8 bits for the exponent, and 23 bits for the significand. With 8 bits, the exponent  $e_b$  ranges from 0 to 255; the actual exponent used,  $e_b$ , is computed by biasing  $e$ :

$$e_b = e - 127.$$

The significand actually has 24 bits of precision when a *normalized* floating-point value is stored. When a number expressed with significand and exponent is normalized, there

<sup>11</sup> The word *mantissa* is often used in place of *significand*, though floating-point purists note that *mantissa* has a different meaning in the context of logarithms and thus prefer *significand*. We follow this usage here.

are no leading zeros in the significand. In binary, this means that the leading digit of the significand must be one; in turn, there's no need to store this value explicitly. Thus, the implicit leading one digit with the 23 digits encoding the fractional part of the significand gives a total of 24 bits of precision.

Given a sign  $s = \pm 1$ , significand  $m$ , and exponent  $e$ , the corresponding floating-point value is

$$s \times 1.m \times 2^{e-127}.$$

For example, with a normalized significand, the floating-point number 6.5 is written as  $1.101_2 \times 2^2$ , where the 2 subscript denotes a base-2 value. (If binary decimals aren't immediately intuitive, note that the first number to the right of the decimal contributes  $2^{-1} = 1/2$ , and so forth.) Thus, we have

$$(1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}) \times 2^2 = 1.625 \times 2^2 = 6.5.$$

$e_b = 2$ , so  $e = 129 = 1000001_2$  and  $m = 10100000000000000000000000_2$ .

Floats are laid out in memory with the sign bit at the most significant bit of the 32-bit value (with negative signs encoded with a one bit), then the exponent, and the significand. Thus, for the value 6.5 the binary in-memory representation of the value is

$$0\ 10000001\ 10100000000000000000000000000000 = 40d00000_{16}.$$

Similarly the floating-point value 1.0 has  $m = 0 \dots 0_2$  and  $e_b = 0$ , so  $e = 127 = 01111111_2$  and its binary representation is:

$$0\ 01111111\ 00000000000000000000000000000000 = 3f800000_{16}.$$

This hexadecimal number is a value worth remembering, as it often comes up in memory dumps when debugging.

An implication of this representation is that the spacing between representable floats between two adjacent powers of two is uniform throughout the range. (It corresponds to increments of the significand bits by one.) In a range  $[2^e, 2^{e+1}]$ , the spacing is

$$2^{e-23}. \quad [3.6]$$

Thus, for floating-point numbers between 1 and 2,  $e = 0$ , and the spacing between floating-point values is  $2^{-23} \approx 1.19209 \dots \times 10^{-7}$ . This spacing is also referred to as the magnitude of a *unit in last place* (“ulp”); note that the magnitude of an ulp is determined by the floating-point value that it is with respect to—ulps are relatively larger at numbers with bigger magnitudes than they are at numbers with smaller magnitudes.

As we've described the representation so far, it's impossible to exactly represent zero as a floating-point number. This is obviously an unacceptable state of affairs, so the minimum exponent  $e = 0$ , or  $e_b = -127$ , is set aside for special treatment. With this exponent, the floating-point value is interpreted as not having the implicit leading one bit in the significand, which means that a significand of all zero bits results in

$$s \times 0.0 \dots 0_2 \times 2^{-127} = 0.$$

Eliminating the leading one significand bit also makes it possible to represent *denormalized* numbers: if the leading one was always present, then the smallest 32-bit float would be

$$1.0 \dots 0_2 \times 2^{-127} \approx 5.8774718 \times 10^{-39}.$$

Without the leading one bit, the minimum value is

$$0.00 \dots 1_2 \times 2^{-126} = 2^{-126} \times 2^{-23} \approx 1.4012985 \times 10^{-45}.$$

Providing some capability to represent these small values can make it possible to avoid needing to round very small values to zero.

Note that there is both a “positive” and “negative” zero value with this representation. This detail is mostly transparent to the programmer. For example, the standard guarantees that the comparison `-0.0 == 0.0` evaluates to true, even though the in-memory representations of these two values are different.

The maximum exponent,  $e = 255$ , is also reserved for special treatment. Therefore, the largest regular floating-point value that can be represented has  $e = 254$  (or  $e_b = 127$ ) and is approximately

$$3.402823 \dots \times 10^{38}.$$

With  $e_b = 255$ , if the significand bits are all zero, the value corresponds to positive or negative infinity, according to the sign bit. Infinite values result when performing computations like  $1/0$  in floating point, for example. Arithmetic operations with infinity result in infinity. For comparisons, positive infinity is larger than any non-infinite value and similarly for negative infinity.

The `MaxFloat` and `Infinity` constants are initialized to be the largest representable and “infinity” floating-point values, respectively. We make them available in separate constants so that code that uses these values doesn’t need to use the wordy C++ standard library calls to get their values.

*(Global Constants) ≡*

```
static constexpr Float MaxFloat = std::numeric_limits<Float>::max();
static constexpr Float Infinity = std::numeric_limits<Float>::infinity();
```

With  $e_b = 255$ , non-zero significand bits correspond to special NaN values, which result from operations like taking the square root of a negative number or trying to compute  $0/0$ . NaNs propagate through computations: any arithmetic operation where one of the operands is a NaN itself always returns NaN. Thus, if a NaN emerges from a long chain of computations, we know that something went awry somewhere along the way. In debug builds, pbrt has many `Assert()` statements that check for NaN values, as we almost never expect them to come up in the regular course of events. Any comparison with a NaN value returns false; thus, checking for `!(x == x)` serves to check if a value is not a

`Assert()` 1069

`Float` 1062

number.<sup>12</sup> For clarity, we use the C++ standard library function `std::isnan()` to check for not-a-number values.

### Utility Routines

For certain low-level operations, it can be useful to be able to interpret a floating-point value in terms of its constituent bits and to convert the bits representing a floating-point value to an actual `float` or `double`.

One natural approach to this would be to take a pointer to a value to be converted and cast it to a pointer to the other type:

```
float f = ...;
uint32_t bits = *((uint32_t *)&f);
```

However, modern versions of C++ specify that it's illegal to cast a pointer of one type, `float`, to a different type, `uint32_t`. (This restriction allows the compiler to optimize more aggressively in its analysis of whether two pointers may point to the same memory location, which can inhibit storing values in registers.)

Another common approach is to use a union with elements of both types, assigning to one type and reading from the other:

```
union FloatBits {
    float f;
    uint32_t ui;
};

FloatBits fb;
fb.f = ...;
uint32_t bits = fb.ui;
```

This, too, is illegal: the C++ standard says that reading an element of a union different from the one last one assigned to is undefined behavior.

These conversions can be properly made using `memcpy()` to copy from a pointer to the source type to a pointer to the destination type:

```
(Global Inline Functions) ≡
inline uint32_t FloatToBits(float f) {
    uint32_t ui;
    memcpy(&ui, &f, sizeof(float));
    return ui;
}
```

---

<sup>12</sup> This is one of a few places where compilers must not perform seemingly obvious and safe algebraic simplifications with expressions that include floating-point values—this particular comparison must not be simplified to `false`. Enabling compiler “fast math” or “perform unsafe math optimizations” flags may allow these optimizations to be performed. In turn, buggy behavior may be introduced in `pbrt`.

```
(Global Inline Functions) +≡
    inline float BitsToFloat(uint32_t ui) {
        float f;
        memcpy(&f, &ui, sizeof(uint32_t));
        return f;
    }
```

While a call to the `memcpy()` function may seem gratuitously expensive to avoid these issues, in practice good compilers turn this into a no-op and just reinterpret the contents of the register or memory as the other type. (Versions of these functions that convert between `double` and `uint64_t` are also available in `pbrt` but are similar and are therefore not included here.)

These conversions can be used to implement functions that bump a floating-point value up or down to the next greater or next smaller representable floating-point value.<sup>13</sup> They are useful for some conservative rounding operations that we'll need in code to follow. Thanks to the specifics of the in-memory representation of floats, these operations are quite efficient.

```
(Global Inline Functions) +≡
    inline float NextFloatUp(float v) {
        (Handle infinity and negative zero for NextFloatUp() 212)
        (Advance v to next higher float 213)
    }
```

There are two important special cases: if `v` is positive infinity, then this function just returns `v` unchanged. Negative zero is skipped forward to positive zero before continuing on to the code that advances the significand. This step must be handled explicitly, since the bit patterns for `-0.0` and `0.0` aren't adjacent.

```
(Handle infinity and negative zero for NextFloatUp()) ≡ 212
    if (std::isinf(v) && v > 0.)
        return v;
    if (v == -0.f)
        v = 0.f;
```

Conceptually, given a floating-point value, we would like to increase the significand by one, where if the result overflows, the significand is reset to zero and the exponent is increased by one. Fortunately, adding one to the in-memory integer representation of a float achieves this: because the exponent lies at the high bits above the significand, adding one to the low bit of the significand will cause a one to be carried all the way up into the exponent if the significand is all ones and otherwise will advance to the next higher significand for the current exponent. Note also that when the highest representable finite floating-point value's bit representation is incremented, the bit pattern for positive floating-point infinity is the result.

---

<sup>13</sup> These functions are equivalent to `std::nextafter(v, Infinity)` and `std::nextafter(v, -Infinity)`, but are more efficient since they don't try to handle NaN values or deal with signaling floating-point exceptions.

For negative values, subtracting one from the bit representation similarly advances to the next value.

```
(Advance v to next higher float) ≡
    uint32_t ui = FloatToBits(v);
    if (v >= 0) ++ui;
    else         --ui;
    return BitsToFloat(ui);
```

212

The `NextFloatDown()` function, not included here, follows the same logic but effectively in reverse. `pbrt` also provides versions of these functions for doubles.

### Arithmetic Operations

IEEE 754 provides important guarantees about the properties of floating-point arithmetic: specifically, it guarantees that addition, subtraction, multiplication, division, and square root give the same results given the same inputs and that these results are the floating-point number that is closest to the result of the underlying computation if it had been performed in infinite-precision arithmetic.<sup>14</sup> It is remarkable that this is possible on finite-precision digital computers at all; one of the achievements in IEEE 754 was the demonstration that this level of accuracy is possible and can be implemented fairly efficiently in hardware.

Using circled operators to denote floating-point arithmetic operations and `sqrt` for floating-point square root, these precision guarantees can be written as:

$$\begin{aligned} a \oplus b &= \text{round}(a + b) \\ a \ominus b &= \text{round}(a - b) \\ a \otimes b &= \text{round}(a * b) \\ a \oslash b &= \text{round}(a / b) \\ \text{sqrt}(a) &= \text{round}(\sqrt{a}) \end{aligned} \tag{3.7}$$

where  $\text{round}(x)$  indicates the result of rounding a real number to the closest floating-point value.

This bound on the rounding error can also be represented with an interval of real numbers: for example, for addition, we can say that the rounded result is within an interval

$$\begin{aligned} a \oplus b &= \text{round}(a + b) \subset (a + b)(1 \pm \epsilon) \\ &= [(a + b)(1 - \epsilon), (a + b)(1 + \epsilon)] \end{aligned} \tag{3.8}$$

for some  $\epsilon$ . The amount of error introduced from this rounding can be no more than half the floating-point spacing at  $a + b$ —if it was more than half the floating-point spacing, then it would be possible to round to a different floating-point number with less error (Figure 3.41).

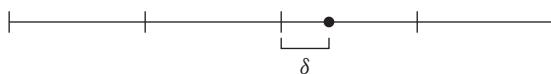
For 32-bit floats, we can bound the floating-point spacing at  $a + b$  from above using Equation (3.6) (i.e., an ulp at that value) by  $(a + b)2^{-23}$ , so half the spacing is bounded

`BitsToFloat()` 212

`FloatToBits()` 211

---

<sup>14</sup> IEEE float allows the user to select one of a number of rounding modes, but we will assume the default—round to nearest even—here.



**Figure 3.41:** The IEEE standard specifies that floating-point calculations must be implemented as if the calculation was performed with infinite-precision real numbers and then rounded to the nearest representable float. Here, an infinite precision result in the real numbers is denoted by a filled dot, with the representable floats around it denoted by ticks in a number line. We can see that the error introduced by rounding to the nearest float,  $\delta$ , can be no more than half the spacing between floats.

from above by  $(a + b)2^{-24}$  and so  $|\epsilon| \leq 2^{-24}$ . This bound is the *machine epsilon*.<sup>15</sup> For 32-bit floats,  $\epsilon_m = 2^{-24} \approx 5.960464 \dots \times 10^{-8}$ .

*(Global Constants) + ≡*

```
static constexpr Float MachineEpsilon =
    std::numeric_limits<Float>::epsilon() * 0.5;
```

Thus, we have

$$\begin{aligned} a \oplus b &= \text{round}(a + b) \subset (a + b)(1 \pm \epsilon_m) \\ &= [(a + b)(1 - \epsilon_m), (a + b)(1 + \epsilon_m)]. \end{aligned}$$

Analogous relations hold for the other arithmetic operators and the square root operator.<sup>16</sup>

A number of useful properties follow directly from Equation (3.7). For a floating-point number  $x$ ,

- $1 \otimes x = x$ .
- $x \oslash x = 1$ .
- $x \oplus 0 = x$ .
- $x \ominus x = 0$ .
- $2 \otimes x$  and  $x \oslash 2$  are exact; no rounding is performed to compute the final result. More generally, any multiplication by or division by a power of two gives an exact result (assuming there's no overflow or underflow).
- $x \oslash 2^i = x \otimes 2^{-i}$  for all integer  $i$ , assuming  $2^i$  doesn't overflow.

All of these properties follow from the principle that the result must be the nearest floating-point value to the actual result; when the result can be represented exactly, the exact result must be computed.

## Error Propagation

Using the guarantees of IEEE floating-point arithmetic, it is possible to develop methods to analyze and bound the error in a given floating-point computation. For more details

<sup>15</sup> The C and C++ standards unfortunately define the machine epsilon in their own special way, which is that it is the magnitude of one ulp above the number 1. For a 32-bit float, this value is  $2^{-23}$ , which is twice as large as the machine epsilon as the term is used in numerical analysis.

Float 1062

<sup>16</sup> This bound assumes that there's no overflow or underflow in the computation; these possibilities can be easily handled (Higham 2002, p. 56) but aren't generally important for our application here.

on this topic, see the excellent book by Higham (2002), as well as Wilkinson's earlier classic (1963).

Two measurements of error are useful in this effort: absolute and relative. If we perform some floating-point computation and get a rounded result  $\tilde{a}$ , we say that the magnitude of the difference between  $\tilde{a}$  and the result of doing that computation in the real numbers is the *absolute error*,  $\delta_a$ :

$$\delta_a = |\tilde{a} - a|.$$

*Relative error*,  $\delta_r$ , is the ratio of the absolute error to the precise result:

$$\delta_r = \left| \frac{\tilde{a} - a}{a} \right| = \left| \frac{\delta_a}{a} \right|, \quad [3.9]$$

as long as  $a \neq 0$ . Using the definition of relative error, we can thus write the computed value  $\tilde{a}$  as a perturbation of the exact result  $a$ :

$$\tilde{a} = a \pm \delta_a = a(1 \pm \delta_r).$$

As a first application of these ideas, consider computing the sum of four numbers,  $a$ ,  $b$ ,  $c$ , and  $d$ , represented as floats. If we compute this sum as  $r = (((a + b) + c) + d)$ , Equation (3.8) gives us

$$\begin{aligned} (((a \oplus b) \oplus c) \oplus d) &\subset (((a + b)(1 \pm \epsilon_m)) + c)(1 \pm \epsilon_m) + d)(1 \pm \epsilon_m) \\ &= (a + b)(1 \pm \epsilon_m)^3 + c(1 \pm \epsilon_m)^2 + d(1 \pm \epsilon_m). \end{aligned}$$

Because  $\epsilon_m$  is small, higher order powers of  $\epsilon_m$  can be bounded by an additional  $\epsilon_m$  term, and so we can bound the  $(1 \pm \epsilon_m)^n$  terms with

$$(1 \pm \epsilon_m)^n \leq (1 \pm (n+1)\epsilon_m).$$

(As a practical matter,  $(1 \pm n\epsilon_m)$  almost bounds these terms, since higher powers of  $\epsilon_m$  get very small very quickly, but the above is a fully conservative bound.)

This bound lets us simplify the result of the addition to:

$$\begin{aligned} (a + b)(1 \pm 4\epsilon_m) + c(1 \pm 3\epsilon_m) + d(1 \pm 2\epsilon_m) &= \\ a + b + c + d + [\pm 4\epsilon_m(a + b) \pm 3\epsilon_m c \pm 2\epsilon_m d]. \end{aligned}$$

The term in square brackets gives the absolute error: its magnitude is bounded by

$$4\epsilon_m|a + b| + 3\epsilon_m|c| + 2\epsilon_m|d|. \quad [3.10]$$

Thus, if we add four floating-point numbers together with the above parenthesization, we can be certain that the difference between the final rounded result and the result we would get if we added them with infinite-precision real numbers is bounded by Equation (3.10); this error bound is easily computed given specific values of  $a$ ,  $b$ ,  $c$ , and  $d$ .

This is a fairly interesting result; we see that the magnitude of  $a + b$  makes a relatively large contribution to the error bound, especially compared to  $d$ . (This result gives a sense for why, if adding a large number of floating-point numbers together, sorting them from

small to large magnitudes generally gives a result with a lower final error than an arbitrary ordering.)

Our analysis here has implicitly assumed that the compiler would generate instructions according to the expression used to define the sum. Compilers are required to follow the form of the given floating-point expressions in order to not break carefully crafted computations that may have been designed to minimize round-off error. Here again is a case where certain transformations that would be valid on expressions with integers can not be safely applied when floats are involved.

What happens if we change the expression to the algebraically equivalent  $\text{float } r = (a + b) + (c + d)$ ? This corresponds to the floating-point computation

$$((a \oplus b) \oplus (c \oplus d)).$$

If we apply the same process of applying Equation (3.8), expanding out terms, converting higher-order  $(1 \pm \epsilon_m)^n$  terms to  $(1 \pm (n+1)\epsilon_m)$ , we get absolute error bounds of

$$3\epsilon_m|a + b| + 3\epsilon_m|c + d|,$$

which are lower than the first formulation if  $|a + b|$  is relatively large, but possibly higher if  $|d|$  is relatively large.

This approach to computing error is known as *forward error analysis*; given inputs to a computation, we can apply a fairly mechanical process that provides conservative bounds on the error in the result. The derived bounds in the result may overstate the actual error—in practice, the signs of the error terms are often mixed, so that there is cancellation when they are added.<sup>17</sup> An alternative approach is *backward error analysis*, which treats the computed result as exact and provides bounds on perturbations on the inputs that give the same result. This approach can be more useful when analyzing the stability of a numerical algorithm but is less applicable to deriving conservative error bounds on the geometric computations we’re interested in here.

The conservative bounding of  $(1 \pm \epsilon_m)^n$  by  $(1 \pm (n+1)\epsilon_m)$  is somewhat unsatisfying since it adds a whole  $\epsilon_m$  term purely to conservatively bound the sum of various higher powers of  $\epsilon_m$ . Higham (2002, Section 3.1) gives an approach to more tightly bound products of  $(1 \pm \epsilon_m)$  error terms. If we have  $(1 \pm \epsilon_m)^n$ , it can be shown that this value is bounded by  $1 + \theta_n$ , where

$$|\theta_n| \leq \frac{n \epsilon_m}{1 - n \epsilon_m}, \quad [3.11]$$

as long as  $n \epsilon_m < 1$  (which will certainly be the case for the calculations we’re considering). Note that the denominator of this expression will be just less than one for reasonable  $n$  values, so it just barely increases  $n\epsilon_m$  to achieve a conservative bound.

We will denote this bound by  $\gamma_n$ :

$$\gamma_n = \frac{n \epsilon_m}{1 - n \epsilon_m}.$$

<sup>17</sup> Some numerical analysts use a rule of thumb that the number of ulps of error in practice is often close to the square root of the bound’s number of ulps, thanks to the cancellation of error in intermediate results.

The function that computes its value is declared as `constexpr` so that any invocations with compile-time constants will generally be replaced with the corresponding floating-point return value.

```
<Global Inline Functions> +≡
    inline constexpr Float gamma(int n) {
        return (n * MachineEpsilon) / (1 - n * MachineEpsilon);
    }
```

Using the  $\gamma$  notation, our bound on the error of the sum of the four values is

$$|a + b|\gamma_3 + |c|\gamma_2 + |d|\gamma_1.$$

An advantage of this approach is that quotients of  $(1 \pm \epsilon_m)^n$  terms can also be bounded with the  $\gamma$  function. Given

$$\frac{(1 \pm \epsilon_m)^m}{(1 \pm \epsilon_m)^n},$$

the interval is bounded by  $(1 \pm \gamma_{m+n})$ . Thus,  $\gamma$  can be used to collect  $\epsilon_m$  terms from both sides of an equality over to one side by dividing them through; this will be useful in some of the following derivations. (Note that because  $(1 \pm \epsilon_m)$  terms represent intervals, canceling them would be incorrect:

$$\frac{(1 \pm \epsilon_m)^m}{(1 \pm \epsilon_m)^n} \neq (1 \pm \epsilon_m)^{m-n};$$

the  $\gamma_{m+n}$  bounds must be used instead.)

Given inputs to some computation that themselves carry some amount of error, it's instructive to see how this error is carried through various elementary arithmetic operations. Given two values,  $a(1 \pm \gamma_i)$  and  $b(1 \pm \gamma_j)$  that each carry accumulated error from earlier operations, consider their product. Using the definition of  $\otimes$ , the result is in the interval:

$$a(1 \pm \gamma_i) \otimes b(1 \pm \gamma_j) \subset ab(1 \pm \gamma_{i+j+1}),$$

where we've used the relationship  $(1 \pm \gamma_i)(1 \pm \gamma_j) \subset (1 \pm \gamma_{i+j})$ , which follows directly from Equation (3.11).

The relative error in this result is bounded by:

$$\left| \frac{ab \gamma_{i+j+1}}{ab} \right| = \gamma_{i+j+1},$$

and so the final error is thus just roughly  $(i + j + 1)/2$  ulps at the value of the product—about as good as we might hope for given the error going into the multiplication. (The situation for division is similarly good.)

Unfortunately, with addition and subtraction, it's possible for the relative error to increase substantially. Using the same definitions of the values being operated on, consider

$$a(1 \pm \gamma_i) \oplus b(1 \pm \gamma_j),$$

which is in the interval  $a(1 \pm \gamma_{i+1}) + b(1 \pm \gamma_{j+1})$ , and so the absolute error is bounded by  $|a|\gamma_{i+1} + |b|\gamma_{j+1}$ .

If the signs of  $a$  and  $b$  are the same, then the absolute error is bounded by  $|a + b|\gamma_{i+j+1}$  and the relative error is around  $(i + j + 1)/2$  ulps around the computed value.

However, if the signs of  $a$  and  $b$  differ (or, equivalently, they are the same but subtraction is performed), then the relative error can be quite high. Consider the case where  $a \approx -b$ : the relative error is

$$\frac{|a|\gamma_{i+1} + |b|\gamma_{j+1}}{a + b} \approx \frac{2|a|\gamma_{i+j+1}}{a + b}.$$

The numerator's magnitude is proportional to the original value  $|a|$  yet is divided by a very small number, and thus the relative error is quite high. This substantial increase in relative error is called *catastrophic cancellation*. Equivalently, we can have a sense of the issue from the fact that the absolute error is in terms of the magnitude of  $|a|$ , though it's now in relation to a value much smaller than  $a$ .

### Running Error Analysis

In addition to working out error bounds algebraically, we can also have the computer do this work for us as some computation is being performed. This approach is known as *running error analysis*. The idea behind it is simple: each time a floating-point operation is performed, we also compute terms that compute intervals based on Equation (3.7) to compute a running bound on the error that has been accumulated so far. While this approach can have higher run-time overhead than deriving expressions that give an error bound directly, it can be convenient when derivations become unwieldy.

`pbrt` provides a simple `EFloat` class, which mostly acts like a regular `float` but uses operator overloading to provide all of the regular arithmetic operations on floats while computing these error bounds.

Similar to the `Interval` class from Chapter 2, `EFloat` keeps track of an interval that describes the uncertainty of a value of interest. In contrast to `Interval`, `EFloat`'s intervals arise due to errors in intermediate floating-point arithmetic rather than uncertainty of the input parameters.

```
(EFloat Public Methods) ≡
EFloat() { }
EFloat(float v, float err = 0.f) : v(v), err(err) {
    (Store high-precision reference value in EFloat 219)
}
```

`EFloat` maintains a computed value `v` and the absolute error bound, `err`.

```
(EFloat Private Data) ≡
float v;
float err;
```

In debug builds, `EFloat` also maintains a highly precise version of `v` that can be used as a reference value to compute an accurate approximation of the relative error. In optimized builds, we'd generally rather not pay the overhead for computing this additional value.

`EFloat` 218

`Interval` 112

```

⟨Store high-precision reference value in EFloat⟩ ≡
#ifndef NDEBUG
ld = v;
#endif // NDEBUG

⟨EFloat Private Data⟩ +≡
#ifndef NDEBUG
long double ld;
#endif // NDEBUG

```

The implementation of the addition operation for this class is essentially an implementation of the relevant definitions. We have:

$$\begin{aligned}
(a \pm \delta_a) \oplus (b \pm \delta_b) &= ((a \pm \delta_a) + (b \pm \delta_b))(1 \pm \gamma_1) \\
&= a + b + [\pm \delta_a \pm \delta_b \pm (a + b)\gamma_1 \pm \gamma_1 \delta_a \pm \gamma_1 \delta_b].
\end{aligned}$$

And so the absolute error (in brackets) is bounded by

$$\delta_a + \delta_b + \gamma_1(|a + b| + \delta_a + \delta_b).$$

```

⟨EFloat Public Methods⟩ +≡
EFloat operator+(EFloat f) const {
    EFloat r;
    r.v = v + f.v;
#ifndef NDEBUG
    r.ld = ld + f.ld;
#endif // DEBUG
    r.err = err + f.err +
        gamma(1) * (std::abs(v + f.v) + err + f.err);
    return r;
}

```

The implementations for the other arithmetic operations for EFloat are analogous.

Note that this implementation neglects the issue that the computation of errors will itself be affected by rounding error. If this was a concern, we could switch the floating-point rounding mode so that it always rounded the error bounds up to positive infinity, but this tends to be a fairly expensive operation since it causes a full pipeline flush on current processors. Here, we use the default rounding mode; in the following, the error bounds are expanded by one ulp when they are used to account for this issue.

The `float` value in an EFloat is available via a type conversion operator; it has an explicit qualifier to require the caller to have an explicit (`float`) cast to extract the floating-point value. The requirement to use an explicit cast reduces the risk of an unintended round trip from EFloat to Float and back, thus losing the accumulated error bounds.

```

EFloat 218
EFloat::err 218
EFloat::ld 219
EFloat::v 218

```

```

⟨EFloat Public Methods⟩ +≡
explicit operator float() const { return v; }

```

If a series of computations is performed using EFloat rather than `float`-typed variables, then at any point in the computation, the `GetAbsoluteError()` method can be called to find a bound on the absolute error of the computed value.

```
{EFloat Public Methods} +≡
float GetAbsoluteError() const { return err; }
```

The bounds of the error interval are available via the `UpperBound()` and `LowerBound()` methods. Their implementations use `NextFloatUp()` and `NextFloatDown()` to expand the returned values by one ulp, respectively, ensuring that the interval is conservative.

```
{EFloat Public Methods} +≡
float UpperBound() const { return NextFloatUp(v + err); }
float LowerBound() const { return NextFloatDown(v - err); }
```

In debug builds, methods are available to get both the relative error as well as the precise value maintained in `ld`.

```
{EFloat Public Methods} +≡
#ifndef NDEBUG
float GetRelativeError() const { return std::abs((ld - v)/ld); }
long double PreciseValue() const { return ld; }
#endif
```

`pbrt` also provides a variant of the `Quadratic()` function that operates on coefficients that may have error and returns error bounds with the `t0` and `t1` values. The implementation is the same as the regular `Quadratic()` function, just using `EFloat`.

```
{EFloat Inline Functions} ≡
inline bool Quadratic(EFloat A, EFloat B, EFloat C,
                      EFloat *t0, EFloat *t1);
```

With the floating-point error fundamentals in place, we'll now focus on using these tools to provide robust intersection operations.

### 3.9.2 CONSERVATIVE RAY-BOUNDS INTERSECTIONS

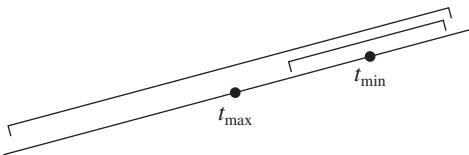
Floating-point round-off error can cause the ray–bounding box intersection test to miss cases where a ray actually does intersect the box. While it's acceptable to have occasional false positives from ray–box intersection tests, we'd like to never miss an actual intersection—getting this right is important for the correctness of the `BVHAccel` acceleration data structure in Section 4.3 so that valid ray–shape intersections aren't missed.

The ray–bounding box test introduced in Section 3.1.2 is based on computing a series of ray–slab intersections to find the parametric  $t_{\min}$  along the ray where the ray enters the bounding box and the  $t_{\max}$  where it exits. If  $t_{\min} < t_{\max}$ , the ray passes through the box; otherwise it misses it. With floating-point arithmetic, there may be error in the computed  $t$  values—if the computed  $t_{\min}$  value is greater than  $t_{\max}$  purely due to round-off error, the intersection test will incorrectly return a false result.

Recall that the computation to find the  $t$  value for a ray intersection with a plane perpendicular to the  $x$  axis at a point  $x$  is  $t = (x - o_x)/d_x$ . Expressed as a floating-point computation and applying Equation (3.7), we have

$$t = (x \ominus o_x) \otimes (1 \oslash d_x) \subset \frac{x - o_x}{d_x} (1 \pm \epsilon)^3,$$

`BVHAccel` 256  
`EFloat` 218  
`EFloat::err` 218  
`EFloat::v` 218  
`NextFloatDown()` 213  
`NextFloatUp()` 212



**Figure 3.42:** If the error bounds of the computed  $t_{\min}$  and  $t_{\max}$  values overlap, the comparison  $t_{\min} < t_{\max}$  may not actually indicate if a ray hit a bounding box. It's better to conservatively return true in this case than to miss an actual intersection. Extending  $t_{\max}$  by twice its error bound ensures that the comparison is conservative.

and so

$$t(1 \pm \gamma_3) = \frac{x - o_x}{d_x}.$$

The difference between the computed result  $t$  and the precise result is bounded by  $\gamma_3|t|$ .

If we consider the intervals around the computed  $t$  values that bound the fully precise value of  $t$ , then the case we're concerned with is when the intervals overlap; if they don't, then the comparison of computed values will give the correct result (Figure 3.42). If the intervals do overlap, it's impossible to know the actual ordering of the  $t$  values. In this case, increasing  $t_{\max}$  by twice the error bound,  $2\gamma_3 t_{\max}$ , before performing the comparison ensures that we conservatively return true in this case.

We can now define the fragment for the ray–bounding box test in Section 3.1.2 that makes this adjustment.

*(Update tFar to ensure robust ray–bounds intersection)* ≡  
`tFar *= 1 + 2 * gamma(3);`

128

The fragments for the `Bounds3::IntersectP()` method, *(Update tMax and tyMax to ensure robust bounds intersection)* and *(Update tzMax to ensure robust bounds intersection)*, are similar and therefore not included here.

### 3.9.3 ROBUST TRIANGLE INTERSECTIONS

The details of the ray–triangle intersection algorithm in Section 3.6.2 were carefully designed to avoid cases where rays could incorrectly pass through an edge or vertex shared by two adjacent triangles without generating an intersection. Fittingly, an intersection algorithm with this guarantee is referred to as being *watertight*.

Recall that the algorithm is based on transforming triangle vertices into a coordinate system with the ray's origin at its origin and the ray's direction aligned along the  $+z$  axis. Although round-off error may be introduced by transforming the vertex positions to this coordinate system, this error doesn't affect the watertightness of the intersection test, since the same transformation is applied to all triangles. (Further, this error is quite small, so it doesn't significantly impact the accuracy of the computed intersection points.)

Given vertices in this coordinate system, the three edge functions defined in Equation (3.1) are evaluated at the point  $(0, 0)$ ; the corresponding expressions, Equation (3.2), are quite straightforward. The key to the robustness of the algorithm is that with floating-point arithmetic, the edge function evaluations are guaranteed to have the correct sign. In general, we have

$$(a \otimes b) \ominus (c \otimes d). \quad [3.12]$$

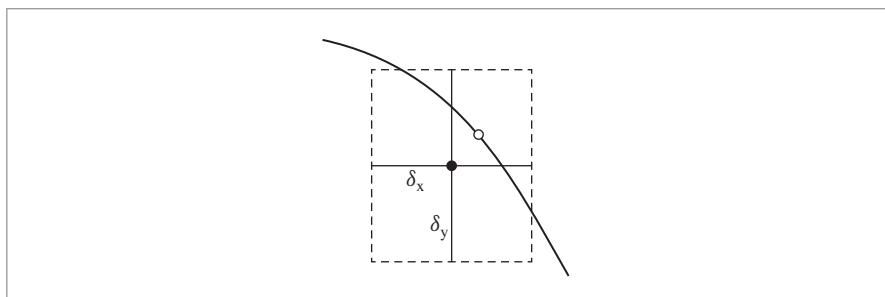
First, note that if  $ab = cd$ , then Equation (3.12) evaluates to exactly zero, even in floating point. We therefore just need to show that if  $ab > cd$ , then  $(a \otimes b) \ominus (c \otimes d)$  is never negative. If  $ab > cd$ , then  $(a \otimes b)$  must be greater than or equal to  $(c \otimes d)$ . In turn, their difference must be greater than or equal to zero. (These properties both follow from the fact that floating-point arithmetic operations are all rounded to the nearest representable floating-point value.)

If the value of the edge function is zero, then it's impossible to tell whether it is exactly zero or whether a small positive or negative value has rounded to zero. In this case, the fragment (*Fall back to double-precision test at triangle edges*) reevaluates the edge function with double precision; it can be shown that doubling the precision suffices to accurately distinguish these cases, given 32-bit floats as input.

The overhead caused by this additional precaution is minimal: in a benchmark with 88 million ray intersection tests, the double-precision fallback had to be used in less than 0.0000023% of the cases.

### 3.9.4 BOUNDING INTERSECTION POINT ERROR

We'll now apply this machinery for analyzing rounding error to derive conservative bounds on the absolute error in computed ray-shape intersection points, which allows us to construct bounding boxes that are guaranteed to include an intersection point on the actual surface (Figure 3.43). These bounding boxes provide the basis of the algorithm for generating spawned ray origins that will be introduced in Section 3.9.5.



**Figure 3.43:** Shape intersection algorithms in pbrt compute an intersection point, shown here in the 2D setting with a filled circle. The absolute error in this point is bounded by  $\delta_x$  and  $\delta_y$ , giving a small box around the point. Because these bounds are conservative, we know that the actual intersection point on the surface (open circle) must lie somewhere within the box.

It's useful to start by looking at the sources of error in conventional approaches to computing intersection points. It is common practice in ray tracing to compute 3D intersection points by first solving the parametric ray equation  $\mathbf{o} + t\mathbf{d}$  for a value  $t_{\text{hit}}$  where a ray intersects a surface and then computing the hit point  $\mathbf{p}$  with  $\mathbf{p} = \mathbf{o} + t_{\text{hit}}\mathbf{d}$ . If  $t_{\text{hit}}$  carries some error  $\delta_t$ , then we can bound the error in the computed intersection point. Considering the  $x$  coordinate, for example, we have

$$\begin{aligned} x &= \mathbf{o}_x \oplus (t_{\text{hit}} \pm \delta_t) \otimes \mathbf{d}_x \\ &\subset \mathbf{o}_x \oplus (t_{\text{hit}} \pm \delta_t)\mathbf{d}_x(1 \pm \gamma_1) \\ &\subset \mathbf{o}_x(1 \pm \gamma_1) + (t_{\text{hit}} \pm \delta_t)\mathbf{d}_x(1 \pm \gamma_2) \\ &= \mathbf{o}_x + t_{\text{hit}}\mathbf{d}_x + [\pm \mathbf{o}_x\gamma_1 \pm \delta_t\mathbf{d}_x \pm t_{\text{hit}}\mathbf{d}_x\gamma_2 \pm \delta_t\mathbf{d}_x\gamma_2]. \end{aligned}$$

The error term (in square brackets) is bounded by

$$\gamma_1|\mathbf{o}_x| + \delta_t(1 \pm \gamma_2)|\mathbf{d}_x| + \gamma_2|t_{\text{hit}}\mathbf{d}_x|. \quad [3.13]$$

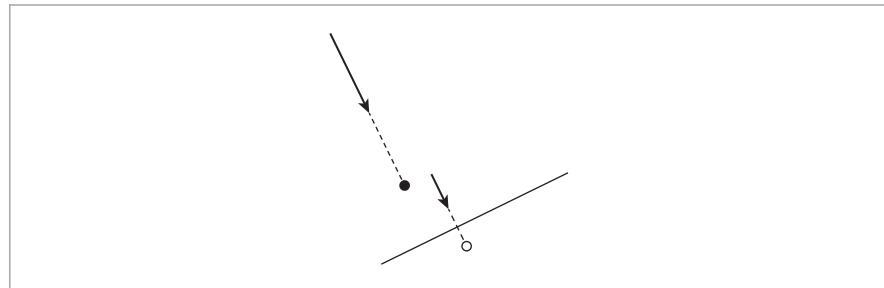
There are two things to see from Equation (3.13): first, the magnitudes of the terms that contribute to the error in the computed intersection point ( $\mathbf{o}_x$ ,  $\mathbf{d}_x$ , and  $t_{\text{hit}}\mathbf{d}_x$ ) may be quite different from the magnitude of the intersection point. Thus, there is a danger of catastrophic cancellation in the intersection point's computed value. Second, ray intersection algorithms generally perform tens of floating-point operations to compute  $t$  values, which in turn means that we can expect  $\delta_t$  to be at least of magnitude  $\gamma_n t$ , with  $n$  in the tens (and possibly much more, due to catastrophic cancellation). Each of these terms may be significant with respect to the magnitude of the computed point  $x$ .

Together, these factors can lead to relatively large error in the computed intersection point. We'll develop better approaches shortly.

### Reprojection: Quadrics

We'd like to reliably compute intersection points on surfaces with just a few ulps of error rather than the hundreds of ulps of error that intersection points computed with the parametric ray equation may have. Previously, Woo et al. (1996) suggested using the first intersection point computed as a starting point for a second ray–plane intersection, for ray–polygon intersections. From the bounds in Equation (3.13), we can see why the second intersection point will be much closer to the surface than the first: the  $t_{\text{hit}}$  value along the second ray will be quite close to zero, so that the magnitude of the absolute error in  $t_{\text{hit}}$  will be quite small, and thus using this value in the parametric ray equation will give a point quite close to the surface (Figure 3.44). Further, the ray origin will have similar magnitude to the intersection point, so the  $\gamma_1|\mathbf{o}_x|$  term won't introduce much additional error.

Although the second intersection point computed with this approach is much closer to the plane of the surface, it still suffers from error by being offset due to error in the first computed intersection. The farther away the ray origin from the intersection point (and thus, the larger the absolute error in  $t_{\text{hit}}$ ), the larger this error will be. In spite of this error, the approach has merit: we're generally better off with a computed intersection point that is quite close to the actual surface, even if offset from the most accurate possible intersection point, than we are with a point that is some distance above or below the surface (and likely also far from the most accurate intersection point).



**Figure 3.44: Re-intersection to Improve the Accuracy of the Computed Intersection Point.**

Given a ray and a surface, an initial intersection point has been computed with the ray equation (filled circle). This point may be fairly inaccurate due to rounding error but can be used as the origin for a second ray–shape intersection. The intersection point computed from this second intersection (open circle) is much closer to the surface, though it may be shifted from the true intersection point due to error in the first computed intersection.

Rather than doing a full re-intersection computation, which may not only be computationally costly but also will still have error in the computed  $t$  value, an effective approach is to refine computed intersection points by reprojecting them to the surface. The error bounds for these reprojected points are often remarkably small.

It should be noted that these reprojection error bounds don't capture tangential errors that were present in the original intersection  $p$ —the main focus here is to detect errors that might cause the reprojected point  $p'$  to fall below the surface.

Consider a ray–sphere intersection: given a computed intersection point (e.g., from the ray equation)  $p$  with a sphere at the origin with radius  $r$ , we can reproject the point onto the surface of the sphere by scaling it with the ratio of the sphere's radius to the computed point's distance to the origin, computing a new point  $p' = (x', y', z')$  with

$$x' = x \frac{r}{\sqrt{x^2 + y^2 + z^2}},$$

and so forth. The floating-point computation is

$$\begin{aligned} x' &= x \otimes r \oslash \text{sqrt}((x \otimes x) \oplus (y \otimes y) \oplus (z \otimes z)) \\ &\subset \frac{xr(1 \pm \epsilon_m)^2}{\sqrt{x^2(1 \pm \epsilon_m)^3 + y^2(1 \pm \epsilon_m)^3 + z^2(1 \pm \epsilon_m)^2(1 \pm \epsilon_m)}} \\ &\subset \frac{xr(1 \pm \gamma_2)}{\sqrt{x^2(1 \pm \gamma_3) + y^2(1 \pm \gamma_3) + z^2(1 \pm \gamma_2)}(1 \pm \gamma_1)} \end{aligned}$$

Because  $x^2$ ,  $y^2$ , and  $z^2$  are all positive, the terms in the square root can share the same  $\gamma$  term, and we have

$$\begin{aligned}
 x' &\subset \frac{xr(1 \pm \gamma_2)}{\sqrt{(x^2 + y^2 + z^2)(1 \pm \gamma_4)}(1 \pm \gamma_1)} \\
 &= \frac{xr(1 \pm \gamma_2)}{\sqrt{(x^2 + y^2 + z^2)}\sqrt{(1 \pm \gamma_4)}(1 \pm \gamma_1)} \\
 &\subset \frac{xr}{\sqrt{(x^2 + y^2 + z^2)}}(1 \pm \gamma_5) \\
 &= x'(1 \pm \gamma_5).
 \end{aligned} \tag{3.14}$$

Thus, the absolute error of the reprojected  $x$  coordinate is bounded by  $\gamma_5|x'|$  (and similarly for  $y'$  and  $z'$ ) and is thus no more than 2.5 ulps in each dimension from a point on the surface of the sphere.

Here is the fragment that reprojects the intersection point for the Sphere shape.

```
<Refine sphere intersection point> ≡
    pHit *= radius / Distance(pHit, Point3f(0, 0, 0));
```

137

The error bounds follow from Equation (3.14).

```
<Compute error bounds for sphere intersection> ≡
    Vector3f pError = gamma(5) * Abs((Vector3f)pHit);
```

134

Reprojection algorithms and error bounds for other quadrics can be defined similarly: for example, for a cylinder along the  $z$  axis, only the  $x$  and  $y$  coordinates need to be re-projected, and the error bounds in  $x$  and  $y$  turn out to be only  $\gamma_3$  times their magnitudes.

```
<Refine cylinder intersection point> ≡
    Float hitRad = std::sqrt(pHit.x * pHit.x + pHit.y * pHit.y);
    pHit.x *= radius / hitRad;
    pHit.y *= radius / hitRad;
```

145

```
<Compute error bounds for cylinder intersection> ≡
    Vector3f pError = gamma(3) * Abs(Vector3f(pHit.x, pHit.y, 0));
```

144

The disk shape is particularly easy; we just need to set the  $z$  coordinate of the point to lie on the plane of the disk.

```
<Refine disk intersection point> ≡
    pHit.z = height;
```

148

In turn, we have a point with zero error; it lies exactly on the surface on the disk.

```
<Compute error bounds for disk intersection> ≡
    Vector3f pError(0, 0, 0);
```

148

### Parametric Evaluation: Triangles

Another effective approach to computing precise intersection points is to use the parametric representation of a shape to compute accurate intersection points. For example, the triangle intersection algorithm in Section 3.6.2 computes three edge function values  $e_0$ ,  $e_1$ , and  $e_2$  and reports an intersection if all three have the same sign. Their values can

```
Cylinder::radius 143
Disk::height 147
Float 1062
gamma() 217
Point3f 68
Sphere 133
Sphere::radius 133
Vector3f 60
```

be used to find the barycentric coordinates

$$b_i = \frac{e_i}{e_0 + e_1 + e_2}.$$

Attributes  $v_i$  at the triangle vertices (including the vertex positions) can be interpolated across the face of the triangle by

$$v' = b_0 v_0 + b_1 v_1 + b_2 v_2.$$

We can show that interpolating the positions of the vertices in this manner gives a point very close to the surface of the triangle. First consider precomputing the inverse sum of  $e_i$ :

$$\begin{aligned} d &= 1 \oslash (e_0 \oplus e_1 \oplus e_2) \\ &\subset \frac{1}{(e_0 + e_1)(1 \pm \epsilon_m)^2 + e_2(1 \pm \epsilon_m)} (1 \pm \epsilon_m). \end{aligned}$$

Because all  $e_i$  have the same sign if there is an intersection, we can collect the  $e_i$  terms and conservatively bound  $d$ :

$$\begin{aligned} d &\subset \frac{1}{(e_0 + e_1 + e_2)(1 \pm \epsilon_m)^2} (1 \pm \epsilon_m) \\ &\subset \frac{1}{e_0 + e_1 + e_2} (1 \pm \gamma_3). \end{aligned}$$

If we now consider interpolation of the  $x$  coordinate of the position in the triangle corresponding to the edge function values, we have

$$\begin{aligned} x' &= ((e_0 \otimes x_0) \oplus (e_1 \otimes x_1) \oplus (e_2 \otimes x_2)) \otimes d \\ &\subset (e_0 x_0 (1 \pm \epsilon_m)^3 + e_1 x_1 (1 \pm \epsilon_m)^3 + e_2 x_2 (1 \pm \epsilon_m)^2) d (1 \pm \epsilon_m) \\ &\subset (e_0 x_0 (1 \pm \gamma_4) + e_1 x_1 (1 \pm \gamma_4) + e_2 x_2 (1 \pm \gamma_3)) d. \end{aligned}$$

Using the bounds on  $d$ ,

$$\begin{aligned} x &\subset \frac{e_0 x_0 (1 \pm \gamma_7) + e_1 x_1 (1 \pm \gamma_7) + e_2 x_2 (1 \pm \gamma_6)}{e_0 + e_1 + e_2} \\ &= b_0 x_0 (1 \pm \gamma_7) + b_1 x_1 (1 \pm \gamma_7) + b_2 x_2 (1 \pm \gamma_6). \end{aligned}$$

Thus, we can finally see that the absolute error in the computed  $x'$  value is in the interval

$$\pm b_0 x_0 \gamma_7 \pm b_1 x_1 \gamma_7 \pm b_2 x_2 \gamma_7,$$

which is bounded by

$$\gamma_7 (|b_0 x_0| + |b_1 x_1| + |b_2 x_2|). \quad [3.15]$$

(Note that the  $b_2 x_2$  term could have a  $\gamma_6$  factor instead of  $\gamma_7$ , but the difference between the two is very small so we choose a slightly simpler final expression.) Equivalent bounds hold for  $y'$  and  $z'$ .

`Triangle::Intersect()` 157

Equation (3.15) lets us bound the error in the interpolated point computed in `Triangle::Intersect()`.

*(Compute error bounds for triangle intersection) ≡*

```
Float xAbsSum = (std::abs(b0 * p0.x) + std::abs(b1 * p1.x) +
                 std::abs(b2 * p2.x));
Float yAbsSum = (std::abs(b0 * p0.y) + std::abs(b1 * p1.y) +
                 std::abs(b2 * p2.y));
Float zAbsSum = (std::abs(b0 * p0.z) + std::abs(b1 * p1.z) +
                 std::abs(b2 * p2.z));
Vector3f pError = gamma(7) * Vector3f(xAbsSum, yAbsSum, zAbsSum);
```

157

## Other Shapes

For shapes where we may not want to derive reprojection methods and tight error bounds, running error analysis can be quite useful: we implement all of the intersection calculations using EFloat instead of Float, compute a  $t_{hit}$  value, and use the parametric ray equation to compute a hit point. We can then find conservative bounds on the error in the computed intersection point via the EFloat GetAbsoluteError() method.

*(Compute error bounds for intersection computed with ray equation) ≡*

```
EFloat px = ox + tShapeHit * dx;
EFloat py = oy + tShapeHit * dy;
EFloat pz = oz + tShapeHit * dz;
Vector3f pError = Vector3f(px.GetAbsoluteError(), py.GetAbsoluteError(),
                           pz.GetAbsoluteError());
```

227

This approach is used for cones, paraboloids, and hyperboloids in pbrt.

*(Compute error bounds for cone intersection) ≡*

*(Compute error bounds for intersection computed with ray equation 227)*

Because the Curve shape orients itself to face incident rays, rays leaving it must be offset by twice the curve's width in order to not incorrectly re-intersect it when it's reoriented to face them.

*(Compute error bounds for curve intersection) ≡*

```
Vector3f pError(2 * hitWidth, 2 * hitWidth, 2 * hitWidth);
```

180

## Effect of Transformations

The last detail to attend to in order to bound the error in computed intersection points is the effect of transformations, which introduce additional rounding error when they are applied to computed intersection points.

The quadric Shapes in pbrt transform world space rays into object space before performing ray–shape intersections and then transform computed intersection points back to world space. Both of these transformation steps introduce rounding error that needs to be accounted for in order to maintain robust world space bounds around intersection points.

If possible, it's best to try to avoid coordinate-system transformations of rays and intersection points. For example, it's better to transform triangle vertices to world space and intersect world space rays with them than to transform rays to object space and then

Curve 168

EFloat 218

Float 1062

gamma() 217

Vector3f 60

transform intersection points to world space.<sup>18</sup> Transformations are still useful—for example, for the quadrics and for object instancing, so we'll show how to bound the error that they introduce.

We'll start by considering the error introduced by transforming a point  $(x, y, z)$  that is exact—i.e., without any accumulated error. Given a  $4 \times 4$  non-projective transformation matrix with elements denoted by  $m_{i,j}$ , the transformed point  $x'$  is

$$\begin{aligned} x' &= ((m_{0,0} \otimes x) \oplus (m_{0,1} \otimes y)) \oplus ((m_{0,2} \otimes z) \oplus m_{0,3}) \\ &\subset m_{0,0}x(1 \pm \epsilon_m)^3 + m_{0,1}y(1 \pm \epsilon_m)^3 + m_{0,2}z(1 \pm \epsilon_m)^3 + m_{0,3}(1 \pm \epsilon_m)^2 \\ &\subset (m_{0,0}x + m_{0,1}y + m_{0,2}z + m_{0,3}) + \gamma_3(\pm m_{0,0}x \pm m_{0,1}y \pm m_{0,2}z \pm m_{0,3}) \\ &\subset (m_{0,0}x + m_{0,1}y + m_{0,2}z + m_{0,3}) \pm \gamma_3(|m_{0,0}x| + |m_{0,1}y| + |m_{0,2}z| + |m_{0,3}|). \end{aligned}$$

Thus, the absolute error in the result is bounded by

$$\gamma_3(|m_{0,0}x| + |m_{0,1}y| + |m_{0,2}z| + |m_{0,3}|). \quad [3.16]$$

Similar bounds follow for the transformed  $y'$  and  $z'$  coordinates.

We'll use this result to add a method to the `Transform` class that also returns the absolute error in the transformed point due to applying the transformation.

```
(Transform Inline Functions) +≡
template <typename T> inline Point3<T>
Transform::operator()(const Point3<T> &p, Vector3<T> *pError) const {
    T x = p.x, y = p.y, z = p.z;
(Compute transformed coordinates from point pt)
(Compute absolute error for transformed point 229)
    if (wp == 1) return Point3<T>(xp, yp, zp);
    else         return Point3<T>(xp, yp, zp) / wp;
}
```

The fragment *(Compute transformed coordinates from point pt)* isn't included here; it implements the same matrix/point multiplication as in Section 2.8.

Note that the code that computes error bounds is buggy if the matrix is projective and the homogeneous  $w$  coordinate of the projected point is not one; this nit currently isn't a problem for pbrt's usage of this method.

---

<sup>18</sup> Although rounding error is introduced when transforming triangle vertices to world space (for example), this error doesn't add error that needs to be handled in computing intersection points. In other words, the transformed vertices may represent a perturbed representation of the scene, but they are the most precise representation available given the transformation.

```
(Compute absolute error for transformed point) ≡ 228
    T xAbsSum = (std::abs(m.m[0][0] * x) + std::abs(m.m[0][1] * y) +
                  std::abs(m.m[0][2] * z) + std::abs(m.m[0][3]));
    T yAbsSum = (std::abs(m.m[1][0] * x) + std::abs(m.m[1][1] * y) +
                  std::abs(m.m[1][2] * z) + std::abs(m.m[1][3]));
    T zAbsSum = (std::abs(m.m[2][0] * x) + std::abs(m.m[2][1] * y) +
                  std::abs(m.m[2][2] * z) + std::abs(m.m[2][3]));
    *pError = gamma(3) * Vector3<T>(xAbsSum, yAbsSum, zAbsSum);
```

The result in Equation (3.16) assumes that the point being transformed is exact. If the point itself has error bounded by  $\delta_x$ ,  $\delta_y$ , and  $\delta_z$ , then the transformed  $x$  coordinate is given by:

$$x' = (m_{0,0} \otimes (x \pm \delta_x) \oplus m_{0,1} \otimes (y \pm \delta_y)) \oplus (m_{0,2} \otimes (z \pm \delta_z) \oplus m_{0,3}).$$

Applying the definition of floating-point addition and multiplication's error bounds, we have:

$$\begin{aligned} x' = & m_{0,0}(x \pm \delta_x)(1 \pm \epsilon_m)^3 + m_{0,1}(y \pm \delta_y)(1 \pm \epsilon_m)^3 + \\ & m_{0,2}(z \pm \delta_z)(1 \pm \epsilon_m)^3 + m_{0,3}(1 \pm \epsilon_m)^2. \end{aligned}$$

Transforming to use  $\gamma$ , we can find the absolute error term to be bounded by

$$\begin{aligned} (\gamma_3 + 1)(|m_{0,0}|\delta_x + |m_{0,1}|\delta_y + |m_{0,2}|\delta_z) + \\ \gamma_3(|m_{0,0}x| + |m_{0,1}y| + |m_{0,2}z| + |m_{0,3}|). \end{aligned} \quad [3.17]$$

The `Transform` class also provides an `operator()` that takes a point and its own absolute error and returns the absolute error in the result, applying Equation (3.17). The definition is straightforward, so isn't included in the text here.

```
(Transform Public Methods) +≡
    template <typename T> inline Point3<T>
    operator()(const Point3<T> &p, const Vector3<T> &pError,
                Vector3<T> *pTransError) const;
```

83

The `Transform` class also provides methods to transform vectors and rays, returning the resulting error. The vector error bound derivations (and thence, implementations) are very similar to those for points, and so also aren't included here.

```
(Transform Public Methods) +≡
    template <typename T> inline Vector3<T>
    operator()(const Vector3<T> &v, Vector3<T> *vTransError) const;
    template <typename T> inline Vector3<T>
    operator()(const Vector3<T> &v, const Vector3<T> &vError,
                Vector3<T> *vTransError) const;
```

83

This method is used to transform the intersection point and its error bounds in the `Transform::operator()` method for `SurfaceInteractions`.

gamma() 217  
 Interaction::p 115  
 Interaction::pError 115  
 Point3 68  
 SurfaceInteraction 116  
 Transform::m 84  
 Vector3 59

```
(Transform p and pError in SurfaceInteraction) ≡
    ret.p = (*this)(si.p, si.pError, &ret.pError);
```

120

### 3.9.5 ROBUST SPAWNED RAY ORIGINS

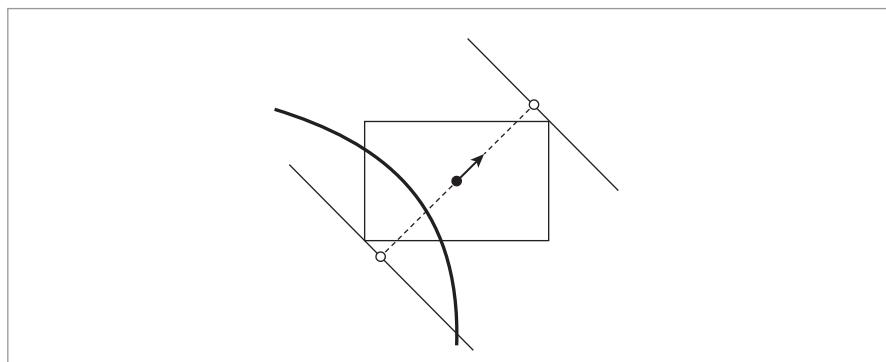
Computed intersection points and their error bounds give us a small 3D box that bounds a region of space. We know that the precise intersection point must be somewhere inside this box and that thus the surface must pass through the box (at least enough to present the point where the intersection is). (Recall Figure 3.43.) Having these boxes makes it possible to position the origins of rays leaving the surface so that they are always on the right side of the surface so that they don't incorrectly reintersect it. When tracing spawned rays leaving the intersection point  $p$ , we offset their origins enough to ensure that they are past the boundary of the error box and thus won't incorrectly re-intersect the surface.

In order to ensure that the spawned ray origin is definitely on the right side of the surface, we move far enough along the normal so that the plane perpendicular to the normal is outside the error bounding box. To see how to do this, consider a computed intersection point at the origin, where the plane equation for the plane going through the intersection point is just

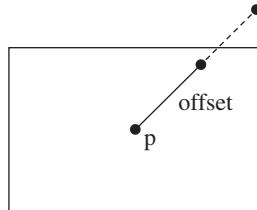
$$f(x, y, z) = \mathbf{n}_x x + \mathbf{n}_y y + \mathbf{n}_z z,$$

the plane is implicitly defined by  $f(x, y, z) = 0$ , and the normal is  $(\mathbf{n}_x, \mathbf{n}_y, \mathbf{n}_z)$ .

For a point not on the plane, the value of the plane equation  $f(x, y, z)$  gives the offset along the normal that gives a plane that goes through the point. We'd like to find the maximum value of  $f(x, y, z)$  for the eight corners of the error bounding box; if we offset the plane plus and minus this offset, we have two planes that don't intersect the error box that should be (locally) on opposite sides of the surface, at least at the computed intersection point offset along the normal (Figure 3.45).



**Figure 3.45:** Given a computed intersection point (filled circle) with surface normal (arrow) and error bounds (rectangle), we compute two planes offset along the normal that are offset just far enough so that they don't intersect the error bounds. The points on these planes along the normal from the computed intersection point give us the origins for spawned rays (open circles); one of the two is selected based on the ray direction so that the spawned ray won't pass through the error bounding box. By construction, such rays can't incorrectly re-intersect the actual surface (thick line).



**Figure 3.46:** The rounded value of the offset point  $p+offset$  computed in `OffsetRayOrigin()` may end up in the interior of the error box rather than on its boundary, which in turn introduces the risk of incorrect self-intersections if the rounded point is on the wrong side of the surface. Advancing each coordinate of the computed point one floating-point value away from  $p$  ensures that it is outside of the error box.

If the eight corners of the error bounding box are given by  $(\pm\delta_x, \pm\delta_y, \pm\delta_z)$ , then the maximum value of  $f(x, y, z)$  is easily computed:

$$d = |\mathbf{n}_x|\delta_x + |\mathbf{n}_y|\delta_y + |\mathbf{n}_z|\delta_z.$$

Computing spawned ray origins by offsetting along the surface normal in this way has a few advantages: assuming that the surface is locally planar (a reasonable assumption, especially at the very small scale of the intersection point error bounds), moving along the normal allows us to get from one side of the surface to the other while moving the shortest distance. In general, minimizing the distance that ray origins are offset is desirable for maintaining shadow and reflection detail.

*(Geometry Inline Functions) +≡*

```
inline Point3f OffsetRayOrigin(const Point3f &p, const Vector3f &pError,
                               const Normal3f &n, const Vector3f &w) {
    Float d = Dot(Abs(n), pError);
    Vector3f offset = d * Vector3f(n);
    if (Dot(w, n) < 0)
        offset = -offset;
    Point3f po = p + offset;
    <Round offset point po away from p 232>
    return po;
}
```

We also must handle round-off error when computing the offset point: when `offset` is added to `p`, the result will in general need to be rounded to the nearest floating-point value. In turn, it may be rounded down toward `p` such that the resulting point is in the interior of the error box rather than in its boundary (Figure 3.46). Therefore, the offset point is rounded away from `p` here to ensure that it's not inside the box.<sup>19</sup>

`Dot()` 63  
`Float` 1062  
`Normal3f` 71  
`OffsetRayOrigin()` 231  
`Point3f` 68  
`Vector3f` 60

19 The observant reader may now wonder about the effect of rounding error when computing the error bounds that are passed into this function. Indeed, these bounds should also be computed with rounding toward positive infinity. We ignore that issue under the expectation that the additional offset of one ulp here will be enough to cover that error.

Alternatively, the floating-point rounding mode could have been set to round toward plus or minus infinity (based on the sign of the value). Changing the rounding mode is generally fairly expensive, so we just shift the floating-point value by one ulp here. This will sometimes cause a value already outside of the error box to go slightly farther outside it, but because the floating-point spacing is so small, this isn't a problem in practice.

*(Round offset point po away from p) ≡*

231

```
for (int i = 0; i < 3; ++i) {
    if (offset[i] > 0)      po[i] = NextFloatUp(po[i]);
    else if (offset[i] < 0) po[i] = NextFloatDown(po[i]);
}
```

Given the `OffsetRayOrigin()` function, we can now implement the `Interaction` methods that generate rays leaving intersection points.

*(Interaction Public Methods) +≡*

115

```
Ray SpawnRay(const Vector3f &d) const {
    Point3f o = OffsetRayOrigin(p, pError, n, d);
    return Ray(o, d, Infinity, time, GetMedium(d));
}
```

The approach we've developed so far addresses the effect of floating-point error at the origins of rays leaving surfaces; there is a related issue for shadow rays to area light sources: we'd like to find any intersections with shapes that are very close to the light source and actually occlude it, while avoiding reporting incorrect intersections with the surface of the light source. Unfortunately, our implementation doesn't address this issue, so we set the `tMax` value of shadow rays to be just under one so that they stop before the surface of light sources.

*(Interaction Public Methods) +≡*

115

```
Ray SpawnRayTo(const Point3f &p2) const {
    Point3f origin = OffsetRayOrigin(p, pError, n, p2 - p);
    Vector3f d = p2 - origin;
    return Ray(origin, d, 1 - ShadowEpsilon, time, GetMedium(d));
}
```

Float 1062

Infinity 210

Interaction 115

Interaction::GetMedium() 688

Interaction:: 116

Interaction::pError 115

Interaction::time 115

OffsetRayOrigin() 231

Point3f 68

Ray 73

ShadowEpsilon 232

Vector3f 60

The other variant of `SpawnRayTo()`, which takes an `Interaction`, is analogous.

One last issue must be dealt with in order to maintain robust spawned ray origins: error introduced when performing transformations. Given a ray in one coordinate system where its origin was carefully computed to be on the appropriate side of some surface, transforming that ray to another coordinate system may introduce error in the transformed origin such that the origin is no longer on the correct side of the surface it was spawned from.

Therefore, whenever a ray is transformed by the `Ray` variant of `Transform::operator()` (which was implemented in Section 2.8.4), its origin is advanced to the edge of the bounds on the error that was introduced by the transformation. This ensures that the

origin conservatively remains on the correct side of the surface it was spawned from, if any.

```
(Offset ray origin to edge of error bounds and compute tMax) ≡ 95
    Float lengthSquared = d.LengthSquared();
    Float tMax = r.tMax;
    if (lengthSquared > 0) {
        Float dt = Dot(Abs(d), oError) / lengthSquared;
        o += d * dt;
        tMax -= dt;
    }
```

### 3.9.6 AVOIDING INTERSECTIONS BEHIND RAY ORIGINS

Bounding the error in computed intersection points allows us to compute ray origins that are guaranteed to be on the right side of the surface so that a ray with infinite precision wouldn't incorrectly intersect the surface it's leaving. However, a second source of rounding error must also be addressed: the error in parametric  $t$  values computed for ray–shape intersections. Rounding error can lead to an intersection algorithm computing a value  $t > 0$  for the intersection point even though the  $t$  value for the actual intersection is negative (and thus should be ignored).

It's possible to show that some intersection test algorithms always return a  $t$  value with the correct sign; this is the best case, as no further computation is needed to bound the actual error in the computed  $t$  value. For example, consider the ray–axis-aligned slab computation:  $t = (x \ominus o_x) \oslash d_x$ . IEEE guarantees that if  $a > b$ , then  $a \ominus b \geq 0$  (and if  $a < b$ , then  $a \ominus b \leq 0$ ). To see why this is so, note that if  $a > b$ , then the real number  $a - b$  must be greater than zero. When rounded to a floating-point number, the result must be either zero or a positive float; there's no way a negative floating-point number could be the closest floating-point number. Second, floating-point division returns the correct sign; these together guarantee that the sign of the computed  $t$  value is correct. (Or that  $t = 0$ , but this case is fine, since our test for an intersection is carefully chosen to be  $t > 0$ .)

For shape intersection routines that use EFloat, the computed  $t$  value in the end has an error bound associated with it, and no further computation is necessary to perform this test. See the definition of the fragment *(Check quadric shape t0 and t1 for nearest intersection)* in Section 3.2.2.

### Triangles

EFloat introduces computational overhead that we'd prefer to avoid for more commonly used shapes where efficient intersection code is more important. For these shapes, we can derive efficient-to-evaluate conservative bounds on the error in computed  $t$  values. The ray–triangle intersection algorithm in Section 3.6.2 computes a final  $t$  value by computing three edge function values  $e_i$  and using them to compute a barycentric-weighted sum of transformed vertex  $z$  coordinates,  $z_i$ :

Dot() 63  
Float 1062  
Vector3::Abs() 63  
Vector3::LengthSquared() 65

$$t = \frac{e_0 z_0 + e_1 z_1 + e_2 z_2}{e_0 + e_1 + e_2} \quad [3.18]$$

By successively bounding the error in these terms and then in the final  $t$  value, we can conservatively check that it is positive.

*(Ensure that computed triangle t is conservatively greater than zero) ≡* 158  
*(Compute  $\delta_z$  term for triangle t error bounds 234)*  
*(Compute  $\delta_x$  and  $\delta_y$  terms for triangle t error bounds 234)*  
*(Compute  $\delta_e$  term for triangle t error bounds 235)*  
*(Compute  $\delta_t$  term for triangle t error bounds and check t 235)*

Given a ray  $r$  with origin  $o$ , direction  $d$ , and a triangle vertex  $p$ , the projected  $z$  coordinate is

$$z = (1 \otimes d_z) \otimes (p_z \ominus o_z)$$

Applying the usual approach, we can find that the maximum error in  $z_i$  for each of three vertices of the triangle  $p_i$  is bounded by  $\gamma_3|z_i|$ , and we can thus find a conservative upper bound for the error in *any* of the  $z$  positions by taking the maximum of these errors:

$$\delta_z = \gamma_3 \max_i |z_i|.$$

*(Compute  $\delta_z$  term for triangle t error bounds) ≡* 234  

```
Float maxZt = MaxComponent(Abs(Vector3f(p0t.z, p1t.z, p2t.z)));
Float deltaZ = gamma(3) * maxZt;
```

The edge function values are computed as the difference of two products of transformed  $x$  and  $y$  vertex positions:

$$\begin{aligned} e_0 &= (x_1 \otimes y_2) \ominus (y_1 \otimes x_2) \\ e_1 &= (x_2 \otimes y_0) \ominus (y_2 \otimes x_0) \\ e_2 &= (x_0 \otimes y_1) \ominus (y_0 \otimes x_1). \end{aligned}$$

Bounds for the error in the transformed positions  $x_i$  and  $y_i$  are

$$\begin{aligned} \delta_x &= \gamma_5(\max_i |x_i| + \max_i |z_i|) \\ \delta_y &= \gamma_5(\max_i |y_i| + \max_i |z_i|). \end{aligned}$$

*(Compute  $\delta_x$  and  $\delta_y$  terms for triangle t error bounds) ≡* 234  

```
Float maxXt = MaxComponent(Abs(Vector3f(p0t.x, p1t.x, p2t.x)));
Float maxYt = MaxComponent(Abs(Vector3f(p0t.y, p1t.y, p2t.y)));
Float deltaX = gamma(5) * (maxXt + maxZt);
Float deltaY = gamma(5) * (maxYt + maxZt);
```

Taking the maximum error over all three of the vertices, the  $x_i \otimes y_j$  products in the edge functions are bounded by

$$(\max_i |x_i| + \delta_x)(\max_i |y_i| + \delta_y)(1 \pm \epsilon_m),$$

which have an absolute error bound of

$$\delta_{xy} = \gamma_2 \max_i |x_i| \max_i |y_i| + \delta_y \max_i |x_i| + \delta_x \max_i |y_i| + \dots$$

Dropping the (negligible) higher order terms of products of  $\gamma$  and  $\delta$  terms, the error bound on the difference of two  $x$  and  $y$  terms for the edge function is

$$\delta_e = 2(\gamma_2 \max_i |x_i| \max_i |y_i| + \delta_y \max_i |x_i| + \delta_x \max_i |y_i|).$$

Float 1062

Vector3f 60

*(Compute  $\delta_e$  term for triangle  $t$  error bounds) ≡*

```
Float deltaE = 2 * (gamma(2) * maxXt * maxYt + deltaY * maxXt +
                    deltaX * maxYt);
```

234

Again bounding error by taking the maximum of error over all of the  $e_i$  terms, the error bound for the computed value of the numerator of  $t$  in Equation (3.18) is

$$\delta_t = 3(\gamma_3 \max_i |e_i| \max_i |z_i| + \delta_e \max_i |z_i| + \delta_z \max_i |e_i|).$$

A computed  $t$  value (before normalization by the sum of  $e_i$ ) must be greater than this value for it to be accepted as a valid intersection that definitely has a positive  $t$  value.

*(Compute  $\delta_t$  term for triangle  $t$  error bounds and check  $t$ ) ≡*

```
Float maxE = MaxComponent(Abs(Vector3f(e0, e1, e2)));
Float deltaT = 3 * (gamma(3) * maxE * maxZt + deltaE * maxZt +
                    deltaZ * maxE) * std::abs(invDet);
if (t <= deltaT)
    return false;
```

234

Although it may seem that we have made a number of choices to compute looser bounds than we could, in the interests of efficiency, in practice the bounds on error in  $t$  are extremely small. For a regular scene that fills a bounding box roughly  $\pm 10$  in each dimension, our  $t$  error bounds near ray origins are generally around  $10^{-7}$ .

### 3.9.7 DISCUSSION

Minimizing and bounding numerical error in other geometric computations (e.g., partial derivatives of surface positions, interpolated texture coordinates, etc.) are much less important than they are for the positions of ray intersections. In a similar vein, the computations involving color and light in physically based rendering generally don't present trouble with respect to round-off error; they involve sums of products of positive numbers (usually with reasonably close magnitudes); hence catastrophic cancellation is not a commonly encountered issue. Furthermore, these sums are of few enough terms that accumulated error is small: the variance that is inherent in the Monte Carlo algorithms used for them dwarfs any floating-point error in computing them.

Interestingly enough, we saw an increase of roughly 20% in overall ray-tracing execution time after replacing the previous version of pbrt's old ad hoc method to avoid incorrect self-intersections with the method described in this section. (In comparison, rendering with double-precision floating point causes an increase in rendering time of roughly 30%.) Profiling showed that very little of the additional time was due to the additional computation to find error bounds; this is not surprising, as the incremental computation our method requires is limited—most of the error bounds are just scaled sums of absolute values of terms that have already been computed.

The majority of this slowdown is actually due to an increase in ray-object intersection tests. The reason for this increase in intersection tests was first identified by Wächter (2008, p. 30); when ray origins are very close to shape surfaces, more nodes of intersection acceleration hierarchies must be visited when tracing spawned rays than if overly loose offsets are used. Thus, more intersection tests are performed near the ray origin.

Float 1062

Vector3f 60

While this reduction in performance is unfortunate, it is actually a direct result of the greater accuracy of the method; it is the price to be paid for more accurate resolution of valid nearby intersections.

## FURTHER READING

*An Introduction to Ray Tracing* has an extensive survey of algorithms for ray–shape intersection (Glassner 1989a). Goldstein and Nagel (1971) discussed ray–quadric intersections, and Heckbert (1984) discussed the mathematics of quadrics for graphics applications in detail, with many citations to literature in mathematics and other fields. Hanrahan (1983) described a system that automates the process of deriving a ray intersection routine for surfaces defined by implicit polynomials; his system emits C source code to perform the intersection test and normal computation for a surface described by a given equation. Mitchell (1990) showed that interval arithmetic could be applied to develop algorithms for robustly computing intersections with implicit surfaces that cannot be described by polynomials and are thus more difficult to accurately compute intersections for; more recent work in this area was done by Knoll et al. (2009). See Moore’s book (1966) for an introduction to interval arithmetic.

Other notable early papers related to ray–shape intersection include Kajiya’s (1983) work on computing intersections with surfaces of revolution and procedurally generated fractal terrains. Fournier et al.’s (1982) paper on rendering procedural stochastic models and Hart et al.’s (1989) paper on finding intersections with fractals illustrate the broad range of shape representations that can be used with ray-tracing algorithms.

Kajiya (1982) developed the first algorithm for computing intersections with parametric patches. Subsequent work on more efficient techniques for direct ray intersection with patches includes papers by Stürzlinger (1998), Martin et al. (2000), and Roth et al. (2001). Benthin et al. (2004) presented more recent results and include additional references to previous work. Ramsey et al. (2004) describe an efficient algorithm for computing intersections with bilinear patches, and Ogaki and Tokuyoshi (2011) introduce a technique for directly intersecting smooth surfaces generated from triangle meshes with per-vertex normals.

An excellent introduction to differential geometry was written by Gray (1993); Section 14.3 of his book presents the Weingarten equations.

The ray–triangle intersection test in Section 3.6 was developed by Woop et al. (2013). See Möller and Trumbore (1997) for another widely used ray–triangle intersection algorithm. A ray–quadrilateral intersection routine was developed by Lagae and Dutré (2005). Shevtsov et al. (2007a) described a highly optimized ray–triangle intersection routine for modern CPU architectures and included a number of references to other recent approaches. An interesting approach for developing a fast ray–triangle intersection routine was introduced by Kensler and Shirley (2006): they implemented a program that performed a search across the space of mathematically equivalent ray–triangle tests, automatically generating software implementations of variations and then benchmark-

ing them. In the end, they found a more efficient ray–triangle routine than had been in use previously.

Phong and Crow (1975) first introduced the idea of interpolating per-vertex shading normals to give the appearance of smooth surfaces from polygonal meshes.

The layout of triangle meshes in memory can have a measurable impact on performance in many situations. In general, if triangles that are close together in 3D space are close together in memory, cache hit rates will be higher, and overall system performance will benefit. See Yoon et al. (2005) and Yoon and Lindstrom (2006) for algorithms for creating cache-friendly mesh layouts in memory.

The curve intersection algorithm in Section 3.7 is based on the approach developed by Nakamaru and Ohno (2002). Earlier methods for computing ray intersections with generalized cylinders are also applicable to rendering curves, though they are much less efficient (Bronsvoort and Klok 1985; de Voogt, van der Helm, and Bronsvoort 2000). The book by Farin (2001) provides an excellent general introduction to splines, and the blossoming approach used in Section 3.7 was introduced by Ramshaw (1987).

One challenge with rendering thin geometry like hair and fur is that thin geometry may require many pixel samples to be accurately resolved, which in turn increases rendering time. van Swaaij (2006) described a system that precomputed voxel grids to represent hair and fur, storing aggregate information about multiple hairs in a small region of space for more efficient rendering. More recently, Qin et al. (2014) described an approach based on cone tracing for rendering fur, where narrow cones are traced instead of rays. In turn, all of the curves that intersect a cone can be considered in computing the cone’s contribution, allowing high-quality rendering with a small number of cones per pixel.

Subdivision surfaces were invented by Doo and Sabin (1978) and Catmull and Clark (1978). The Loop subdivision method was originally developed by Charles Loop (1987), although the implementation in *pbrt* uses the improved rules for subdivision and tangents along boundary edges developed by Hoppe et al. (1994). There has been extensive subsequent work in subdivision surfaces. The SIGGRAPH course notes give a good summary of the state of the art in the year 2000 and also have extensive references (Zorin et al. 2000). See also Warren’s book on the topic (Warren 2002). Müller et al. (2003) described an approach that refines a subdivision surface on demand for the rays to be tested for intersection with it. (See also Benthin et al. (2007) for a related approach.)

An exciting development in subdivision surfaces is the ability to evaluate them at arbitrary points on the surface (Stam 1998). Subdivision surface implementations like the one in this chapter are often relatively inefficient, spending as much time dereferencing pointers as they do applying subdivision rules. Stam’s approach avoids these inefficiencies. Bolz and Schröder (2002) suggest an improved implementation approach that precomputes a number of quantities that make it possible to compute the final mesh much more efficiently. More recently, Patney et al. (2009) have demonstrated a very efficient approach for tessellating subdivision surfaces on data-parallel throughput processors.

Higham’s (2002) book on floating-point computation is excellent; it also develops the  $\gamma_n$  notation that we have used in Section 3.9. Other good references to this topic are Wilkinson (1994) and Goldberg (1991). While we have derived floating-point error bounds

manually, see the *Gappa* system by Daumas and Melquiod (2010) for a tool that automatically derives forward error bounds of floating-point computations.

The incorrect self-intersection problem has been a known problem for ray-tracing practitioners for quite some time (Haines 1989; Amanatides and Mitchell 1990). In addition to offsetting rays by an “epsilon” at their origin, approaches that have been suggested include ignoring intersections with the object that was previously intersected, “root polishing” (Haines 1989; Woo et al. 1996), where the computed intersection point is refined to become more numerically accurate; and using higher precision floating-point representations (e.g., double instead of float).

Kalra and Barr (1989) and Dammertz and Keller (2006) developed algorithms for numerically robust intersections based on recursively subdividing object bounding boxes, discarding boxes that don’t encompass the object’s surface, and discarding boxes missed by the ray. Both of these approaches are much less efficient than traditional ray–object intersection algorithms as well as the techniques introduced in Section 3.9.

Salesin et al. (1989) introduced techniques to derive robust primitive operations for computational geometry that accounted for floating-point round-off error, and Ize showed how to perform numerically robust ray-bounding box intersections (Ize 2013); his approach is implemented in Section 3.9.2. (With a more careful derivation, he shows that a scale factor of  $2\gamma_2$  can actually be used to increase  $t_{\text{Max}}$ , rather than the  $2\gamma_3$  we have derived here.) Wächter (2008) discussed self-intersection issues in his thesis; he suggested recomputing the intersection point starting from the initial intersection (root polishing) and offsetting spawned rays along the normal by a fixed small fraction of the intersection point’s magnitude. The approach implemented in this chapter uses his approach of offsetting ray origins along the normal but is based on conservative bounds on the offsets based on the numerical error present in computed intersection points. (As it turns out, our bounds are generally tighter than Wächter’s offsets while also being provably conservative.)

## EXERCISES

- ➲ 3.1 One nice property of mesh-based shapes like triangle meshes and subdivision surfaces is that the shape’s vertices can be transformed into world space, so that it isn’t necessary to transform rays into object space before performing ray intersection tests. Interestingly enough, it is possible to do the same thing for ray–quadric intersections.

The implicit forms of the quadrics in this chapter were all of the form

$$Ax^2 + Bxy + Cxz + Dy^2 + Eyz + Fz^2 + G = 0,$$

where some of the constants  $A \dots G$  were zero. More generally, we can define quadric surfaces by the equation

$$Ax^2 + By^2 + Cz^2 + 2Dxy + 2Eyz + 2Fxz + 2Gz + 2Hy + 2Iz + J = 0,$$

where most of the parameters  $A \dots J$  don’t directly correspond to the earlier  $A \dots G$ . In this form, the quadric can be represented by a  $4 \times 4$  symmetric matrix  $\mathbf{Q}$ :

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{pmatrix} A & D & F & G \\ D & B & E & H \\ F & E & C & I \\ G & H & I & J \end{pmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \mathbf{p}^T \mathbf{Q} \mathbf{p} = 0.$$

Given this representation, first show that the matrix  $\mathbf{Q}'$  representing a quadric transformed by the matrix  $\mathbf{M}$  is

$$\mathbf{Q}' = (\mathbf{M}^T)^{-1} \mathbf{Q} \mathbf{M}^{-1}.$$

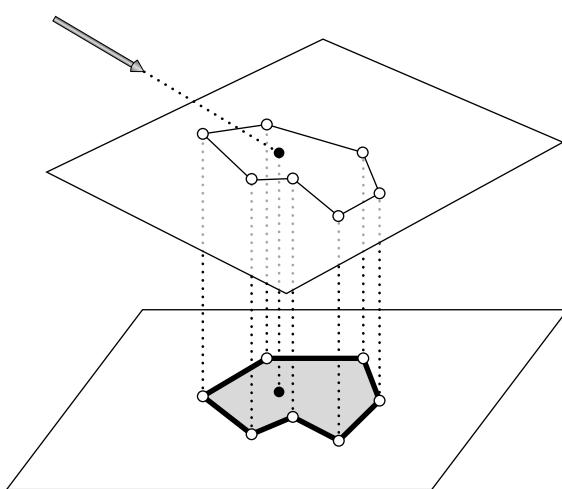
To do so, show that for any point  $\mathbf{p}$  where  $\mathbf{p}^T \mathbf{Q} \mathbf{p} = 0$ , if we apply a transformation  $\mathbf{M}$  to  $\mathbf{p}$  and compute  $\mathbf{p}' = \mathbf{M}\mathbf{p}$ , we'd like to find  $\mathbf{Q}'$  so that  $(\mathbf{p}')^T \mathbf{Q}' \mathbf{p}' = 0$ .

Next, substitute the ray equation into the earlier more general quadric equation to compute coefficients for the quadratic equation  $a t^2 + b t + c = 0$  in terms of entries of the matrix  $\mathbf{Q}$  to pass to the `Quadratic()` function.

Now implement this approach in `pbrt` and use it instead of the original quadric intersection routines. Note that you will still need to transform the resulting world space hit points into object space to test against  $\theta_{\max}$ , if it is not  $2\pi$ , and so on. How does performance compare to the original scheme?

- ① 3.2** Improve the object space bounding box routines for the quadrics to properly account for  $\phi_{\max} < 3\pi/2$ , and compute tighter bounding boxes when possible. How much does this improve performance when rendering scenes with partial quadric shapes?
- ② 3.3** There is room to optimize the implementations of the various quadric primitives in `pbrt` in a number of ways. For example, for complete spheres some of the tests in the intersection routine related to partial spheres are unnecessary. Furthermore, some of the quadrics have calls to trigonometric functions that could be turned into simpler expressions using insight about the geometry of the particular primitives. Investigate ways to speed up these methods. How much does doing so improve the overall run time of `pbrt` when rendering scenes with quadrics?
- ① 3.4** Currently `pbrt` recomputes the partial derivatives  $\partial \mathbf{p} / \partial u$  and  $\partial \mathbf{p} / \partial v$  for triangles every time they are needed, even though they are constant for each triangle. Precompute these vectors and analyze the speed/storage trade-off, especially for large triangle meshes. How do the depth complexity of the scene and the size of triangles in the image affect this trade-off?
- ② 3.5** Implement a general polygon primitive that supports an arbitrary number of vertices and convex or concave polygons as a new `Shape` in `pbrt`. You can assume that a valid polygon has been provided and that all of the vertices of the polygon lie on the same plane, although you might want to issue a warning when this is not the case.

An efficient technique for computing ray–polygon intersections is to find the plane equation for the polygon from its normal and a point on the plane. Then compute the intersection of the ray with that plane and project the intersection point and the polygon vertices to 2D. Then apply a 2D point-in-polygon test



**Figure 3.47:** A ray–polygon intersection test can be performed by finding the point where the ray intersects the polygon’s plane, projecting the hit point and polygon vertices onto an axis-aligned plane, and doing a 2D point-in-polygon test there.

to determine if the point is inside the polygon. An easy way to do this is to effectively do a 2D ray-tracing computation, intersect the ray with each of the edge segments, and count how many it goes through. If it goes through an odd number of them, the point is inside the polygon and there is an intersection. See Figure 3.47 for an illustration of this idea.

You may find it helpful to read the article by Haines (1994) that surveys a number of approaches for efficient point-in-polygon tests. Some of the techniques described there may be helpful for optimizing this test. Furthermore, Section 13.3.3 of Schneider and Eberly (2003) discusses strategies for getting all the corner cases right: for example, when the 2D ray is aligned precisely with an edge or passes through a vertex of the polygon.

- ② 3.6 Constructive solid geometry (CSG) is a classic solid modeling technique, where complex shapes are built up by considering the union, intersection, and differences of more primitive shapes. For example, a sphere could be used to create pits in a cylinder if a shape was modeled as the difference of a cylinder and set of spheres that partially overlapped it. See Hoffmann (1989) for further information about CSG.

Add support for CSG to pbrt and render images that demonstrate interesting shapes that can be rendered using CSG. You may want to read Roth (1982), which first described how ray tracing could be used to render models described by CSG, as well as Amanatides and Mitchell (1990), which discusses precision-related issues for CSG ray tracing.

- ② 3.7** Procedurally described parametric surfaces: Write a Shape that takes a general mathematical expression of the form  $f(u, v) \rightarrow (x, y, z)$  that describes a parametric surface as a function of  $(u, v)$ . Evaluate the given function at a grid of  $(u, v)$  positions, and create a triangle mesh that approximates the given surface.
- ② 3.8** Adaptive curve refinement: adjust the number of levels of recursive refinement used for intersection with Curve shapes based on the on-screen area that they cover. One approach is to take advantage of the RayDifferential class, which represents the image space area that a given ray represents. (However, currently, only Rays—not RayDifferentials—are passed to the Shape::Intersect() method, so you'd need to modify other parts of the system to make ray differentials available.) Alternatively, you could modify the Camera to provide information about the projected length of vectors between points in world space on the image plane and make the camera available during Curve creation.
- ③ 3.9** Almost all methods for subdivision surfaces are based on either refining a mesh of triangles or a mesh of quadrilaterals. If a rendering system only supports one type of mesh, meshes of the other type are typically tessellated to make faces of the expected type in a preprocessing step. However, doing this can introduce artifacts in the final subdivision surface. Read Stam and Loop's paper on a hybrid subdivision scheme that supports meshes with both quadrilateral and triangular faces (Stam and Loop 2003), and implement their method. Demonstrate cases where the subdivision surface that your implementation creates does not have artifacts that are present in the output from pbrt's current subdivision implementation.
- ② 3.10** The smoothness of subdivision surfaces isn't always desirable. Sometimes it is useful to be able to flag some edges of a subdivision control mesh as "creases" and apply different subdivision rules there to preserve a sharp edge. Extend the subdivision surface implementation in this chapter so that some edges can be denoted as creases, and use the boundary subdivision rules to compute the positions of vertices along those edges. Render images showing the difference this makes.
- ③ 3.11** Adaptive subdivision: a weakness of the subdivision surface implementation in Section 3.8 is that each face is always refined a fixed number of times: this may mean that some faces are underrefined, leading to visible facetting in the triangle mesh, and some faces are overrefined, leading to excessive memory use and rendering time. With adaptive subdivision, individual faces are no longer subdivided once a particular error threshold has been reached.

Curve 168

Ray 73

RayDifferential 75

Shape 123

Shape::Intersect() 129

An easy error threshold to implement computes the face normals of each face and its directly adjacent faces. If they are sufficiently close to each other (e.g., as tested via dot products), then the limit surface for that face will be reasonably flat and further refinement will likely make little difference to the final surface. Alternatively, you might want to approximate the area that a subdivided face covers on the image plane and continue subdividing until this area becomes sufficiently small. This approximation could be done using ray differentials;

see Section 10.1.1 for an explanation of how to relate the ray differential to the screen space footprint.

The trickiest part of this exercise is that some faces that don't need subdivision due to the flatness test will still need to be subdivided in order to provide vertices so that neighboring faces that do need to subdivide can get their vertex one-rings. In particular, adjacent faces can differ by no more than one level of subdivision. You may find it useful to read recent papers by Patney et al. (2009) and Fisher et al. (2009) for discussion of how to avoid cracks in adaptively subdivided meshes.

- ③ 3.12 Ray-tracing point-sampled geometry: extending methods for rendering complex models represented as a collection of point samples (Levoy and Whitted 1985; Pfister et al. 2000; Rusinkiewicz and Levoy 2000), Schaufler and Jensen (2000) have described a method for intersecting rays with collections of oriented point samples in space. They probabilistically determined that an intersection has occurred when a ray approaches a sufficient local density of point samples and computes a surface normal with a weighted average of the nearby samples. Read their paper and extend `pbrt` to support a point-sampled geometry shape. Do any of `pbrt`'s basic interfaces need to be extended or generalized to support a shape like this?
- ③ 3.13 Deformation motion blur: the `TransformedPrimitive` in Section 4.1.2 of Chapter 4 supports animated shapes via transformations of primitives that vary over time. However, this type of animation isn't general enough to represent a triangle mesh where each vertex has a position given at the start time and another one at the end time. (For example, this type of animation description can be used to describe a running character model where different parts of the body are moving in different ways.) Implement a more general `Triangle` shape that supports specifying vertex positions at the start and end of frame and interpolates between them based on the ray time passed to the intersection methods. Be sure to update the bounding routines appropriately.

Triangle meshes with very large amounts of motion may exhibit poor performance due to triangles sweeping out very large bounding boxes and thus many ray–triangle intersections being performed that don't hit the triangle. Can you come up with approaches that could be used to reduce the impact of this problem?

- ③ 3.14 Implicit functions: just as implicit definitions of the quadric shapes are a useful starting point for deriving ray-intersection algorithms, more complex implicit functions can also be used to define interesting shapes. In particular, difficult-to-model organic shapes, water drops, and so on can be well represented by implicit surfaces. Blinn (1982a) introduced the idea of directly rendering implicit surfaces, and Wyvill and Wyvill (1989) gave a basis function for implicit surfaces with a number of advantages compared to Blinn's.

Implement a method for finding ray intersections with general implicit surfaces and add it to `pbrt`. You may wish to read papers by Kalra and Barr (1989) and Hart (1996) for methods for ray tracing them. Mitchell's algorithm for

robust ray intersections with implicit surfaces using interval arithmetic gives another effective method for finding these intersections (Mitchell 1990), and more recently Knoll et al. (2009) described refinements to this idea. You may find an approach along these lines easier to implement than the others. See Moore’s book on interval arithmetic as needed for reference (Moore 1966).

- ③ 3.15 L-systems: a very successful technique for procedurally modeling plants was introduced to graphics by Alvy Ray Smith (1984), who applied *Lindenmayer systems* (L-systems) to model branching plant structures. Prusinkiewicz and collaborators have generalized this approach to encompass a much wider variety of types of plants and effects that determine their appearance (Prusinkiewicz 1986; Prusinkiewicz, James, and Mech 1994; Deussen et al. 1998; Prusinkiewicz et al. 2001). L-systems describe the branching structure of these types of shapes via a grammar. The grammar can be evaluated to form expressions that describe a topological representation of a plant, which can then be translated into a geometric representation. Add an L-system primitive to `pbrt` that takes a grammar as input and evaluates it to create the shape it describes.
- ① 3.16 Given an arbitrary point  $(x, y, z)$ , what bound on the error from applying a scale transformation of  $(2, 1, 4)$  is given by Equation (3.16)? How much error is actually introduced?
- ② 3.17 The quadric shapes all use the `EFloat` class for their intersection tests in order to be able to bound the error in the computed  $t$  value so that intersections behind the ray origin aren’t incorrectly reported as actual intersections. First, measure the performance difference when using regular `Floats` for one or more quadrics when rendering a scene that includes those shapes. Next, manually derive conservative error bounds for  $t$  values computed by those shapes as was done for triangles in Section 3.9.6. Implement your method. You may find it useful to use the `EFloat` class to empirically test your derivation’s correctness. Measure the performance difference with your implementation.
- ② 3.18 One detail thwarts the watertightness of the current `Triangle` shape implementation: the translation and shearing of triangle vertices introduces round-off error, which must be accounted for in the extent of triangles’ bounding boxes; see Section 3.3 of Woop et al. (2013) for discussion (and a solution). Modify `pbrt` to incorporate a solution to this shortcoming. Can you find scenes where small image errors are eliminated thanks to your fix?
- ③ 3.19 Rendering in camera space: because floating-point arithmetic provides more precision near the origin than farther away from it, transforming the scene to a coordinate system with the camera at the origin can reduce the impact of error due to insufficient floating-point precision. (For example, consider the difference between rendering a scene in world space with the camera at  $(100000, 100000, 100000)$ , looking at a unit sphere two units away versus a scene with the camera at the origin, also looking at a sphere two units away: many more bits of precision are available to represent intersection points in the latter case.)

Modify pbrt to primarily perform rendering computations in camera space, rather than world space, as it does currently. You'll need to modify the Camera implementations to return camera-space rays and modify shapes to transform incoming rays from camera space to object space. You'll want to transform the vertices of TriangleMeshes all the way to camera space. Measure the performance of your implementation versus an unmodified version of pbrt and render a variety of scenes with both systems. (In particular, make sure you test some scenes that have the camera far from the world space origin.) What sort of image differences do you see?

This page intentionally left blank

# CHAPTER FOUR



## 04 PRIMITIVES AND INTERSECTION ACCELERATION

The classes described in the last chapter focus exclusively on representing geometric properties of 3D objects. Although the `Shape` class provides a convenient abstraction for geometric operations such as intersection and bounding, it doesn't contain enough information to fully describe an object in a scene. For example, it is necessary to bind material properties to each shape in order to specify its appearance. To accomplish these goals, this chapter introduces the `Primitive` abstract base class and provides a number of implementations.

Shapes to be rendered directly are represented by the `GeometricPrimitive` class. This class combines a `Shape` with a description of its appearance properties. So that the geometric and shading portions of `pbrt` can be cleanly separated, these appearance properties are encapsulated in the `Material` class, which is described in Chapter 9.

The `TransformedPrimitive` class handles two more general uses of `Shapes` in the scene: shapes with animated transformation matrices and object instancing, which can greatly reduce the memory requirements for scenes that contain many instances of the same geometry at different locations (such as the one in Figure 4.1). Implementing each of these features essentially requires injecting an additional transformation matrix between the `Shape`'s notion of world space and the actual scene world space. Therefore, both are handled by a single class.

This chapter also introduces the `Aggregate` class, which represents a container that can hold many `Primitives`. `pbrt` uses this class as a base for *acceleration structures*—data structures that help reduce the otherwise  $O(n)$  complexity of testing a ray for intersection with all  $n$  objects in a scene. Most rays will intersect only a few primitives and miss



**Figure 4.1:** This outdoor scene makes heavy use of instancing as a mechanism for compressing the scene’s description. There are only 24 million unique triangles in the scene, although, thanks to object reuse through instancing, the total geometric complexity is 3.1 billion triangles. (*Scene courtesy of Laubwerk.*)

the others by a large distance. If an intersection acceleration algorithm can reject whole groups of primitives at once, there will be a substantial performance improvement compared to simply testing each ray against each primitive in turn. One benefit from reusing the `Primitive` interface for these acceleration structures is that it makes it easy to support hybrid approaches where an accelerator of one type holds accelerators of other types.

This chapter describes the implementation of two accelerators, one, `BVHAccel`, based on building a hierarchy of bounding boxes around objects in the scene, and the second, `KdTreeAccel`, based on adaptive recursive spatial subdivision. While many other acceleration structures have been proposed, almost all ray tracers today use one of these two. The “Further Reading” section at the end of this chapter has extensive references to other possibilities.

## 4.1 PRIMITIVE INTERFACE AND GEOMETRIC PRIMITIVES

The abstract `Primitive` base class is the bridge between the geometry processing and shading subsystems of `pbrt`.

```
(Primitive Declarations) ≡
class Primitive {
public:
    (Primitive Interface 249)
};
```

`BVHAccel` 256  
`KdTreeAccel` 285  
`Primitive` 248

There are a number of geometric routines in the Primitive interface, all of which are similar to a corresponding Shape method. The first, `Primitive::WorldBound()`, returns a box that encloses the primitive's geometry in world space. There are many uses for such a bound; one of the most important is to place the Primitive in the acceleration data structures.

```
(Primitive Interface) ≡ 248
    virtual Bounds3f WorldBound() const = 0;
```

The next two methods provide ray intersection tests. One difference between the two base classes is that `Shape::Intersect()` returns the parametric distance along the ray to the intersection in a `Float *` output variable, while `Primitive::Intersect()` is responsible for updating `Ray::tMax` with this value if an intersection is found.

```
(Primitive Interface) +≡ 248
    virtual bool Intersect(const Ray &r, SurfaceInteraction *) const = 0;
    virtual bool IntersectP(const Ray &r) const = 0;
```

Upon finding an intersection, the Primitive's `Intersect()` method is also responsible for initializing additional `SurfaceInteraction` member variables, including a pointer to the Primitive that the ray hit.

```
(SurfaceInteraction Public Data) +≡ 116
    const Primitive *primitive = nullptr;
```

Primitive objects have a few methods related to non-geometric properties as well. The first, `Primitive::GetAreaLight()`, returns a pointer to the `AreaLight` that describes the primitive's emission distribution, if the primitive is itself a light source. If the primitive is not emissive, this method should return `nullptr`.

```
(Primitive Interface) +≡ 248
    virtual const AreaLight *GetAreaLight() const = 0;
```

`GetMaterial()` returns a pointer to the material instance assigned to the primitive. If `nullptr` is returned, ray intersections with the primitive should be ignored; the primitive only serves to delineate a volume of space for participating media. This method is also used to check if two rays have intersected the same object by comparing their `Material` pointers.

```
(Primitive Interface) +≡ 248
    virtual const Material *GetMaterial() const = 0;
```

The third material-related method, `ComputeScatteringFunctions()`, initializes representations of the light-scattering properties of the material at the intersection point on the surface. The `BSDF` object (introduced in Section 9.1) describes local light-scattering properties at the intersection point. If applicable, this method also initializes a `BSSRDF`, which describes subsurface scattering inside the primitive—light that enters the surface at points far from where it exits. While subsurface light transport has little effect on the appearance of objects like metal, cloth, or plastic, it is the dominant light-scattering mechanism for biological materials like skin, thick liquids like milk, etc. The `BSSRDF` is supported by an extension of the path tracing algorithm discussed in Section 15.

In addition to a `MemoryArena` to allocate memory for the `BSDF` and/or `BSSRDF`, this method takes a `TransportMode` enumerant that indicates whether the ray path that found this intersection point started from the camera or from a light source; as will be discussed further in Section 16.1, this detail has important implications for how some parts of material models are evaluated. The `allowMultipleLobes` parameter controls a detail of how some types of `BRDFs` are represented; it is discussed further in Section 9.2. Section 9.1.1 discusses the use of the `MemoryArena` for `BSDF` memory allocation in more detail.

*(Primitive Interface)* +≡ 248

```
virtual void ComputeScatteringFunctions(SurfaceInteraction *isect,
    MemoryArena &arena, TransportMode mode,
    bool allowMultipleLobes) const = 0;
```

The `BSDF` and `BSSRDF` pointers for the point are stored in the `SurfaceInteraction` passed to `ComputeScatteringFunctions()`.

*(SurfaceInteraction Public Data)* +≡ 116

```
BSDF *bsdf = nullptr;
BSSRDF *bssrdf = nullptr;
```

#### 4.1.1 GEOMETRIC PRIMITIVES

The `GeometricPrimitive` class represents a single shape (e.g., a sphere) in the scene. One `GeometricPrimitive` is allocated for each shape in the scene description provided by the user. This class is implemented in the files `core/primitive.h` and `core/primitive.cpp`.

*(GeometricPrimitive Declarations)* ≡

```
class GeometricPrimitive : public Primitive {
public:
    (GeometricPrimitive Public Methods)
private:
    (GeometricPrimitive Private Data 250)
};
```

Each `GeometricPrimitive` holds a reference to a `Shape` and its `Material`. In addition, because primitives in `pbrt` may be area light sources, it stores a pointer to an `AreaLight` object that describes its emission characteristics (this pointer is set to `nullptr` if the primitive does not emit light). Finally, the `MediumInterface` attribute encodes information about the participating media on the inside and outside of the primitive.

*(GeometricPrimitive Private Data)* ≡ 250

```
std::shared_ptr<Shape> shape;
std::shared_ptr<Material> material;
std::shared_ptr<AreaLight> areaLight;
MediumInterface mediumInterface;
```

The `GeometricPrimitive` constructor just initializes these variables from the parameters passed to it. It's straightforward, so we don't include it here.

Most of the methods of the `Primitive` interface related to geometric processing are simply forwarded to the corresponding `Shape` method. For example, `GeometricPrimitive::`:

`AreaLight` 734  
`BSDF` 572  
`BSSRDF` 692  
`GeometricPrimitive` 250  
`Material` 577  
`MediumInterface` 684  
`MemoryArena` 1074  
`Primitive` 248  
`Shape` 123  
`SurfaceInteraction` 116  
`TransportMode` 960

`Intersect()` calls the `Shape::Intersect()` method of its enclosed `Shape` to do the actual intersection test and initialize a `SurfaceInteraction` to describe the intersection, if any. It also uses the returned parametric hit distance to update the `Ray::tMax` member. The advantage of storing the distance to the closest hit in `Ray::tMax` is that this makes it easy to avoid performing intersection tests with any primitives that lie farther along the ray than any already-found intersections.

```
(GeometricPrimitive Method Definitions) ≡
    bool GeometricPrimitive::Intersect(const Ray &r,
        SurfaceInteraction *isect) const {
        Float tHit;
        if (!shape->Intersect(r, &tHit, isect))
            return false;
        r.tMax = tHit;
        isect->primitive = this;
        <Initialize SurfaceInteraction::mediumInterface after Shape intersection 685>
        return true;
    }
```

We won't include the implementations of the `GeometricPrimitive`'s `WorldBound()` or `IntersectP()` methods here; they just forward these requests on to the `Shape` in a similar manner. Similarly, `GetAreaLight()` just returns the `GeometricPrimitive::areaLight` member.

```
AnimatedTransform 103
Float 1062
GeometricPrimitive 250
GeometricPrimitive::areaLight
    250
GeometricPrimitive::
    Intersect()
    251
GeometricPrimitive::material
    250
GeometricPrimitive::shape
    250
Material 577
Material::
    ComputeScatteringFunctions()
    577
MemoryArena 1074
Primitive 248
Ray 73
Ray::tMax 73
Shape 123
Shape::Intersect() 129
SurfaceInteraction 116
SurfaceInteraction::primitive
    249
TransformedPrimitive 252
TransportMode 960
```

Finally, the `ComputeScatteringFunctions()` method just forwards the request on to the `Material`.

```
(GeometricPrimitive Method Definitions) +≡
void GeometricPrimitive::ComputeScatteringFunctions(
    SurfaceInteraction *isect, MemoryArena &arena, TransportMode mode,
    bool allowMultipleLobes) const {
    if (material)
        material->ComputeScatteringFunctions(isect, arena, mode,
            allowMultipleLobes);
}
```

#### 4.1.2 TransformedPrimitive: OBJECT INSTANCING AND ANIMATED PRIMITIVES

`TransformedPrimitive` holds a single `Primitive` and also includes an `AnimatedTransform` that is injected in between the underlying primitive and its representation in the scene. This extra transformation enables two useful features: object instancing and primitives with animated transformations.

Object instancing is a classic technique in rendering that reuses transformed copies of a single collection of geometry at multiple positions in a scene. For example, in a model of a concert hall with thousands of identical seats, the scene description can be compressed substantially if all of the seats refer to a shared geometric representation of a single seat. The ecosystem scene in Figure 4.1 has 23,241 individual plants of various types, although only 31 unique plant models. Because each plant model is instanced multiple times with a different transformation for each instance, the complete scene has a total of 3.1 billion

triangles, although only 24 million triangles are stored in memory, thanks to primitive reuse through object instancing. pbrt uses just over 7 GB of memory when rendering this scene with object instancing (1.7 GB for BVHs, 2.3 GB for triangle meshes, and 3 GB for texture maps), but would need upward of 516 GB to render it without instancing.

Animated transformations enable rigid-body animation of primitives in the scene via the `AnimatedTransform` class. See Figure 2.15 for an image that exhibits motion blur due to animated transformations.

Recall that the `Shapes` of Chapter 3 themselves had object-to-world transformations applied to them to place them in the scene. If a shape is held by a `TransformedPrimitive`, then the shape's notion of world space isn't the actual scene world space—only after the `TransformedPrimitive`'s transformation is also applied is the shape actually in world space. For the applications here, it makes sense for the shape to not be at all aware of the additional transformations being applied. For animated shapes, it's simpler to isolate all of the handling of animated transformations to a single class here, rather than require all `Shapes` to support `AnimatedTransforms`. Similarly, for instanced primitives, letting `Shapes` know all of the instance transforms is of limited utility: we wouldn't want the `TriangleMesh` to make a copy of its vertex positions for each instance transformation and transform them all the way to world space, since this would negate the memory savings of object instancing.

*(TransformedPrimitive Declarations) ≡*

```
class TransformedPrimitive : public Primitive {
public:
    (TransformedPrimitive Public Methods 252)
private:
    (TransformedPrimitive Private Data 252)
};
```

The `TransformedPrimitive` constructor takes a reference to the `Primitive` that represents the model, and the transformation that places it in the scene. If the geometry is described by multiple `Primitives`, the calling code is responsible for placing them in an `Aggregate` implementation so that only a single `Primitive` needs to be stored here. For the code that creates aggregates as needed, see the `pbrtObjectInstance()` function in Section B.3.6 of Appendix B for the case of primitive instances, and see the `pbrtShape()` function in Section B.3.5 for animated shapes.

*(TransformedPrimitive Public Methods) ≡*

252

```
TransformedPrimitive(std::shared_ptr<Primitive> &primitive,
                    const AnimatedTransform &PrimitiveToWorld)
: primitive(primitive), PrimitiveToWorld(PrimitiveToWorld) { }
```

*(TransformedPrimitive Private Data) ≡*

252

```
std::shared_ptr<Primitive> primitive;
const AnimatedTransform PrimitiveToWorld;
```

The key task of the `TransformedPrimitive` is to bridge the `Primitive` interface that it implements and the `Primitive` that it holds a pointer to, accounting for the effects of the additional transformation that it holds. The `TransformedPrimitive`'s `PrimitiveToWorld`

Aggregate 255  
`AnimatedTransform` 103  
`pbrtObjectInstance()` 1128  
`pbrtShape()` 1124  
`Primitive` 248  
`Shape` 123  
`TransformedPrimitive` 252  
`TriangleMesh` 154

transformation defines the transformation from the coordinate system of this particular instance of the geometry to world space. If the primitive member has its own transformation, that should be interpreted as the transformation from object space to the TransformedPrimitive's coordinate system. The complete transformation to world space requires both of these transformations together.

As such, the TransformedPrimitive::Intersect() method transforms the given ray to the primitive's coordinate system and passes the transformed ray to its Intersect() routine. If a hit is found, the tMax value from the transformed ray needs to be copied into the ray r originally passed to the Intersect() routine.

*(TransformedPrimitive Method Definitions) ≡*

```
bool TransformedPrimitive::Intersect(const Ray &r,
                                     SurfaceInteraction *isect) const {
    (Compute ray after transformation by PrimitiveToWorld 253)
    if (!primitive->Intersect(ray, isect))
        return false;
    r.tMax = ray.tMax;
    (Transform instance's intersection data to world space 253)
    return true;
}
```

AnimatedTransform::  
    Interpolate() 106  
Inverse() 1081  
Primitive 248  
Primitive::Intersect() 249  
Primitive::WorldBound() 249  
Ray 73  
Ray::time 73  
Ray::tMax 73  
SurfaceInteraction 116  
Transform 83  
Transform::IsIdentity() 85  
Transform::operator() 93  
TransformedPrimitive 252  
TransformedPrimitive::  
    Intersect() 253  
TransformedPrimitive::  
    primitive 252  
TransformedPrimitive::  
    PrimitiveToWorld 252

To transform the ray, we need to interpolate the transformation based on the ray's time. Although we want to transform the ray r from world space to primitive space, here we actually interpolate PrimitiveToWorld and then invert the resulting Transform to get the transformation. This surprising approach is necessary because of how the polar decomposition-based transformation interpolation algorithm in Section 2.9.3 works: interpolating PrimitiveToWorld to some time and inverting it doesn't necessarily give the same result as interpolating its inverse, the animated world to primitive transformation, directly. Because Primitive::WorldBound() uses PrimitiveToWorld to compute the primitive's bounding box, we must also interpolate PrimitiveToWorld here for consistency.

*(Compute ray after transformation by PrimitiveToWorld) ≡*

```
Transform InterpolatedPrimToWorld;  
PrimitiveToWorld.Interpolate(r.time, &InterpolatedPrimToWorld);  
Ray ray = Inverse(InterpolatedPrimToWorld)(r);
```

253

Finally, the SurfaceInteraction at the intersection point needs to be transformed to world space; the primitive's intersection member will already have transformed the SurfaceInteraction to its notion of world space, so here we only need to apply the effect of the additional transformation held here.

*(Transform instance's intersection data to world space) ≡*

```
if (!InterpolatedPrimToWorld.IsIdentity())
    *isect = InterpolatedPrimToWorld(*isect);
```

253

The rest of the geometric Primitive methods are forwarded on to the shared instance, with the results similarly transformed as needed by the TransformedPrimitive's transformation.

*(TransformedPrimitive Public Methods)* +≡

```
Bounds3f WorldBound() const {
    return PrimitiveToWorld.MotionBounds(primitive->WorldBound());
}
```

252

The `TransformedPrimitive` `GetAreaLight()`, `GetMaterial()`, and `ComputeScatteringFunctions()` methods should never be called. The corresponding methods of the primitive that the ray actually hit should always be called instead. Therefore, any attempt to call the `TransformedPrimitive` implementations of these methods (not shown here) results in a run-time error.

## 4.2 AGGREGATES

Acceleration structures are one of the components at the heart of any ray tracer. Without algorithms to reduce the number of unnecessary ray intersection tests, tracing a single ray through a scene would take time linear in the number of primitives in the scene, since the ray would need to be tested against each primitive in turn to find the closest intersection. However, doing so is extremely wasteful in most scenes, since the ray passes nowhere near the vast majority of primitives. The goals of acceleration structures are to allow the quick, simultaneous rejection of *groups* of primitives and to order the search process so that nearby intersections are likely to be found first so that farther away ones can potentially be ignored.

Because ray-object intersections can account for the bulk of execution time in ray tracers, there has been a substantial amount of research into algorithms for ray intersection acceleration. We will not try to explore all of this work here but refer the interested reader to references in the “Further Reading” section at the end of this chapter and in particular Arvo and Kirk’s chapter in *An Introduction to Ray Tracing* (Glassner 1989a), which has a useful taxonomy for classifying different approaches to ray-tracing acceleration.

Broadly speaking, there are two main approaches to this problem: spatial subdivision and object subdivision. Spatial subdivision algorithms decompose 3D space into regions (e.g., by superimposing a grid of axis-aligned boxes on the scene) and record which primitives overlap which regions. In some algorithms, the regions may also be adaptively subdivided based on the number of primitives that overlap them. When a ray intersection needs to be found, the sequence of these regions that the ray passes through is computed and only the primitives in the overlapping regions are tested for intersection.

In contrast, object subdivision is based on progressively breaking the objects in the scene down into smaller sets of constituent objects. For example, a model of a room might be broken down into four walls, a ceiling, and a chair. If a ray doesn’t intersect the room’s bounding volume, then all of its primitives can be culled. Otherwise, the ray is tested against each of them. If it hits the chair’s bounding volume, for example, then it might be tested against each of its legs, the seat, and the back. Otherwise, the chair is culled.

Both of these approaches have been quite successful at solving the general problem of ray intersection computational requirements; there’s no fundamental reason to prefer one over the other. The `KdTreeAccel` in this chapter is based on the spatial subdivision approach, and the `BVHAccel` is based on object subdivision.

```
AnimatedTransform:::
MotionBounds()
108
Bounds3f 76
BVHAccel 256
KdTreeAccel 285
Primitive::WorldBound() 249
TransformedPrimitive:::
primitive
252
TransformedPrimitive:::
PrimitiveToWorld
252
```

The Aggregate class provides an interface for grouping multiple Primitive objects together. Because Aggregates themselves implement the Primitive interface, no special support is required elsewhere in pbrt for intersection acceleration. Integrators can be written as if there was just a single Primitive in the scene, checking for intersections without needing to be concerned about how they're actually found. Furthermore, by implementing acceleration in this way, it is easy to experiment with new acceleration techniques by simply adding a new Aggregate primitive to pbrt.

```
(Aggregate Declarations) ≡
class Aggregate : public Primitive {
public:
    (Aggregate Public Methods)
};
```

Like TransformedPrimitives do, the implementation of the Aggregate intersection methods leave the SurfaceInteraction::primitive pointer set to the primitive that the ray actually hit, not the aggregate that holds the primitive. Because pbrt uses this pointer to obtain information about the primitive being hit (its reflection and emission properties), the GetAreaLight(), GetMaterial(), and ComputeScatteringFunctions() methods of Aggregates should never be called, so the implementations of those methods (not shown here) issue a run-time error.

## 4.3 BOUNDING VOLUME HIERARCHIES

Bounding volume hierarchies (BVHs) are an approach for ray intersection acceleration based on primitive subdivision, where the primitives are partitioned into a hierarchy of disjoint sets. (In contrast, spatial subdivision generally partitions space into a hierarchy of disjoint sets.) Figure 4.2 shows a bounding volume hierarchy for a simple scene. Primitives are stored in the leaves, and each node stores a bounding box of the primitives in the nodes beneath it. Thus, as a ray traverses through the tree, any time it doesn't intersect a node's bounds, the subtree beneath that node can be skipped.

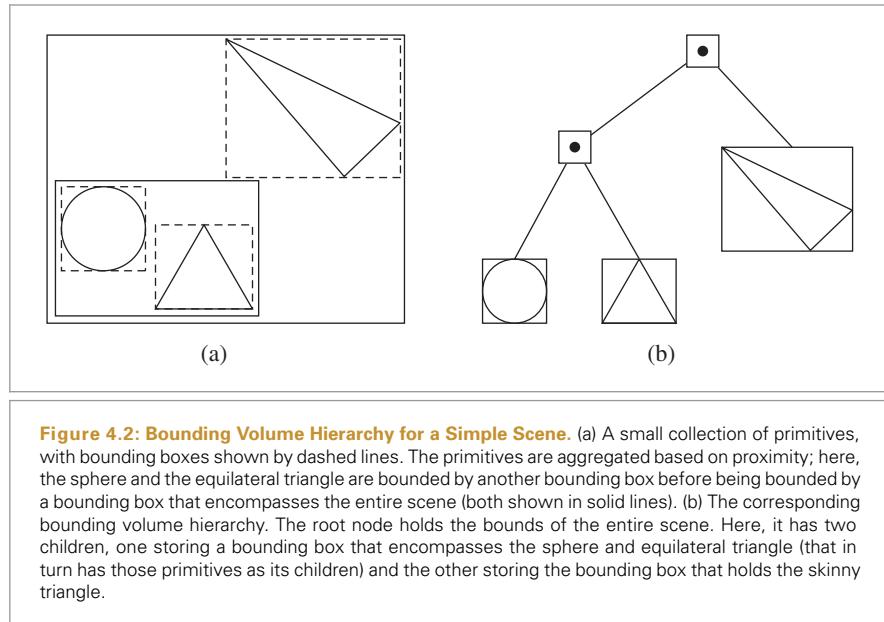
One property of primitive subdivision is that each primitive appears in the hierarchy only once. In contrast, a primitive may overlap multiple spatial regions with spatial subdivision and thus may be tested for intersection multiple times as the ray passes through them.<sup>1</sup> Another implication of this property is that the amount of memory needed to represent the primitive subdivision hierarchy is bounded. For a binary BVH that stores a single primitive in each leaf, the total number of nodes is  $2n - 1$ , where  $n$  is the number of primitives. There will be  $n$  leaf nodes and  $n - 1$  interior nodes. If leaves store multiple primitives, fewer nodes are needed.

BVHs are more efficient to build than kd-trees, which generally deliver slightly faster ray intersection tests than BVHs but take substantially longer to build. On the other hand,

---

Aggregate 255  
 Integrator 25  
 Primitive 248  
 SurfaceInteraction::primitive 249  
 TransformedPrimitive 252

<sup>1</sup> The *mailboxing* technique can be used to avoid these multiple intersections for accelerators that use spatial subdivision, though its implementation can be tricky in the presence of multi-threading. More information on mailboxing is available in the “Further Reading” section.



**Figure 4.2: Bounding Volume Hierarchy for a Simple Scene.** (a) A small collection of primitives, with bounding boxes shown by dashed lines. The primitives are aggregated based on proximity; here, the sphere and the equilateral triangle are bounded by another bounding box before being bounded by a bounding box that encompasses the entire scene (both shown in solid lines). (b) The corresponding bounding volume hierarchy. The root node holds the bounds of the entire scene. Here, it has two children, one storing a bounding box that encompasses the sphere and equilateral triangle (that in turn has those primitives as its children) and the other storing the bounding box that holds the skinny triangle.

BVHs are generally more numerically robust and less prone to missed intersections due to round-off errors than kd-trees are.

The BVH accelerator, `BVHAccel`, is defined in `accelerators/bvh.h` and `accelerators/bvh.cpp`. In addition to the primitives to be stored and the maximum number of primitives that can be in any leaf node, its constructor takes an enumerant value that describes which of four algorithms to use when partitioning primitives to build the tree. The default, `SAH`, indicates that an algorithm based on the “surface area heuristic,” discussed in Section 4.3.2, should be used. An alternative, `HLBVH`, which is discussed in Section 4.3.3, can be constructed more efficiently (and more easily parallelized), but it doesn’t build trees that are as effective as `SAH`. The remaining two approaches use even less computation to build the tree but create fairly low-quality trees.

```
(BVHAccel Public Types) ==
enum class SplitMethod { SAH, HLBVH, Middle, EqualCounts };
```

```
(BVHAccel Method Definitions) ==
BVHAccel::BVHAccel(const std::vector<std::shared_ptr<Primitive>> &p,
                   int maxPrimsInNode, SplitMethod splitMethod)
    : maxPrimsInNode(std::min(255, maxPrimsInNode)), primitives(p),
      splitMethod(splitMethod) {
    if (primitives.size() == 0)
        return;
    {Build BVH from primitives 257}
}
```

`BVHAccel` 256  
`Primitive` 248  
`SplitMethod` 256

```
<BVHAccel Private Data> ≡
    const int maxPrimsInNode;
    const SplitMethod splitMethod;
    std::vector<std::shared_ptr<Primitive>> primitives;
```

### 4.3.1 BVH CONSTRUCTION

There are three stages to BVH construction in the implementation here. First, bounding information about each primitive is computed and stored in an array that will be used during tree construction. Next, the tree is built using the algorithm choice encoded in `splitMethod`. The result is a binary tree where each interior node holds pointers to its children and each leaf node holds references to one or more primitives. Finally, this tree is converted to a more compact (and thus more efficient) pointerless representation for use during rendering. (The implementation is more straightforward with this approach, versus computing the pointerless representation directly during tree construction, which is also possible.)

```
(Build BVH from primitives) ≡
    (Initialize primitiveInfo array for primitives 257)
    (Build BVH tree for primitives using primitiveInfo 258)
    (Compute representation of depth-first traversal of BVH tree 281)
```

256

For each primitive to be stored in the BVH, we store the centroid of its bounding box, its complete bounding box, and its index in the `primitives` array in an instance of the `BVHPrimitiveInfo` structure.

```
(Initialize primitiveInfo array for primitives) ≡
    std::vector<BVHPrimitiveInfo> primitiveInfo(primitives.size());
    for (size_t i = 0; i < primitives.size(); ++i)
        primitiveInfo[i] = { i, primitives[i]->WorldBound() };

(BVHAccel Local Declarations) ≡
    struct BVHPrimitiveInfo {
        BVHPrimitiveInfo(size_t primitiveNumber, const Bounds3f &bounds)
            : primitiveNumber(primitiveNumber), bounds(bounds),
              centroid(.5f * bounds.pMin + .5f * bounds.pMax) { }
        size_t primitiveNumber;
        Bounds3f bounds;
        Point3f centroid;
    };

```

257

Bounds3::pMax 77  
 Bounds3::pMin 77  
 Bounds3f 76  
 BVHAccel::primitives 257  
 BVHBuildNode 258  
 BVHPrimitiveInfo 257  
 Point3f 68  
 Primitive 248  
 Primitive::WorldBound() 249  
 SplitMethod 256

Hierarchy construction can now begin. If the HLBVH construction algorithm has been selected, `HLBVHBuild()` is called to build the tree. The other three construction algorithms are all handled by `recursiveBuild()`. The initial calls to these functions are passed all of the primitives to be stored in the tree. They return a pointer to the root of the tree, which is represented with the `BVHBuildNode` structure. Tree nodes should be allocated with the provided `MemoryArena`, and the total number created should be stored in `*totalNodes`.

One important side effect of the tree construction process is that a new array of pointers to primitives is returned via the `orderedPrims` parameter; this array stores the primitives

ordered so that the primitives in leaf nodes occupy contiguous ranges in the array. It is swapped with the original primitives array after tree construction.

```
(Build BVH tree for primitives using primitiveInfo) ≡ 257
MemoryArena arena(1024 * 1024);
int totalNodes = 0;
std::vector<std::shared_ptr<Primitive>> orderedPrims;
BVHBuildNode *root;
if (splitMethod == SplitMethod::HLBVH)
    root = HLBVHBuild(arena, primitiveInfo, &totalNodes, orderedPrims);
else
    root = recursiveBuild(arena, primitiveInfo, 0, primitives.size(),
                          &totalNodes, orderedPrims);
primitives.swap(orderedPrims);
```

Each `BVHBuildNode` represents a node of the BVH. All nodes store a `Bounds3f`, which represents the bounds of all of the children beneath the node. Each interior node stores pointers to its two children in `children`. Interior nodes also record the coordinate axis along which primitives were partitioned for distribution to their two children; this information is used to improve the performance of the traversal algorithm. Leaf nodes need to record which primitive or primitives are stored in them; the elements of the `BVHAccel::primitives` array from the offset `firstPrimOffset` up to but not including `firstPrimOffset + nPrimitives` are the primitives in the leaf. (Hence the need for reordering the primitives array, so that this representation can be used, rather than, for example, storing a variable-sized array of primitive indices at each leaf node.)

```
(BVHAccel Local Declarations) +≡
struct BVHBuildNode {
    (BVHBuildNode Public Methods 258)
    Bounds3f bounds;
    BVHBuildNode *children[2];
    int splitAxis, firstPrimOffset, nPrimitives;
};
```

We'll distinguish between leaf and interior nodes by whether their `children` pointers have the value `nullptr` or not, respectively.

```
(BVHBuildNode Public Methods) ≡ 258
void InitLeaf(int first, int n, const Bounds3f &b) {
    firstPrimOffset = first;
    nPrimitives = n;
    bounds = b;
    children[0] = children[1] = nullptr;
}
```

The `InitInterior()` method requires that the two children nodes already have been created, so that their pointers can be passed in. This requirement makes it easy to compute the bounds of the interior node, since the children bounds are immediately available.

`Bounds3f` 76  
`BVHAccel::HLBVHBuild()` 271  
`BVHAccel::primitives` 257  
`BVHAccel::recursiveBuild()` 259  
`BVHBuildNode` 258  
`BVHBuildNode::bounds` 258  
`BVHBuildNode::children` 258  
`BVHBuildNode::firstPrimOffset` 258  
`BVHBuildNode::nPrimitives` 258  
`MemoryArena` 1074  
`Primitive` 248  
`SplitMethod` 256

```
(BVHBuildNode Public Methods) +≡
    void InitInterior(int axis, BVHBuildNode *c0, BVHBuildNode *c1) {
        children[0] = c0;
        children[1] = c1;
        bounds = Union(c0->bounds, c1->bounds);
        splitAxis = axis;
        nPrimitives = 0;
    }
```

258

In addition to a `MemoryArena` used for allocating nodes and the array of `BVHPrimitiveInfo` structures, `recursiveBuild()` takes as parameters the range `[start, end]`. It is responsible for returning a BVH for the subset of primitives represented by the range from `primitiveInfo[start]` up to and including `primitiveInfo[end-1]`. If this range covers only a single primitive, then the recursion has bottomed out and a leaf node is created. Otherwise, this method partitions the elements of the array in that range using one of the partitioning algorithms and reorders the array elements in the range accordingly, so that the ranges from `[start, mid]` and `[mid, end)` represent the partitioned subsets. If the partitioning is successful, these two primitive sets are in turn passed to recursive calls that will themselves return pointers to nodes for the two children of the current node.

`totalNodes` tracks the total number of BVH nodes that have been created; this number is used so that exactly the right number of the more compact `LinearBVHNodes` can be allocated later. Finally, the `orderedPrims` array is used to store primitive references as primitives are stored in leaf nodes of the tree. This array is initially empty; when a leaf node is created, `recursiveBuild()` adds the primitives that overlap it to the end of the array, making it possible for leaf nodes to just store an offset into this array and a primitive count to represent the set of primitives that overlap it. Recall that when tree construction is finished, `BVHAccel::primitives` is replaced with the ordered primitives array created here.

```
(BVHAccel Method Definitions) +≡
    BVHBuildNode *BVHAccel::recursiveBuild(MemoryArena &arena,
                                            std::vector<BVHPrimitiveInfo> &primitiveInfo, int start,
                                            int end, int *totalNodes,
                                            std::vector<std::shared_ptr<Primitive>> &orderedPrims) {
        BVHBuildNode *node = arena.Alloc<BVHBuildNode>();
        (*totalNodes)++;
        (Compute bounds of all primitives in BVH node 260)
        int nPrimitives = end - start;
        if (nPrimitives == 1) {
            (Create leaf BVHBuildNode 260)
        } else {
            (Compute bound of primitive centroids, choose split dimension dim 261)
            (Partition primitives into two sets and build children 261)
        }
        return node;
    }
```

Bounds3::Union() 78  
 BVHAccel::primitives 257  
 BVHBuildNode 258  
 BVHBuildNode::bounds 258  
 BVHBuildNode::children 258  
 BVHBuildNode::nPrimitives 258  
 BVHBuildNode::splitAxis 258  
 BVHPrimitiveInfo 257  
 LinearBVHNode 281  
 MemoryArena 1074  
 Primitive 248

*(Compute bounds of all primitives in BVH node) ≡*

259

```
Bounds3f bounds;
for (int i = start; i < end; ++i)
    bounds = Union(bounds, primitiveInfo[i].bounds);
```

At leaf nodes, the primitives overlapping the leaf are appended to the orderedPrims array and a leaf node object is initialized.

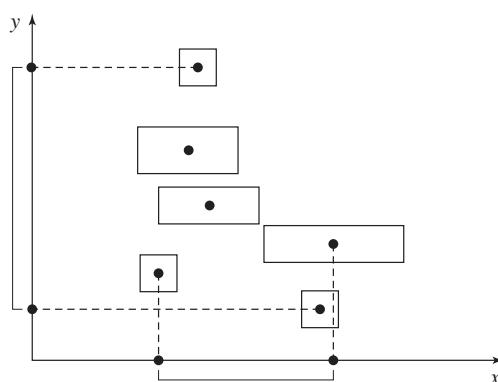
*(Create leaf BVHBuildNode) ≡*

259, 261, 268

```
int firstPrimOffset = orderedPrims.size();
for (int i = start; i < end; ++i) {
    int primNum = primitiveInfo[i].primitiveNumber;
    orderedPrims.push_back(primitives[primNum]);
}
node->InitLeaf(firstPrimOffset, nPrimitives, bounds);
return node;
```

For interior nodes, the collection of primitives must be partitioned between the two children subtrees. Given  $n$  primitives, there are in general  $2^n - 2$  possible ways to partition them into two nonempty groups. In practice when building BVHs, one generally considers partitions along a coordinate axis, meaning that there are about  $6n$  candidate partitions. (Along each axis, each primitive may be put into the first partition or the second partition.)

Here, we choose just one of the three coordinate axes to use in partitioning the primitives. We select the axis associated with the largest extent when projecting the bounding box centroid for the current set of primitives. (An alternative would be to try all three axes and select the one that gave the best result, but in practice this approach works well.) This approach gives good partitions in many scenes; Figure 4.3 illustrates the strategy.



**Figure 4.3: Choosing the Axis along Which to Partition Primitives.** The BVHAccel chooses an axis along which to partition the primitives based on which axis has the largest range of the centroids of the primitives' bounding boxes. Here, in two dimensions, their extent is largest along the  $y$  axis (filled points on the axes), so the primitives will be partitioned in  $y$ .

Bounds3f 76

BVHBuildNode::InitLeaf() 258

BVHPrimitiveInfo::bounds 257

BVHPrimitiveInfo::
 primitiveNumber  
257

The general goal in partitioning here is to select a partition of primitives that doesn't have too much overlap of the bounding boxes of the two resulting primitive sets—if there is substantial overlap then it will more frequently be necessary to traverse both children subtrees when traversing the tree, requiring more computation than if it had been possible to more effectively prune away collections of primitives. This idea of finding effective primitive partitions will be made more rigorous shortly, in the discussion of the surface area heuristic.

*(Compute bound of primitive centroids, choose split dimension dim)* ≡

```
Bounds3f centroidBounds;
for (int i = start; i < end; ++i)
    centroidBounds = Union(centroidBounds, primitiveInfo[i].centroid);
int dim = centroidBounds.MaximumExtent();
```

259

If all of the centroid points are at the same position (i.e., the centroid bounds have zero volume), then recursion stops and a leaf node is created with the primitives; none of the splitting methods here is effective in that (unusual) case. The primitives are otherwise partitioned using the chosen method and passed to two recursive calls to `recursiveBuild()`.

*(Partition primitives into two sets and build children)* ≡

```
int mid = (start + end) / 2;
if (centroidBounds.pMax[dim] == centroidBounds.pMin[dim]) {
    (Create leaf BVHBuildNode 260)
} else {
    (Partition primitives based on splitMethod)
    node->InitInterior(dim,
        recursiveBuild(arena, primitiveInfo, start, mid,
                      totalNodes, orderedPrims),
        recursiveBuild(arena, primitiveInfo, mid, end,
                      totalNodes, orderedPrims));
}
```

259

```
Bounds3::MaximumExtent() 80
Bounds3::pMax 77
Bounds3::pMin 77
Bounds3::Union() 78
Bounds3f 76
BVHAccel::recursiveBuild()
259
BVHAccel::splitMethod 257
BVHBuildNode::InitInterior()
259
BVHPrimitiveInfo::centroid
257
```

We won't include the code fragment *(Partition primitives based on splitMethod)* here; it just uses the value of `BVHAccel::splitMethod` to determine which primitive partitioning scheme to use. These three schemes will be described in the following few pages.

A simple `splitMethod` is `Middle`, which first computes the midpoint of the primitives' centroids along the splitting axis. This method is implemented in the fragment *(Partition primitives through node's midpoint)*. The primitives are classified into the two sets, depending on whether their centroids are above or below the midpoint. This partitioning is easily done with the `std::partition()` C++ standard library function, which takes a range of elements in an array and a comparison function and orders the elements in the array so that all of the elements that return true for the given predicate function appear

in the range before those that return `false` for it.<sup>2</sup> `std::partition()` returns a pointer to the first element that had a `false` value for the predicate, which is converted into an offset into the `primitiveInfo` array so that we can pass it to the recursive call. Figure 4.4 illustrates this approach, including cases where it does and does not work well.

If the primitives all have large overlapping bounding boxes, this splitting method may fail to separate the primitives into two groups. In that case, execution falls through to the `SplitMethod::EqualCounts` approach to try again.

```
(Partition primitives through node's midpoint) ≡
Float pmid = (centroidBounds.pMin[dim] + centroidBounds.pMax[dim]) / 2;
BVHPrimitiveInfo *midPtr =
    std::partition(&primitiveInfo[start], &primitiveInfo[end-1]+1,
    [dim, pmid](const BVHPrimitiveInfo &pi) {
        return pi.centroid[dim] < pmid;
    });
mid = midPtr - &primitiveInfo[0];
if (mid != start && mid != end)
    break;
```

When `splitMethod` is `SplitMethod::EqualCounts`, the *(Partition primitives into equally sized subsets)* fragment runs. It partitions the primitives into two equal-sized subsets such that the first half of the  $n$  of them are the  $n/2$  with smallest centroid coordinate values along the chosen axis, and the second half are the ones with the largest centroid coordinate values. While this approach can sometimes work well, the case in Figure 4.4(b) is one where this method also fares poorly.

This scheme is also easily implemented with a standard library call, `std::nth_element()`. It takes a start, middle, and ending pointer as well as a comparison function. It orders the array so that the element at the middle pointer is the one that would be there if the array was fully sorted, and such that all of the elements before the middle one compare to less than the middle element and all of the elements after it compare to greater than it. This ordering can be done in  $O(n)$  time, with  $n$  the number of elements, which is more efficient than the  $O(n \log n)$  of completely sorting the array.

*(Partition primitives into equally sized subsets)* ≡

265

```
mid = (start + end) / 2;
std::nth_element(&primitiveInfo[start], &primitiveInfo[mid],
    &primitiveInfo[end-1]+1,
    [dim](const BVHPrimitiveInfo &a, const BVHPrimitiveInfo &b) {
        return a.centroid[dim] < b.centroid[dim];
});
```

---

2 In the call to `std::partition()`, note the unusual expression of the indexing of the `primitiveInfo` array, `&primitiveInfo[end-1]+1`. The code is written in this way for somewhat obscure reasons. In the C and C++ programming languages, it is legal to compute the pointer one element past the end of an array so that iteration over array elements can continue until the current pointer is equal to the endpoint. To that end, we would like to just write the expression `&primitiveInfo[end]` here. However, `primitiveInfo` was allocated as a C++ vector; some vector implementations issue a run-time error if the offset passed to their `[]` operator is past the end of the array. Because we're not trying to reference the value of the element one past the end of the array but just compute its address, this operation is in fact safe. Therefore, we compute the same address in the end with the expression used here, while also satisfying any vector error checking.

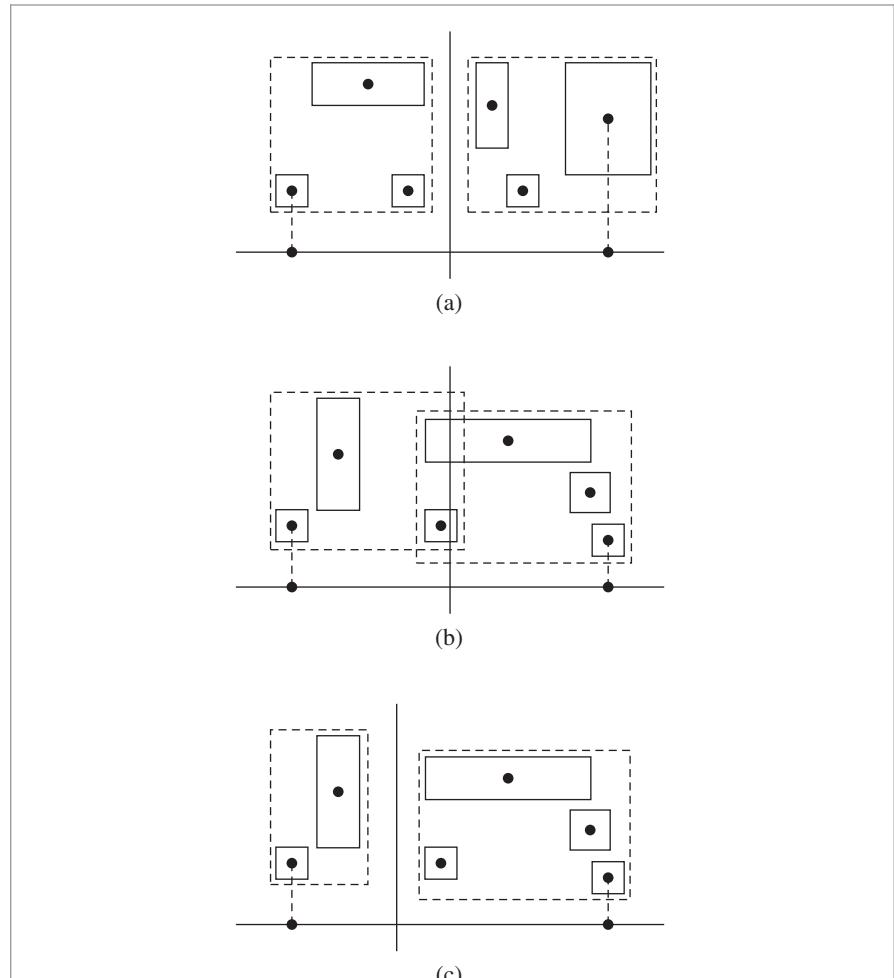
Bounds3::pMax 77

Bounds3::pMin 77

BVHPrimitiveInfo 257

BVHPrimitiveInfo::centroid  
257

Float 1062



**Figure 4.4: Splitting Primitives Based on the Midpoint of Centroids on an Axis.** (a) For some distributions of primitives, such as the one shown here, splitting based on the midpoint of the centroids along the chosen axis works well. (The bounding boxes of the two resulting primitive groups are shown with dashed lines.) (b) For distributions like this one, the midpoint is a suboptimal choice; the two resulting bounding boxes overlap substantially. (c) If the same group of primitives from (b) is instead split along the line shown here, the resulting bounding boxes are smaller and don't overlap at all, leading to better performance when rendering.

#### 4.3.2 THE SURFACE AREA HEURISTIC

The two primitive partitioning approaches above can work well for some distributions of primitives, but they often choose partitions that perform poorly in practice, leading to more nodes of the tree being visited by rays and hence unnecessarily inefficient ray-primitive intersection computations at rendering time. Most of the best current algorithms for building acceleration structures for ray-tracing are based on the “surface

area heuristic” (SAH), which provides a well-grounded cost model for answering questions like “which of a number of partitions of primitives will lead to a better BVH for ray–primitive intersection tests?,” or “which of a number of possible positions to split space in a spatial subdivision scheme will lead to a better acceleration structure?”

The SAH model estimates the computational expense of performing ray intersection tests, including the time spent traversing nodes of the tree and the time spent on ray–primitive intersection tests for a particular partitioning of primitives. Algorithms for building acceleration structures can then follow the goal of minimizing total cost. Typically, a greedy algorithm is used that minimizes the cost for each single node of the hierarchy being built individually.

The ideas behind the SAH cost model are straightforward: at any point in building an adaptive acceleration structure (primitive subdivision or spatial subdivision), we could just create a leaf node for the current region and geometry. In that case, any ray that passes through this region will be tested against all of the overlapping primitives and will incur a cost of

$$\sum_{i=1}^N t_{\text{isect}}(i),$$

where  $N$  is the number of primitives and  $t_{\text{isect}}(i)$  is the time to compute a ray–object intersection with the  $i$ th primitive.

The other option is to split the region. In that case, rays will incur the cost

$$c(A, B) = t_{\text{trav}} + p_A \sum_{i=1}^{N_A} t_{\text{isect}}(a_i) + p_B \sum_{i=1}^{N_B} t_{\text{isect}}(b_i), \quad [4.1]$$

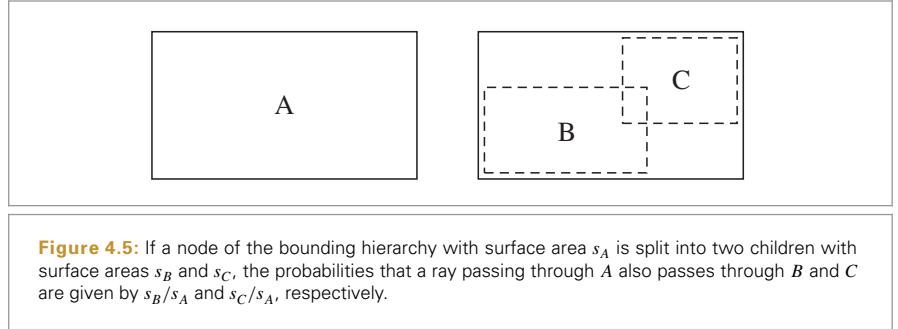
where  $t_{\text{trav}}$  is the time it takes to traverse the interior node and determine which of the children the ray passes through,  $p_A$  and  $p_B$  are the probabilities that the ray passes through each of the child nodes (assuming binary subdivision),  $a_i$  and  $b_i$  are the indices of primitives in the two children nodes, and  $N_A$  and  $N_B$  are the number of primitives that overlap the regions of the two child nodes, respectively. The choice of how primitives are partitioned affects both the values of the two probabilities as well as the set of primitives on each side of the split.

In pbrt, we will make the simplifying assumption that  $t_{\text{isect}}(i)$  is the same for all of the primitives; this assumption is probably not too far from reality, and any error that it introduces doesn’t seem to affect the performance of accelerators very much. Another possibility would be to add a method to `Primitive` that returned an estimate of the number of CPU cycles its intersection test requires.

The probabilities  $p_A$  and  $p_B$  can be computed using ideas from geometric probability. It can be shown that for a convex volume  $A$  contained in another convex volume  $B$ , the conditional probability that a uniformly distributed random ray passing through  $B$  will also pass through  $A$  is the ratio of their surface areas,  $s_A$  and  $s_B$ :

$$p(A|B) = \frac{s_A}{s_B}.$$

Primitive 248



Because we are interested in the cost for rays passing through the node, we can use this result directly. Thus, if we are considering refining a region of space  $A$  such that there are two new subregions with bounds  $B$  and  $C$  (Figure 4.5), the probability that a ray passing through  $A$  will also pass through either of the subregions is easily computed.

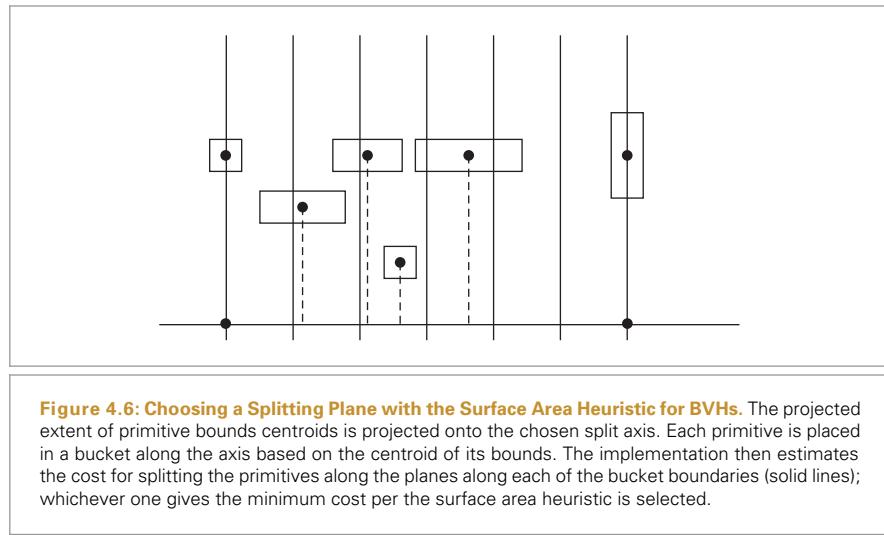
When `splitMethod` has the value `SplitMethod::SAH`, the SAH is used for building the BVH; a partition of the primitives along the chosen axis that gives a minimal SAH cost estimate is found by considering a number of candidate partitions. (This is the default `SplitMethod`, and it creates the most efficient trees for rendering.) However, once it has refined down to a small handful of primitives, the implementation switches over to partitioning into equally sized subsets. The incremental computational cost for applying the SAH at this point isn't worthwhile.

```

⟨Partition primitives using approximate SAH⟩ ≡
  if (nPrimitives <= 4) {
    ⟨Partition primitives into equally sized subsets 262⟩
  } else {
    ⟨Allocate BucketInfo for SAH partition buckets 266⟩
    ⟨Initialize BucketInfo for SAH partition buckets 266⟩
    ⟨Compute costs for splitting after each bucket 267⟩
    ⟨Find bucket to split at that minimizes SAH metric 267⟩
    ⟨Either create leaf or split primitives at selected SAH bucket 268⟩
  }
}

```

Rather than exhaustively considering all  $2n$  possible partitions along the axis, computing the SAH for each to select the best, the implementation here instead divides the range along the axis into a small number of buckets of equal extent. It then only considers partitions at bucket boundaries. This approach is more efficient than considering all partitions while usually still producing partitions that are nearly as effective. This idea is illustrated in Figure 4.6.



*(Allocate BucketInfo for SAH partition buckets) ≡*

265

```
constexpr int nBuckets = 12;
struct BucketInfo {
    int count = 0;
    Bounds3f bounds;
};
BucketInfo buckets[nBuckets];
```

For each primitive in the range, we determine the bucket that its centroid lies in and update the bucket's bounds to include the primitive's bounds.

*(Initialize BucketInfo for SAH partition buckets) ≡*

265

```
for (int i = start; i < end; ++i) {
    int b = nBuckets *
        centroidBounds.Offset(primitiveInfo[i].centroid)[dim];
    if (b == nBuckets) b = nBuckets - 1;
    buckets[b].count++;
    buckets[b].bounds = Union(buckets[b].bounds, primitiveInfo[i].bounds);
}
```

For each bucket, we now have a count of the number of primitives and the bounds of all of their respective bounding boxes. We want to use the SAH to estimate the cost of splitting at each of the bucket boundaries. The fragment below loops over all of the buckets and initializes the cost[i] array to store the estimated SAH cost for splitting after the *i*th bucket. (It doesn't consider a split after the last bucket, which by definition wouldn't split the primitives.)

We arbitrarily set the estimated intersection cost to 1, and then set the estimated traversal cost to 1/8. (One of the two of them can always be set to 1 since it is the relative, rather than absolute, magnitudes of the estimated traversal and intersection costs that

Bounds3::Offset() 81  
 Bounds3f 76  
 BucketInfo::bounds 266  
 BucketInfo::count 266  
 BVHPrimitiveInfo::bounds 257  
 BVHPrimitiveInfo::centroid 257

determine their effect.) While the absolute amount of computation for node traversal—a ray–bounding box intersection—is only slightly less than the amount of computation needed to intersect a ray with a shape, ray–primitive intersection tests in `pbrt` go through two virtual function calls, which add significant overhead, so we estimate their cost here as eight times more than the ray–box intersection.

This computation has  $O(n^2)$  complexity in the number of buckets, though a linear-time implementation based on a forward scan over the buckets and a backward scan over the buckets that incrementally compute and store bounds and counts is possible. For the small  $n$  here, the performance impact is generally acceptable, though for a more highly optimized renderer addressing this inefficiency may be worthwhile.

```
(Compute costs for splitting after each bucket) ≡ 265
    Float cost[nBuckets - 1];
    for (int i = 0; i < nBuckets - 1; ++i) {
        Bounds3f b0, b1;
        int count0 = 0, count1 = 0;
        for (int j = 0; j <= i; ++j) {
            b0 = Union(b0, buckets[j].bounds);
            count0 += buckets[j].count;
        }
        for (int j = i+1; j < nBuckets; ++j) {
            b1 = Union(b1, buckets[j].bounds);
            count1 += buckets[j].count;
        }
        cost[i] = .125f + (count0 * b0.SurfaceArea() +
                           count1 * b1.SurfaceArea()) / bounds.SurfaceArea();
    }
```

Given all of the costs, a linear scan through the cost array finds the partition with minimum cost.

```
(Find bucket to split at that minimizes SAH metric) ≡ 265
    Float minCost = cost[0];
    int minCostSplitBucket = 0;
    for (int i = 1; i < nBuckets - 1; ++i) {
        if (cost[i] < minCost) {
            minCost = cost[i];
            minCostSplitBucket = i;
        }
    }
```

`Bounds3::SurfaceArea()` 80  
`Bounds3::Union()` 78  
`Bounds3f` 76  
`BucketInfo::bounds` 266  
`BucketInfo::count` 266  
`Float` 1062

If the chosen bucket boundary for partitioning has a lower estimated cost than building a node with the existing primitives or if more than the maximum number of primitives allowed in a node is present, the `std::partition()` function is used to do the work of reordering nodes in the `primitiveInfo` array. Recall from its usage earlier that this function ensures that all elements of the array that return `true` from the given predicate appear before those that return `false` and that it returns a pointer to the first element where the predicate returns `false`. Because we arbitrarily set the estimated intersection

cost to 1 previously, the estimated cost for just creating a leaf node is equal to the number of primitives, `nPrimitives`.

*(Either create leaf or split primitives at selected SAH bucket) ≡* 265

```

Float leafCost = nPrimitives;
if (nPrimitives > maxPrimsInNode || minCost < leafCost) {
    BVHPrimitiveInfo *pmid = std::partition(&primitiveInfo[start],
                                             &primitiveInfo[end-1]+1,
                                             [=](const BVHPrimitiveInfo &pi) {
        int b = nBuckets * centroidBounds.Offset(pi.centroid)[dim];
        if (b == nBuckets) b = nBuckets - 1;
        return b <= minCostSplitBucket;
    });
    mid = pmid - &primitiveInfo[0];
} else {
    (Create leaf BVHBuildNode 260)
}

```

### 4.3.3 LINEAR BOUNDING VOLUME HIERARCHIES

While building bounding volume hierarchies using the surface area heuristic gives very good results, that approach does have two disadvantages: first, many passes are taken over the scene primitives to compute the SAH costs at all of the levels of the tree. Second, top-down BVH construction is difficult to parallelize well: the most obvious parallelization approach—performing parallel construction of independent subtrees—suffers from limited independent work until the top few levels of the tree have been built, which in turn inhibits parallel scalability. (This second issue is particularly an issue on GPUs, which perform poorly if massive parallelism isn’t available.)

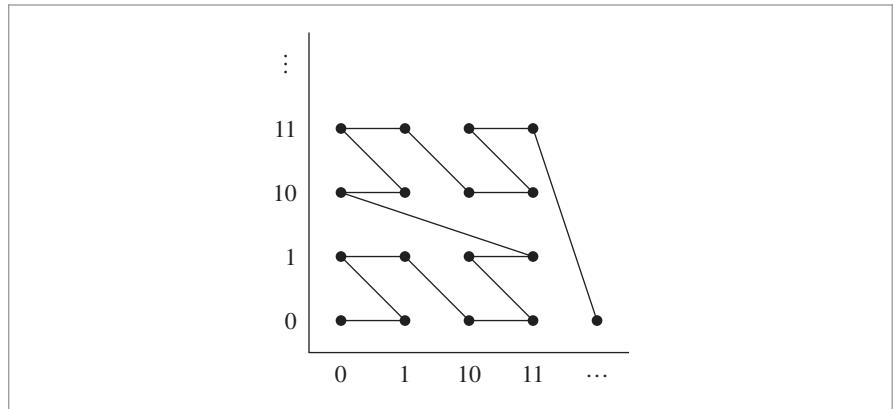
*Linear bounding volume hierarchies* (LBVs) were developed to address these issues. With LBVs, the tree is built with a small number of lightweight passes over the primitives; tree construction time is linear in the number of primitives. Further, the algorithm quickly partitions the primitives into clusters that can be processed independently. This processing can be fairly easily parallelized and is well suited to GPU implementation.

The key idea behind LBVs is to turn BVH construction into a sorting problem. Because there’s no single ordering function for sorting multi-dimensional data, LBVs are based on *Morton codes*, which map nearby points in  $n$  dimensions to nearby points along the 1D line, where there is an obvious ordering function. After the primitives have been sorted, spatially nearby clusters of primitives are in contiguous segments of the sorted array.

Morton codes are based on a simple transformation: given  $n$ -dimensional integer coordinate values, their Morton-coded representation is found by interleaving the bits of the coordinates in base 2. For example, consider a 2D coordinate  $(x, y)$  where the bits of  $x$  and  $y$  are denoted by  $x_i$  and  $y_i$ . The corresponding Morton-coded value is

$$\cdots y_3 \, x_3 \, y_2 \, x_2 \, y_1 \, x_1 \, y_0 \, x_0.$$

`Bounds3::Offset()` [81](#)  
`BVHAccel::maxPrimsInNode` [257](#)  
`BVHPrimitiveInfo` [257](#)  
`Float` [1062](#)



**Figure 4.7: The Order That Points Are Visited along the Morton Curve.** Coordinate values along the  $x$  and  $y$  axes are shown in binary. If we connect the integer coordinate points in the order of their Morton indices, we see that the Morton curve visits the points along a hierarchical “z”-shaped path.

Figure 4.7 shows a plot of the 2D points in Morton order—note that they are visited along a path that follows a reversed “z” shape. (The Morton path is sometimes called “z-order” for this reason.) We can see that points with coordinates that are close together in 2D are generally close together along the Morton curve.<sup>3</sup>

A Morton-encoded value also encodes useful information about the position of the point that it represents. Consider the case of 4-bit coordinate values in 2D: the  $x$  and  $y$  coordinates are integers in  $[0, 15]$  and the Morton code has 8 bits:  $y_3 \ x_3 \ y_2 \ x_2 \ y_1 \ x_1 \ y_0 \ x_0$ . Many interesting properties follow from the encoding; a few examples include:

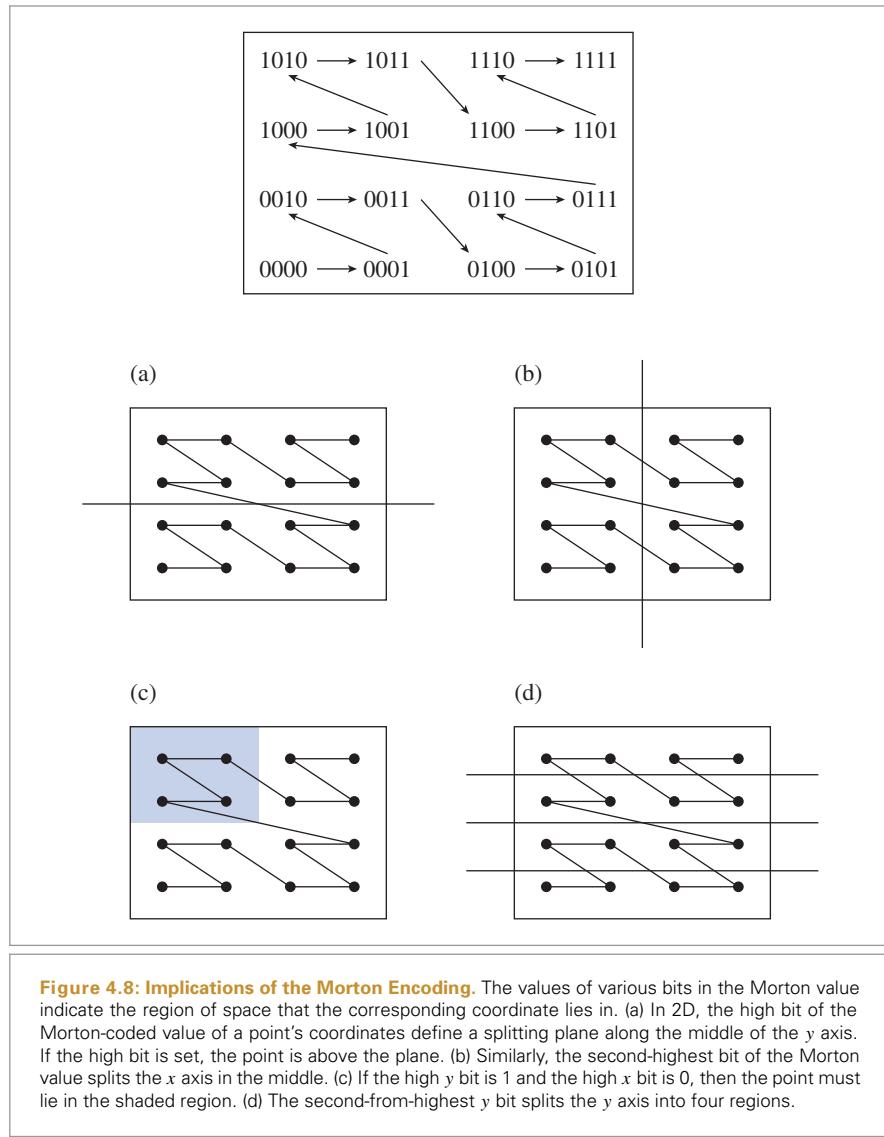
- For a Morton-encoded 8-bit value where the high bit  $y_3$  is set, then we know that the high bit of its underlying  $y$  coordinate is set and thus  $y \geq 8$  (Figure 4.8(a)).
- The next bit value,  $x_3$ , splits the  $x$  axis in the middle (Figure 4.8(b)). If  $y_3$  is set and  $x_3$  is off, for example, then the corresponding point must lie in the shaded area of Figure 4.8(c). In general, points with a number of matching high bits lie in a power-of-two sized and aligned region of space determined by the matching bit values.
- The value of  $y_2$  splits the  $y$  axis into four regions (Figure 4.8(d)).

Another way to interpret these bit-based properties is in terms of Morton-coded values. For example, Figure 4.8(a) corresponds to the index being in the range [128, 255], and Figure 4.8(c) corresponds to [128, 191]. Thus, given a set of sorted Morton indices, we could find the range of points corresponding to an area like Figure 4.8(c) by performing a binary search to find each endpoint in the array.

LBVHs are BVHs built by partitioning primitives using splitting planes that are at the midpoint of each region of space (i.e., equivalent to the `SplitMethod::Middle` path

---

<sup>3</sup> Many GPUs store texture maps in memory using a Morton layout. One advantage of doing so is that when performing bilinear interpolation between four texel values, the values are much more likely to be close together in memory than if the texture is laid out in scanline order. In turn, texture cache performance benefits.



defined earlier). Partitioning is extremely efficient, as it's based on properties of the Morton encoding described above.

Just reimplementing `Middle` in a different manner isn't particularly interesting, so in the implementation here, we'll build a *hierarchical linear bounding volume hierarchy* (HLBVH). With this approach, Morton-curve-based clustering is used to first build trees for the lower levels of the hierarchy (referred to as "treelets" in the following) and the top levels of the tree are then created using the surface area heuristic. The `HLBVHBuild()` method implements this approach and returns the root node of the resulting tree.

*(BVHAccel Method Definitions) +≡*

```
BVHBuildNode *BVHAccel::HLBVHBuild(MemoryArena &arena,
    const std::vector<BVHPrimitiveInfo> &primitiveInfo,
    int *totalNodes,
    std::vector<std::shared_ptr<Primitive>> &orderedPrims) const {
    (Compute bounding box of all primitive centroids 271)
    (Compute Morton indices of primitives 271)
    (Radix sort primitive Morton indices 273)
    (Create LBVH treelets at bottom of BVH 275)
    (Create and return SAH BVH from LBVH treelets 280)
}
```

The BVH is built using only the centroids of primitive bounding boxes to sort them—it doesn't account for the actual spatial extent of each primitive. This simplification is critical to the performance that HLVBVs offer, but it also means that for scenes with primitives that span a wide range of sizes, the tree that is built won't account for this variation as an SAH-based tree would.

Because the Morton encoding operates on integer coordinates, we first need to bound the centroids of all of the primitives so that we can quantize centroid positions with respect to the overall bounds.

*(Compute bounding box of all primitive centroids) ≡*

```
Bounds3f bounds;
for (const BVHPrimitiveInfo &pi : primitiveInfo)
    bounds = Union(bounds, pi.centroid);
```

271

Given the overall bounds, we can now compute the Morton code for each primitive. This is a fairly lightweight calculation, but given that there may be millions of primitives, it's worth parallelizing. Note that a loop chunk size of 512 is passed to `ParallelFor()` below; this causes worker threads to be given groups of 512 primitives to process rather than one at a time as would otherwise be the default. Because the amount of work performed per primitive to compute the Morton code is relatively small, this granularity better amortizes the overhead of distributing tasks to the worker threads.

*(Compute Morton indices of primitives) ≡*

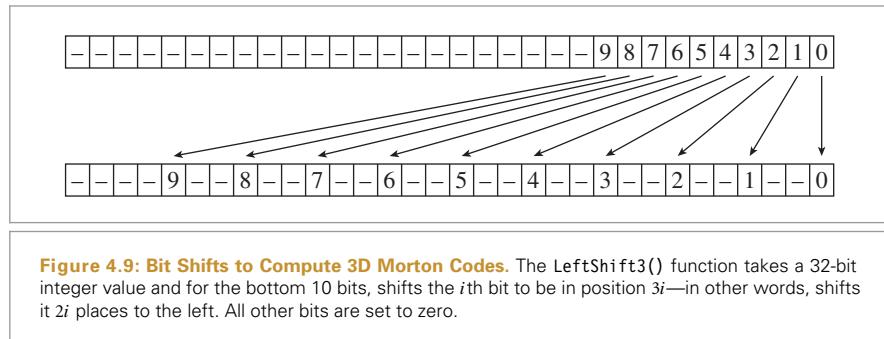
```
std::vector<MortonPrimitive> mortonPrims(primitiveInfo.size());
ParallelFor(
    [&](int i) {
        (Initialize mortonPrims[i] for ith primitive 272)
    }, primitiveInfo.size(), 512);
```

271

A `MortonPrimitive` instance is created for each primitive; it stores the index of the primitive in the `primitiveInfo` array as well as its Morton code.

*(BVHAccel Local Declarations) +≡*

```
struct MortonPrimitive {
    int primitiveIndex;
    uint32_t mortonCode;
};
```



We use 10 bits for each of the  $x$ ,  $y$ , and  $z$  dimensions, giving a total of 30 bits for the Morton code. This granularity allows the values to fit into a single 32-bit variable. Floating-point centroid offsets inside the bounding box are in  $[0, 1]$ , so we scale them by  $2^{10}$  to get integer coordinates that fit in 10 bits. (For the edge case of offsets exactly equal to 1, an out-of-range quantized value of 1024 may result; this case is handled in the forthcoming `EncodeMorton3()` function.)

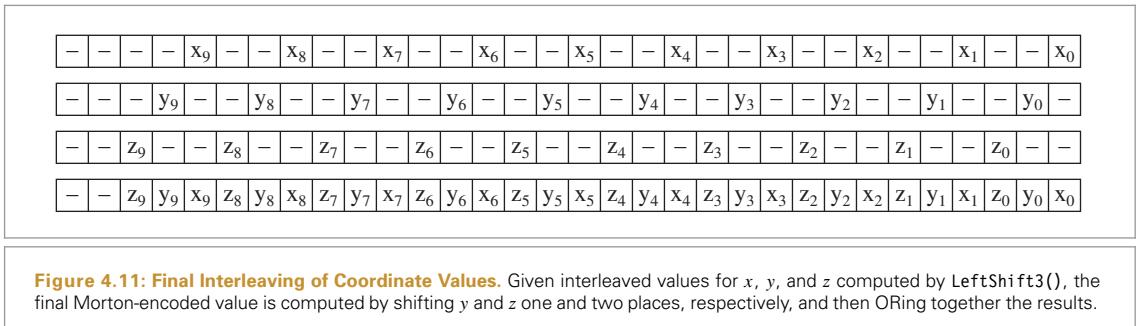
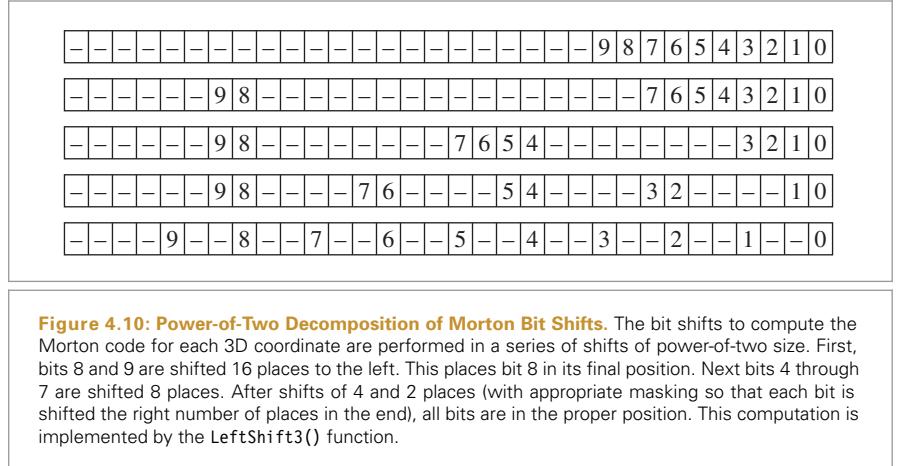
```
<Initialize mortonPrims[i] for ith primitive> ≡ 271
    constexpr int mortonBits = 10;
    constexpr int mortonScale = 1 << mortonBits;
    mortonPrims[i].primitiveIndex = primitiveInfo[i].primitiveNumber;
    Vector3f centroidOffset = bounds.Offset(primitiveInfo[i].centroid);
    mortonPrims[i].mortonCode = EncodeMorton3(centroidOffset * mortonScale);
```

To compute 3D Morton codes, first we'll define a helper function: `LeftShift3()` takes a 32-bit value and returns the result of shifting the  $i$ th bit to be at the  $3i$ th bit, leaving zeros in other bits. Figure 4.9 illustrates this operation.

The most obvious approach to implement this operation, shifting each bit value individually, isn't the most efficient. (It would require a total of 9 shifts, along with logical ORs to compute the final value.) Instead, we can decompose each bit's shift into multiple shifts of power-of-two size that together shift the bit's value to its final position. Then, all of the bits that need to be shifted a given power-of-two number of places can be shifted together. The `LeftShift3()` function implements this computation, and Figure 4.10 shows how it works.

```
<BVHAccel Utility Functions> ≡
    inline uint32_t LeftShift3(uint32_t x) {
        if (x == (1 << 10)) --x;
        x = (x | (x << 16)) & 0b0000001100000000000000001111111;
        x = (x | (x << 8)) & 0b0000001100000000111000000001111;
        x = (x | (x << 4)) & 0b00000011000011000011000011000011;
        x = (x | (x << 2)) & 0b00001001001001001001001001001;
        return x;
    }
```

Bounds3::Offset() 81  
 BVHPrimitiveInfo::centroid 257  
 BVHPrimitiveInfo::primitiveNumber 257  
 EncodeMorton3() 273  
 MortonPrimitive::mortonCode 271  
 MortonPrimitive::primitiveIndex 271  
 Vector3f 60



`EncodeMorton3()` takes a 3D coordinate value where each component is a floating-point value between 0 and  $2^{10}$ . It converts these values to integers and then computes the Morton code by expanding the three 10-bit quantized values so that their  $i$ th bits are at position  $3i$ , then shifting the  $y$  bits over one more, the  $z$  bits over two more, and ORing together the result (Figure 4.11).

```
{BVHAccel Utility Functions} +≡
    inline uint32_t EncodeMorton3(const Vector3f &v) {
        return (LeftShift3(v.z) << 2) | (LeftShift3(v.y) << 1) |
            LeftShift3(v.x);
    }
```

`LeftShift3()` 272  
`RadixSort()` 274  
`Vector3f` 60

Once the Morton indices have been computed, we'll sort the `mortonPrims` by Morton index value using a radix sort. We have found that for BVH construction, our radix sort implementation is noticeably faster than using `std::sort()` from our system's standard library (which is a mixture of a quicksort and an insertion sort).

```
{Radix sort primitive Morton indices} ≡
    RadixSort(&mortonPrims);
```

Recall that a radix sort differs from most sorting algorithms in that it isn't based on comparing pairs of values but rather is based on bucketing items based on some key. Radix sort can be used to sort integer values by sorting them one digit at a time, going from the rightmost digit to the leftmost. Especially with binary values, it's worth sorting multiple digits at a time; doing so reduces the total number of passes taken over the data. In the implementation here, `bitsPerPass` sets the number of bits processed per pass; with the value 6, we have 5 passes to sort the 30 bits.

```
(BVHAccel Utility Functions) +≡
static void RadixSort(std::vector<MortonPrimitive> *v) {
    std::vector<MortonPrimitive> tempVector(v->size());
    constexpr int bitsPerPass = 6;
    constexpr int nBits = 30;
    constexpr int nPasses = nBits / bitsPerPass;
    for (int pass = 0; pass < nPasses; ++pass) {
        <Perform one pass of radix sort, sorting bitsPerPass bits 274>
    }
    <Copy final result from tempVector, if needed 275>
}
```

The current pass will sort `bitsPerPass` bits, starting at `lowBit`.

```
<Perform one pass of radix sort, sorting bitsPerPass bits> ≡ 274
int lowBit = pass * bitsPerPass;
<Set in and out vector pointers for radix sort pass 274>
<Count number of zero bits in array for current radix sort bit 274>
<Compute starting index in output array for each bucket 275>
<Store sorted values in output array 275>
```

The `in` and `out` references correspond to the vector to be sorted and the vector to store the sorted values in, respectively. Each pass through the loop alternates between the input vector `*v` and the temporary vector for each of them.

```
<Set in and out vector pointers for radix sort pass> ≡ 274
std::vector<MortonPrimitive> &in = (pass & 1) ? tempVector : *v;
std::vector<MortonPrimitive> &out = (pass & 1) ? *v : tempVector;
```

If we're sorting  $n$  bits per pass, then there are  $2^n$  buckets that each value may land in. We first count how many values will land in each bucket; this will let us determine where to store sorted values in the output array. To compute the bucket index for the current value, the implementation shifts the index so that the bit at index `lowBit` is at bit 0 and then masks off the low `bitsPerPass` bits.

```
<Count number of zero bits in array for current radix sort bit> ≡ 274
constexpr int nBuckets = 1 << bitsPerPass;
int bucketCount[nBuckets] = { 0 };
constexpr int bitMask = (1 << bitsPerPass) - 1;
for (const MortonPrimitive &mp : in) {
    int bucket = (mp.mortonCode >> lowBit) & bitMask;
    ++bucketCount[bucket];
}
```

MortonPrimitive 271

MortonPrimitive::mortonCode  
271

Given the count of how many values land in each bucket, we can compute the offset in the output array where each bucket's values start; this is just the sum of how many values land in the preceding buckets.

*(Compute starting index in output array for each bucket) ≡*

```
int outIndex[nBuckets];
outIndex[0] = 0;
for (int i = 1; i < nBuckets; ++i)
    outIndex[i] = outIndex[i - 1] + bucketCount[i - 1];
```

274

Now that we know where to start storing values for each bucket, we can take another pass over the primitives to recompute the bucket that each one lands in and to store their MortonPrimitives in the output array. This completes the sorting pass for the current group of bits.

*(Store sorted values in output array) ≡*

```
for (const MortonPrimitive &mp : in) {
    int bucket = (mp.mortonCode >> lowBit) & bitMask;
    out[outIndex[bucket]++] = mp;
}
```

274

When sorting is done, if an odd number of radix sort passes were performed, then the final sorted values need to be copied from the temporary vector to the output vector that was originally passed to RadixSort().

*(Copy final result from tempVector, if needed) ≡*

```
if (nPasses & 1)
    std::swap(*v, tempVector);
```

274

Given the sorted array of primitives, we'll find clusters of primitives with nearby centroids and then create an LBVH over the primitives in each cluster. This step is a good one to parallelize as there are generally many clusters and each cluster can be processed independently.

*(Create LBVH treelets at bottom of BVH) ≡*

*(Find intervals of primitives for each treelet 276)*

*(Create LBVHs for treelets in parallel 277)*

271

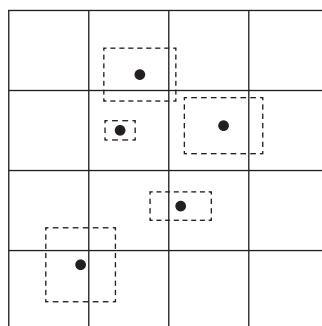
Each primitive cluster is represented by an LBVHTreelet. It encodes the index in the mortonPrims array of the first primitive in the cluster as well as the number of following primitives. (See Figure 4.12.)

*(BVHAccel Local Declarations) +≡*

```
struct LBVHTreelet {
    int startIndex, nPrimitives;
    BVHBuildNode *buildNodes;
};
```

BVHBuildNode 258  
MortonPrimitive 271  
MortonPrimitive::mortonCode  
271

Recall from Figure 4.8 that a set of points with Morton codes that match in their high bit values lie in a power-of-two aligned and sized subset of the original volume. Because we have already sorted the mortonPrims array by Morton-coded value, primitives with matching high bit values are already together in contiguous sections of the array.



**Figure 4.12: Primitive Clusters for LBVH Treelets.** Primitive centroids are clustered in a uniform grid over their bounds. An LBVH is created for each cluster of primitives within a cell that are in contiguous sections of the sorted Morton index values.

Here we'll find sets of primitives that have the same values for the high 12 bits of their 30-bit Morton codes. Clusters are found by taking a linear pass through the `mortonPrims` array and finding the offsets where any of the high 12 bits changes. This corresponds to clustering primitives in a regular grid of  $2^{12} = 4096$  total grid cells with  $2^4 = 16$  cells in each dimension. In practice, many of the grid cells will be empty, though we'll still expect to find many independent clusters here.

*(Find intervals of primitives for each treelet) ≡* 275

```
std::vector<LBVHTreelet> treeletsToBuild;
for (int start = 0, end = 1; end <= (int)mortonPrims.size(); ++end) {
    uint32_t mask = 0b00111111111110000000000000000000;
    if (end == (int)mortonPrims.size() ||
        ((mortonPrims[start].mortonCode & mask) !=
         (mortonPrims[end].mortonCode & mask))) {
        {Add entry to treeletsToBuild for this treelet 277}
        start = end;
    }
}
```

When a cluster of primitives has been found for a treelet, `BVHBuildNode`s are immediately allocated for it. (Recall that the number of nodes in a BVH is bounded by twice the number of leaf nodes, which in turn is bounded by the number of primitives). It's simpler to pre-allocate this memory now in a serial phase of execution than during parallel construction of LBVHs.

One important detail here is the `false` value passed to `MemoryArena::Alloc()`; it indicates that the constructors of the underlying objects being allocated should not be executed. To our surprise, running the `BVHBuildNode` constructors introduced significant overhead and meaningfully reduced overall HLBVH construction performance. Because all of the members of `BVHBuildNode` will be initialized in code to follow, the initialization performed by the constructor is unnecessary here in any case.

`BVHBuildNode` 258

`LBVHTreelet` 275

`MemoryArena::Alloc()` 1074

`MortonPrimitive::mortonCode`

271

```
{Add entry to treeletsToBuild for this treelet} ≡ 276
    int nPrimitives = end - start;
    int maxBVHNodes = 2 * nPrimitives;
    BVHBuildNode *nodes = arena.Alloc<BVHBuildNode>(maxBVHNodes, false);
    treeletsToBuild.push_back({start, nPrimitives, nodes});
```

Once the primitives for each treelet have been identified, we can create LBVs for them in parallel. When construction is finished, the `buildNodes` pointer for each `LBVHTreelet` will point to the root of the corresponding LBVH.

There are two places where the worker threads building LBVs must coordinate with each other. First, the total number of nodes in all of the LBVs needs to be computed and returned via the `totalNodes` pointer passed to `HLBVHBuild()`. Second, when leaf nodes are created for the LBVs, a contiguous segment of the `orderedPrims` array is needed to record the indices of the primitives in the leaf node. Our implementation uses atomic variables for both—`atomicTotal` to track the number of nodes and `orderedPrimsOffset` for the index of the next available entry in `orderedPrims`.

```
{Create LBVs for treelets in parallel} ≡ 275
    std::atomic<int> atomicTotal(0), orderedPrimsOffset(0);
    orderedPrims.resize(primitives.size());
    ParallelFor(
        [&](int i) {
            (Generate ith LBVH treelet 277)
        }, treeletsToBuild.size());
    *totalNodes = atomicTotal;
```

The work of building the treelet is performed by `emitLBVH()`, which takes primitives with centroids in some region of space and successively partitions them with splitting planes that divide the current region of space into two halves along the center of the region along one of the three axes.

Note that instead of taking a pointer to the atomic variable `atomicTotal` to count the number of nodes created, `emitLBVH()` updates a non-atomic local variable. The fragment here then only updates `atomicTotal` once per treelet when each treelet is done. This approach gives measurably better performance than the alternative—having the worker threads frequently modify `atomicTotal` over the course of their execution. (See the discussion of the overhead of multi-core memory coherence models in Appendix A.6.1.)

```
{Generate ith LBVH treelet} ≡ 277
BVHAccel::emitLBVH() 278
BVHBuildNode 258
LBVHTreelet 275
LBVHTreelet::buildNodes 275
LBVHTreelet::nPrimitives 275
LBVHTreelet::startIndex 275
ParallelFor() 1088
    int nodesCreated = 0;
    const int firstBitIndex = 29 - 12;
    LBVHTreelet &tr = treeletsToBuild[i];
    tr.buildNodes =
        emitLBVH(tr.buildNodes, primitiveInfo, &mortonPrims[tr.startIndex],
                  tr.nPrimitives, &nodesCreated, orderedPrims,
                  &orderedPrimsOffset, firstBitIndex);
    atomicTotal += nodesCreated;
```

Thanks to the Morton encoding, the current region of space doesn't need to be explicitly represented in `emitLBVH()`: the sorted `MortonPrims` passed in have some number of matching high bits, which in turn corresponds to a spatial bound. For each of the remaining bits in the Morton codes, this function tries to split the primitives along the plane corresponding to the bit `bitIndex` (recall Figure 4.8(d)) and then calls itself recursively. The index of the next bit to try splitting with is passed as the last argument to the function: initially it's  $29 - 12$ , since 29 is the index of the 30th bit with zero-based indexing, and we previously used the high 12 bits of the Morton-coded value to cluster the primitives; thus, we know that those bits must all match for the cluster.

```
(BVHAccel Method Definitions) +≡
BVHBuildNode *BVHAccel::emitLBVH(BVHBuildNode *&buildNodes,
    const std::vector<BVHPrimitiveInfo> &primitiveInfo,
    MortonPrimitive *mortonPrims, int nPrimitives, int *totalNodes,
    std::vector<std::shared_ptr<Primitive>> &orderedPrims,
    std::atomic<int> *orderedPrimsOffset, int bitIndex) const {
    if (bitIndex == -1 || nPrimitives < maxPrimsInNode) {
        ⟨Create and return leaf node of LBVH treelet 278⟩
    } else {
        int mask = 1 << bitIndex;
        ⟨Advance to next subtree level if there's no LBVH split for this bit 279⟩
        ⟨Find LBVH split point for this dimension 279⟩
        ⟨Create and return interior LBVH node 279⟩
    }
}
```

After `emitLBVH()` has partitioned the primitives with the final low bit, no more splitting is possible and a leaf node is created. Alternatively, it also stops and makes a leaf node if it's down to a small number of primitives.

Recall that `orderedPrimsOffset` is the offset to the next available element in the `orderedPrims` array. Here, the call to `fetch_add()` atomically adds the value of `nPrimitives` to `orderedPrimsOffset` and returns its old value before the addition. Because these operations are atomic, multiple LBVH construction threads can concurrently carve out space in the `orderedPrims` array without data races. Given space in the array, leaf construction is similar to the approach implemented earlier in `(Create leaf BVHBuildNode)`.

```
(Create and return leaf node of LBVH treelet) ≡
(*totalNodes)++;
BVHBuildNode *node = buildNodes++;
Bounds3f bounds;
int firstPrimOffset = orderedPrimsOffset->fetch_add(nPrimitives);
for (int i = 0; i < nPrimitives; ++i) {
    int primitiveIndex = mortonPrims[i].primitiveIndex;
    orderedPrims[firstPrimOffset + i] = primitives[primitiveIndex];
    bounds = Union(bounds, primitiveInfo[primitiveIndex].bounds);
}
node->InitLeaf(firstPrimOffset, nPrimitives, bounds);
return node;
```

278  
 Bounds3::Union() 78  
 Bounds3f 76  
 BVHAccel::maxPrimsInNode 257  
 BVHBuildNode 258  
 BVHBuildNode::InitLeaf() 258  
 BVHPrimitiveInfo 257  
 BVHPrimitiveInfo::bounds 257  
 MortonPrimitive 271  
 MortonPrimitive::primitiveIndex 271  
 Primitive 248

It may be the case that all of the primitives lie on the same side of the splitting plane; since the primitives are sorted by their Morton index, this case can be efficiently checked by seeing if the first and last primitive in the range both have the same bit value for this plane. In this case, `emitLBVH()` proceeds to the next bit without unnecessarily creating a node.

```
(Advance to next subtree level if there's no LBVH split for this bit) ≡ 278
if ((mortonPrims[0].mortonCode & mask) ==
    (mortonPrims[nPrimitives - 1].mortonCode & mask))
    return emitLBVH(buildNodes, primitiveInfo, mortonPrims, nPrimitives,
                    totalNodes, orderedPrims, orderedPrimsOffset,
                    bitIndex - 1);
```

If there are primitives on both sides of the splitting plane, then a binary search efficiently finds the dividing point where the `bitIndex`th bit goes from 0 to 1 in the current set of primitives.

```
(Find LBVH split point for this dimension) ≡ 278
int searchStart = 0, searchEnd = nPrimitives - 1;
while (searchStart + 1 != searchEnd) {
    int mid = (searchStart + searchEnd) / 2;
    if ((mortonPrims[searchStart].mortonCode & mask) ==
        (mortonPrims[mid].mortonCode & mask))
        searchStart = mid;
    else
        searchEnd = mid;
}
int splitOffset = searchEnd;
```

Given the split offset, the method can now claim a node to use as an interior node and recursively build LBVHs for both partitioned sets of primitives. Note a further efficiency benefit from Morton encoding: entries in the `mortonPrims` array don't need to be copied or reordered for the partition: because they are all sorted by their Morton code value and because it is processing bits from high to low, the two spans of primitives are already on the correct sides of the partition plane.

```
(Create and return interior LBVH node) ≡ 278
(*totalNodes)++;
BVHBuildNode *node = buildNodes++;
BVHBuildNode *lbvh[2] = {
    emitLBVH(buildNodes, primitiveInfo, mortonPrims, splitOffset,
              totalNodes, orderedPrims, orderedPrimsOffset, bitIndex - 1),
    emitLBVH(buildNodes, primitiveInfo, &mortonPrims[splitOffset],
              nPrimitives - splitOffset, totalNodes, orderedPrims,
              orderedPrimsOffset, bitIndex - 1)
};
int axis = bitIndex % 3;
node->InitInterior(axis, lbvh[0], lbvh[1]);
return node;
```

BVHAccel::emitLBVH() 278  
BVHBuildNode 258  
BVHBuildNode::InitInterior() 259  
MortonPrimitive::mortonCode 271

Once all of the LBVH treelets have been created, `buildUpperSAH()` creates a BVH of all the treelets. Since there are generally tens or hundreds of them (and in any case, no more than 4096), this step takes very little time.

*(Create and return SAH BVH from LBVH treelets) ≡* 271

```
std::vector<BVHBuildNode *> finishedTreelets;
for (LBVHTreelet &treelet : treeletsToBuild)
    finishedTreelets.push_back(treelet.buildNodes);
return buildUpperSAH(arena, finishedTreelets, 0,
                     finishedTreelets.size(), totalNodes);
```

The implementation of this method isn't included here, as it follows the same approach as fully SAH-based BVH construction, just over treelet root nodes rather than scene primitives.

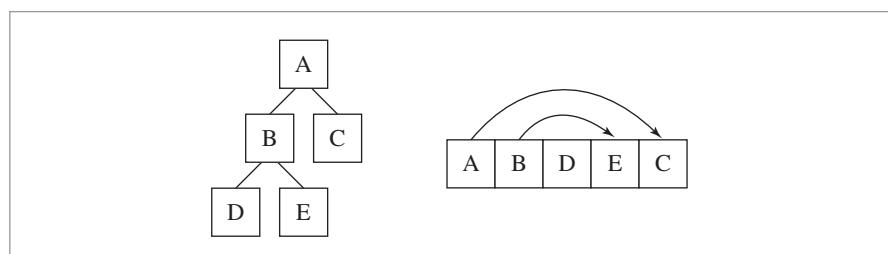
*(BVHAccel Private Methods) ≡*

```
BVHBuildNode *buildUpperSAH(MemoryArena &arena,
                           std::vector<BVHBuildNode *> &treeletRoots, int start, int end,
                           int *totalNodes) const;
```

#### 4.3.4 COMPACT BVH FOR TRAVERSAL

Once the BVH tree is built, the last step is to convert it into a compact representation—doing so improves cache, memory, and thus overall system performance. The final BVH is stored in a linear array in memory. The nodes of the original tree are laid out in depth-first order, which means that the first child of each interior node is immediately after the node in memory. In this case, only the offset to the second child of each interior node must be stored explicitly. See Figure 4.13 for an illustration of the relationship between tree topology and node order in memory.

The `LinearBVHNode` structure stores the information needed to traverse the BVH. In addition to the bounding box for each node, for leaf nodes it stores the offset and



**Figure 4.13: Linear Layout of a BVH in Memory.** The nodes of the BVH (left) are stored in memory in depth-first order (right). Therefore, for any interior node of the tree (A and B in this example), the first child is found immediately after the parent node in memory. The second child is found via an offset pointer, represented here with lines with arrows. Leaf nodes of the tree (D, E, and C) have no children.

`BVHBuildNode` 258  
`LBVHTreelet` 275  
`LBVHTreelet::buildNodes` 275  
`LinearBVHNode` 281  
`MemoryArena` 1074

primitive count for the primitives in the node. For interior nodes, it stores the offset to the second child as well as which of the coordinate axes the primitives were partitioned along when the hierarchy was built; this information is used in the traversal routine below to try to visit nodes in front-to-back order along the ray.

```
(BVHAccel Local Declarations) +≡
struct LinearBVHNode {
    Bounds3f bounds;
    union {
        int primitivesOffset; // leaf
        int secondChildOffset; // interior
    };
    uint16_t nPrimitives; // 0 -> interior node
    uint8_t axis; // interior node: xyz
    uint8_t pad[1]; // ensure 32 byte total size
};
```

This structure is padded to ensure that it's 32 bytes large. Doing so ensures that, if the nodes are allocated such that the first node is cache-line aligned, then none of the subsequent nodes will straddle cache lines (as long as the cache line size is at least 32 bytes, which is the case on modern CPU architectures).

The built tree is transformed to the `LinearBVHNode` representation by the `flattenBVHTree()` method, which performs a depth-first traversal and stores the nodes in memory in linear order.

```
(Compute representation of depth-first traversal of BVH tree) ≡
nodes = AllocAligned<LinearBVHNode>(totalNodes);
int offset = 0;
flattenBVHTree(root, &offset);
```

257

The pointer to the array of `LinearBVHNodes` is stored as a `BVHAccel` member variable so that it can be freed in the `BVHAccel` destructor.

```
(BVHAccel Private Data) +≡
LinearBVHNode *nodes = nullptr;
```

```
Bounds3f 76
BVHAccel 256
BVHAccel::flattenBVHTree()
  282
BVHAccel::nodes 281
LinearBVHNode 281
```

Flattening the tree to the linear representation is straightforward; the `*offset` parameter tracks the current offset into the `BVHAccel::nodes` array. Note that the current node is added to the array before the recursive calls to process its children (if the node is an interior node).

```
(BVHAccel Method Definitions) +≡
int BVHAccel::flattenBVHTree(BVHBuildNode *node, int *offset) {
    LinearBVHNode *linearNode = &nodes[*offset];
    linearNode->bounds = node->bounds;
    int myOffset = (*offset)++;
    if (node->nPrimitives > 0) {
        linearNode->primitivesOffset = node->firstPrimOffset;
        linearNode->nPrimitives = node->nPrimitives;
    } else {
        ⟨Create interior flattened BVH node 282⟩
    }
    return myOffset;
}
```

At interior nodes, recursive calls are made to flatten the two subtrees. The first one ends up immediately after the current node in the array, as desired, and the offset of the second one, returned by its recursive flattenBVHTree() call, is stored in this node's secondChildOffset member.

```
(Create interior flattened BVH node) ≡ 282
linearNode->axis = node->splitAxis;
linearNode->nPrimitives = 0;
flattenBVHTree(node->children[0], offset);
linearNode->secondChildOffset =
    flattenBVHTree(node->children[1], offset);
```

### 4.3.5 TRAVERSAL

The BVH traversal code is quite simple—there are no recursive function calls and only a tiny amount of data to maintain about the current state of the traversal. The Intersect() method starts by precomputing a few values related to the ray that will be used repeatedly.

```
(BVHAccel Method Definitions) +≡
bool BVHAccel::Intersect(const Ray &ray,
    SurfaceInteraction *isect) const {
    bool hit = false;
    Vector3f invDir(1 / ray.d.x, 1 / ray.d.y, 1 / ray.d.z);
    int dirIsNeg[3] = { invDir.x < 0, invDir.y < 0, invDir.z < 0 };
    ⟨Follow ray through BVH nodes to find primitive intersections 283⟩
    return hit;
}
```

Each time the while loop in Intersect() starts an iteration, currentNodeIndex holds the offset into the nodes array of the node to be visited. It starts with a value of 0, representing the root of the tree. The nodes that still need to be visited are stored in the nodesToVisit[] array, which acts as a stack; toVisitOffset holds the offset to the next free element in the stack.

BVHAccel 256  
BVHAccel::flattenBVHTree() 282  
BVHAccel::nodes 281  
BVHBuildNode 258  
BVHBuildNode::bounds 258  
BVHBuildNode::children 258  
BVHBuildNode::firstPrimOffset 258  
BVHBuildNode::nPrimitives 258  
BVHBuildNode::splitAxis 258  
LinearBVHNode 281  
LinearBVHNode::axis 281  
LinearBVHNode::bounds 281  
LinearBVHNode::nPrimitives 281  
LinearBVHNode::primitivesOffset 281  
LinearBVHNode::secondChildOffset 281  
Ray 73  
SurfaceInteraction 116  
Vector3f 60

*(Follow ray through BVH nodes to find primitive intersections) ≡*

```

int toVisitOffset = 0, currentNodeIndex = 0;
int nodesToVisit[64];
while (true) {
    const LinearBVHNode *node = &nodes[currentNodeIndex];
    (Check ray against BVH node 283)
}
```

282

At each node, we check to see if the ray intersects the node's bounding box (or starts inside of it). We visit the node if so, testing for intersection with its primitives if it's a leaf node or processing its children if it's an interior node. If no intersection is found, then the offset of the next node to be visited is retrieved from `nodesToVisit[]` (or traversal is complete if the stack is empty).

*(Check ray against BVH node) ≡*

```

if (node->bounds.IntersectP(ray, invDir, dirIsNeg)) {
    if (node->nPrimitives > 0) {
        (Intersect ray with primitives in leaf BVH node 283)
    } else {
        (Put far BVH node on nodesToVisit stack, advance to near node 284)
    }
} else {
    if (toVisitOffset == 0) break;
    currentNodeIndex = nodesToVisit[--toVisitOffset];
}
```

283

284

If the current node is a leaf, then the ray must be tested for intersection with the primitives inside it. The next node to visit is then found from the `nodesToVisit` stack; even if an intersection is found in the current node, the remaining nodes must be visited, in case one of them yields a closer intersection. However, if an intersection is found, the ray's `tMax` value will be updated to the intersection distance; this makes it possible to efficiently discard any remaining nodes that are farther away than the intersection.

*(Intersect ray with primitives in leaf BVH node) ≡*

```

for (int i = 0; i < node->nPrimitives; ++i)
    if (primitives[node->primitivesOffset + i]->Intersect(ray, isect))
        hit = true;
    if (toVisitOffset == 0) break;
    currentNodeIndex = nodesToVisit[--toVisitOffset];
```

283

284

For an interior node that the ray hits, it is necessary to visit both of its children. As described above, it's desirable to visit the first child that the ray passes through before visiting the second one, in case there is a primitive that the ray intersects in the first one, so that the ray's `tMax` value can be updated, thus reducing the ray's extent and thus the number of node bounding boxes it intersects.

An efficient way to perform a front-to-back traversal without incurring the expense of intersecting the ray with both child nodes and comparing the distances is to use the sign of the ray's direction vector for the coordinate axis along which primitives were

Bounds3::IntersectP() 127

BVHAccel::primitives 257

LinearBVHNode 281

LinearBVHNode::nPrimitives 281

Primitive::Intersect() 249

partitioned for the current node: if the sign is negative, we should visit the second child before the first child, since the primitives that went into the second child's subtree were on the upper side of the partition point. (And conversely for a positive-signed direction.) Doing this is straightforward: the offset for the node to be visited first is copied to `currentNodeIndex`, and the offset for the other node is added to the `nodesToVisit` stack. (Recall that the first child is immediately after the current node due to the depth-first layout of nodes in memory.)

*(Put far BVH node on nodesToVisit stack, advance to near node) ≡*

283

```
if (dirIsNeg[node->axis]) {
    nodesToVisit[toVisitOffset++] = currentNodeIndex + 1;
    currentNodeIndex = node->secondChildOffset;
} else {
    nodesToVisit[toVisitOffset++] = node->secondChildOffset;
    currentNodeIndex = currentNodeIndex + 1;
}
```

The `BVHAccel::IntersectP()` method is essentially the same as the regular intersection method, with the two differences that `Primitive`'s `IntersectP()` methods are called rather than `Intersect()`, and traversal stops immediately when any intersection is found.

## 4.4 KD-TREE ACCELERATOR

*Binary space partitioning* (BSP) trees adaptively subdivide space with planes. A BSP tree starts with a bounding box that encompasses the entire scene. If the number of primitives in the box is greater than some threshold, the box is split in half by a plane. Primitives are then associated with whichever half they overlap, and primitives that lie in both halves are associated with both of them. (This is in contrast to BVHs, where each primitive is assigned to only one of the two subgroups after a split.)

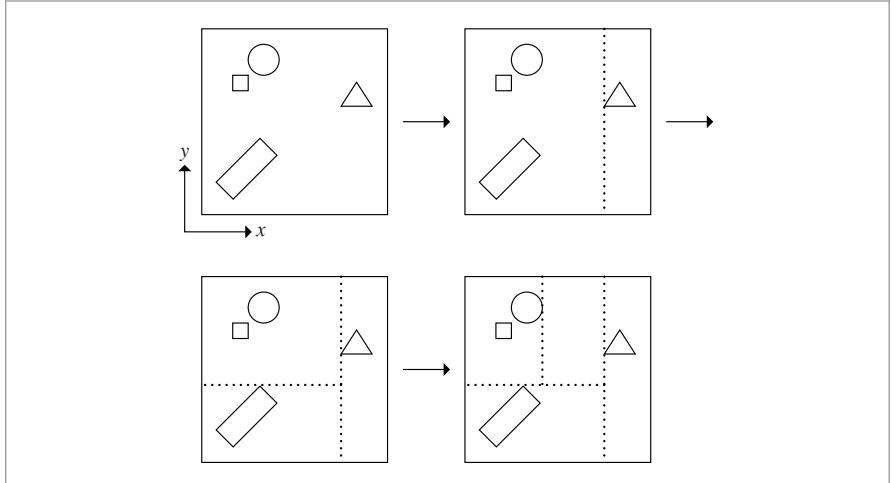
The splitting process continues recursively either until each leaf region in the resulting tree contains a sufficiently small number of primitives or until a maximum depth is reached. Because the splitting planes can be placed at arbitrary positions inside the overall bound and because different parts of 3D space can be refined to different degrees, BSP trees can easily handle uneven distributions of geometry.

Two variations of BSP trees are *kd-trees* and *octrees*. A kd-tree simply restricts the splitting plane to be perpendicular to one of the coordinate axes; this makes both traversal and construction of the tree more efficient, at the cost of some flexibility in how space is subdivided. The octree uses three axis-perpendicular planes to simultaneously split the box into eight regions at each step (typically by splitting down the center of the extent in each direction). In this section, we will implement a kd-tree for ray intersection acceleration in the `KdTreeAccel` class. Source code for this class can be found in the files `accelerators/kdtreeaccel.h` and `accelerators/kdtreeaccel.cpp`.

`KdTreeAccel` 285

`LinearBVHNode::`  
`secondChildOffset`  
281

`Primitive` 248



**Figure 4.14:** The kd-tree is built by recursively splitting the bounding box of the scene geometry along one of the coordinate axes. Here, the first split is along the  $x$  axis; it is placed so that the triangle is precisely alone in the right region and the rest of the primitives end up on the left. The left region is then refined a few more times with axis-aligned splitting planes. The details of the refinement criteria—which axis is used to split space at each step, at which position along the axis the plane is placed, and at what point refinement terminates—can all substantially affect the performance of the tree in practice.

```
(KdTreeAccel Declarations) ≡
class KdTreeAccel : public Aggregate {
public:
    (KdTreeAccel Public Methods 302)
private:
    (KdTreeAccel Private Methods)
    (KdTreeAccel Private Data 286)
};
```

In addition to the primitives to be stored, the `KdTreeAccel` constructor takes a few parameters that are used to guide the decisions that will be made as the tree is built; these parameters are stored in member variables (`isectCost`, `traversalCost`, `maxPrims`, `maxDepth`, and `emptyBonus`) for later use. See Figure 4.14 for an overview of how the tree is built.

```
(KdTreeAccel Method Definitions) ≡
KdTreeAccel::KdTreeAccel(
    const std::vector<std::shared_ptr<Primitive>> &p,
    int isectCost, int traversalCost, Float emptyBonus,
    int maxPrims, int maxDepth)
: isectCost(isectCost), traversalCost(traversalCost),
  maxPrims(maxPrims), emptyBonus(emptyBonus), primitives(p) {
    (Build kd-tree for accelerator 289)
}
```

Aggregate 255  
 Float 1062  
 KdTreeAccel 285  
 Primitive 248

<i>(KdTreeAccel Private Data) ≡</i> <pre>const int isectCost, traversalCost, maxPrims; const Float emptyBonus; std::vector&lt;std::shared_ptr&lt;Primitive&gt;&gt; primitives;</pre>	285
---	-----

#### 4.4.1 TREE REPRESENTATION

The kd-tree is a binary tree, where each interior node always has both children and where leaves of the tree store the primitives that overlap them. Each interior node must provide access to three pieces of information:

- Split axis: which of the  $x$ ,  $y$ , or  $z$  axes was split at this node
- Split position: the position of the splitting plane along the axis
- Children: information about how to reach the two child nodes beneath it

Each leaf node needs to record only which primitives overlap it.

It is worth going through a bit of trouble to ensure that all interior nodes and many leaf nodes use just 8 bytes of memory (assuming 4-byte `Floats`) because doing so ensures that eight nodes will fit into a 64-byte cache line. Because there are often many nodes in the tree and because many nodes are generally accessed for each ray, minimizing the size of the node representation substantially improves cache performance. Our initial implementation used a 16-byte node representation; when we reduced the size to 8 bytes we obtained nearly a 20% speed increase.

Both leaves and interior nodes are represented by the following `KdAccelNode` structure. The comments after each union member indicate whether a particular field is used for interior nodes, leaf nodes, or both.

<i>(KdTreeAccel Local Declarations) ≡</i> <pre>struct KdAccelNode {     <i>(KdAccelNode Methods 288)</i>     union {         Float split;           // Interior         int onePrimitive;      // Leaf         int primitiveIndicesOffset; // Leaf     };     union {         int flags;            // Both         int nPrims;           // Leaf         int aboveChild;        // Interior     }; };</pre>	<a href="#">Float 1062</a> <a href="#">KdAccelNode 286</a> <a href="#">KdAccelNode::flags 286</a> <a href="#">KdAccelNode::nPrims 286</a> <a href="#">Primitive 248</a>
---	---

The two low-order bits of the `KdAccelNode::flags` variable are used to differentiate between interior nodes with  $x$ ,  $y$ , and  $z$  splits (where these bits hold the values 0, 1, and 2, respectively) and leaf nodes (where these bits hold the value 3). It is relatively easy to store leaf nodes in 8 bytes: since the low 2 bits of `KdAccelNode::flags` are used to indicate that this is a leaf, the upper 30 bits of `KdAccelNode::nPrims` are available to record how many primitives overlap it. Then, if just a single primitive overlaps a

KdAccelNode leaf, an integer index into the KdTreeAccel::primitives array identifies the Primitive. If more than one primitive overlaps, then their indices are stored in a segment of KdTreeAccel::primitiveIndices. The offset to the first index for the leaf is stored in KdAccelNode::primitiveIndicesOffset and the indices for the rest directly follow.

*(KdTreeAccel Private Data) +≡*  
*std::vector<int> primitiveIndices;*

285

Leaf nodes are easy to initialize, though we have to be careful with the details since both flags and nPrims share the same storage; we need to be careful to not clobber data for one of them while initializing the other. Furthermore, the number of primitives must be shifted two bits to the left before being stored so that the low two bits of KdAccelNode::flags can both be set to 1 to indicate that this is a leaf node.

*(KdTreeAccel Method Definitions) +≡*  
*void KdAccelNode::InitLeaf(int \*primNums, int np,*  
*std::vector<int> \*primitiveIndices) {*  
*flags = 3;*  
*nPrims |= (np << 2);*  
**(Store primitive ids for leaf node 287)**  
*}*

287

For leaf nodes with zero or one overlapping primitives, no additional memory allocation is necessary thanks to the KdAccelNode::onePrimitive field. For the case where multiple primitives overlap, storage is allocated in the primitiveIndices array.

*(Store primitive ids for leaf node) ≡*  
*if (np == 0)*  
*onePrimitive = 0;*  
*else if (np == 1)*  
*onePrimitive = primNums[0];*  
*else {*  
*primitiveIndicesOffset = primitiveIndices->size();*  
*for (int i = 0; i < np; ++i)*  
*primitiveIndices->push\_back(primNums[i]);*  
*}*

287

KdAccelNode 286  
KdAccelNode::flags 286  
KdAccelNode::nPrims 286  
KdAccelNode::onePrimitive 286  
KdAccelNode::primitiveIndicesOffset 286  
KdTreeAccel::primitiveIndices 287  
KdTreeAccel::primitives 286  
Primitive 248

Getting interior nodes down to 8 bytes is also reasonably straightforward. A Float (which is 32 bits in size when Floats are defined to be floats) stores the position along the chosen split axis where the node splits space, and, as explained earlier, the lowest 2 bits of KdAccelNode::flags are used to record which axis the node was split along. All that is left is to store enough information to find the two children of the node as we're traversing the tree.

Rather than storing two pointers or offsets, we lay the nodes out in a way that lets us only store one child pointer: all of the nodes are allocated in a single contiguous block of memory, and the child of an interior node that is responsible for space below the splitting plane is always stored in the array position immediately after its parent (this layout also improves cache performance, by keeping at least one child close to its parent in memory). The other child, representing space above the splitting plane, will end up somewhere else

in the array; a single integer offset, `KdAccelNode::aboveChild`, stores its position in the nodes array. This representation is similar to the one used for BVH nodes in Section 4.4.3.

Given all those conventions, the code to initialize an interior node is straightforward. As in the `InitLeaf()` method, it's important to assign the value to `flags` before setting `aboveChild` and to compute the logical OR of the shifted above child value so as not to clobber the bits stored in `flags`.

*(KdAccelNode Methods) ≡*

286

```
void InitInterior(int axis, int ac, Float s) {
    split = s;
    flags = axis;
    aboveChild |= (ac << 2);
}
```

Finally, we'll provide a few methods to extract various values from the node, so that callers don't have to be aware of the details of its representation in memory.

*(KdAccelNode Methods) +≡*

286

```
Float SplitPos() const { return split; }
int nPrimitives() const { return nPrims >> 2; }
int SplitAxis() const { return flags & 3; }
bool IsLeaf() const { return (flags & 3) == 3; }
int AboveChild() const { return aboveChild >> 2; }
```

#### 4.4.2 TREE CONSTRUCTION

The kd-tree is built with a recursive top-down algorithm. At each step, we have an axis-aligned region of space and a set of primitives that overlap that region. Either the region is split into two subregions and turned into an interior node or a leaf node is created with the overlapping primitives, terminating the recursion.

As mentioned in the discussion of `KdAccelNodes`, all tree nodes are stored in a contiguous array. `KdTreeAccel::nextFreeNode` records the next node in this array that is available, and `KdTreeAccel::nAllocatedNodes` records the total number that have been allocated. By setting both of them to 0 and not allocating any nodes at start-up, the implementation here ensures that an allocation will be done immediately when the first node of the tree is initialized.

It is also necessary to determine a maximum tree depth if one wasn't given to the constructor. Although the tree construction process will normally terminate naturally at a reasonable depth, it is important to cap the maximum depth so that the amount of memory used for the tree cannot grow without bound in pathological cases. We have found that the value  $8 + 1.3 \log(N)$  gives a reasonable maximum depth for a variety of scenes.

Float 1062  
 KdAccelNode 286  
`KdAccelNode::aboveChild` 286  
`KdAccelNode::flags` 286  
`KdAccelNode::nPrims` 286  
`KdAccelNode::split` 286  
`KdTreeAccel::nAllocatedNodes` 289  
`KdTreeAccel::nextFreeNode` 289

```
<Build kd-tree for accelerator> ≡ 285
    nextFreeNode = nAllocedNodes = 0;
    if (maxDepth <= 0)
        maxDepth = std::round(8 + 1.3f * Log2Int(primitives.size()));
(Compute bounds for kd-tree construction 289)
(Allocate working memory for kd-tree construction 292)
(Initialize primNums for kd-tree construction 289)
(Start recursive construction of kd-tree 289)
```

```
(KdTreeAccel Private Data) +≡ 285
    KdAccelNode *nodes;
    int nAllocedNodes, nextFreeNode;
```

Because the construction routine will be repeatedly using the bounding boxes of the primitives along the way, they are stored in a vector before tree construction starts so that the potentially slow `Primitive::WorldBound()` methods don't need to be called repeatedly.

```
(Compute bounds for kd-tree construction) ≡ 289
    std::vector<Bounds3f> primBounds;
    for (const std::shared_ptr<Primitive> &prim : primitives) {
        Bounds3f b = prim->WorldBound();
        bounds = Union(bounds, b);
        primBounds.push_back(b);
    }
```

```
(KdTreeAccel Private Data) +≡ 285
    Bounds3f bounds;
```

One of the parameters to the tree construction routine is an array of primitive indices indicating which primitives overlap the current node. Because all primitives overlap the root node (when the recursion begins) we start with an array initialized with values from zero through `primitives.size()-1`.

```
(Initialize primNums for kd-tree construction) ≡ 289
    std::unique_ptr<int[]> primNums(new int[primitives.size()]);
    for (size_t i = 0; i < primitives.size(); ++i)
        primNums[i] = i;
```

`Bounds3::Union()` 78  
`Bounds3f` 76  
`KdAccelNode` 286  
`KdTreeAccel::buildTree()` 290  
`KdTreeAccel::primitives` 286  
`Log2Int()` 1064  
`Primitive` 248  
`Primitive::WorldBound()` 249

`KdTreeAccel::buildTree()` is called for each tree node. It is responsible for deciding if the node should be an interior node or leaf and updating the data structures appropriately. The last three parameters, `edges`, `prims0`, and `prims1`, are pointers to data that is allocated in the *(Allocate working memory for kd-tree construction)* fragment, which will be defined and documented in a few pages.

```
(Start recursive construction of kd-tree) ≡ 289
    buildTree(0, bounds, primBounds, primNums.get(), primitives.size(),
              maxDepth, edges, prims0.get(), prims1.get());
```

The main parameters to `KdTreeAccel::buildTree()` are the offset into the array of `KdAccelNodes` to use for the node that it creates, `nodeNum`; the bounding box that gives

the region of space that the node covers, `nodeBounds`; and the indices of primitives that overlap it, `primNums`. The remainder of the parameters will be described later, closer to where they are used.

```
(KdTreeAccel Method Definitions) +≡
void KdTreeAccel::buildTree(int nodeNum, const Bounds3f &nodeBounds,
    const std::vector<Bounds3f> &allPrimBounds, int *primNums,
    int nPrimitives, int depth,
    const std::unique_ptr<BoundEdge[]> edges[3],
    int *prims0, int *prims1, int badRefines) {
    (Get next free node from nodes array 290)
    (Initialize leaf node if termination criteria met 290)
    (Initialize interior node and continue recursion 290)
}
```

If all of the allocated nodes have been used up, node memory is reallocated with twice as many entries and the old values are copied. The first time `KdTreeAccel::buildTree()` is called, `KdTreeAccel::nAllocatedNodes` is 0 and an initial block of tree nodes is allocated.

```
(Get next free node from nodes array) ≡ 290
if (nextFreeNode == nAllocatedNodes) {
    int nNewAllocNodes = std::max(2 * nAllocatedNodes, 512);
    KdAccelNode *n = AllocAligned<KdAccelNode>(nNewAllocNodes);
    if (nAllocatedNodes > 0) {
        memcpy(n, nodes, nAllocatedNodes * sizeof(KdAccelNode));
        FreeAligned(nodes);
    }
    nodes = n;
    nAllocatedNodes = nNewAllocNodes;
}
++nextFreeNode;
```

A leaf node is created (stopping the recursion) either if there are a sufficiently small number of primitives in the region or if the maximum depth has been reached. The `depth` parameter starts out as the tree's maximum depth and is decremented at each level.

```
(Initialize leaf node if termination criteria met) ≡ 290
if (nPrimitives <= maxPrims || depth == 0) {
    nodes[nodeNum].InitLeaf(primNums, nPrimitives, &primitiveIndices);
    return;
}
```

If this is an internal node, it is necessary to choose a splitting plane, classify the primitives with respect to that plane, and recurse.

```
(Initialize interior node and continue recursion) ≡ 290
(Choose split axis position for interior node 293)
(Create leaf if no good splits were found 296)
(Classify primitives with respect to split 296)
(Recursively initialize children nodes 297)
```

`AllocAligned()` 1072  
`BoundEdge` 292  
`Bounds3f` 76  
`KdAccelNode` 286  
`KdAccelNode::InitLeaf()` 287  
`KdTreeAccel` 285  
`KdTreeAccel::buildTree()` 290  
`KdTreeAccel::maxPrims` 286  
`KdTreeAccel::nAllocatedNodes` 289  
`KdTreeAccel::nextFreeNode` 289  
`KdTreeAccel::nodes` 289  
`KdTreeAccel::primitiveIndices` 287

Our implementation chooses a split using the SAH introduced in Section 4.3.2. The SAH is applicable to kd-trees as well as BVHs; here, the estimated cost is computed for a series of candidate splitting planes in the node, and the split that gives the lowest cost is chosen.

In the implementation here, the intersection cost  $t_{\text{isect}}$  and the traversal cost  $t_{\text{trav}}$  can be set by the user; their default values are 80 and 1, respectively. Ultimately, it is the ratio of these two values that determines the behavior of the tree-building algorithm.<sup>4</sup> The greater ratio between these values compared to the values used for BVH construction reflects the fact that visiting a kd-tree node is less expensive than visiting a BVH node.

One modification to the SAH used for BVH trees is that for kd-trees it is worth giving a slight preference to choosing splits where one of the children has no primitives overlapping it, since rays passing through these regions can immediately advance to the next kd-tree node without any ray-primitive intersection tests. Thus, the revised costs for unsplit and split regions are, respectively,

$$t_{\text{isect}}N \quad \text{and} \quad t_{\text{trav}} + (1 - b_e)(p_B N_B t_{\text{isect}} + p_A N_A t_{\text{isect}}),$$

where  $b_e$  is a “bonus” value that is zero unless one of the two regions is completely empty, in which case it takes on a value between 0 and 1.

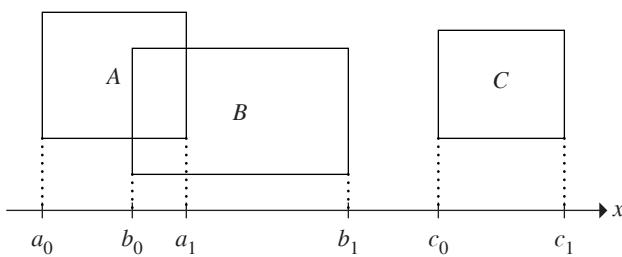
Given a way to compute the probabilities for the cost model, the only problem to address is how to generate candidate splitting positions and how to efficiently compute the cost for each candidate. It can be shown that the minimum cost with this model will be attained at a split that is coincident with one of the faces of one of the primitive’s bounding boxes—there’s no need to consider splits at intermediate positions. (To convince yourself of this, consider the behavior of the cost function between the edges of the faces.) Here, we will consider all bounding box faces inside the region for one or more of the three coordinate axes.

The cost for checking all of these candidates thus can be kept relatively low with a carefully structured algorithm. To compute these costs, we will sweep across the projections of the bounding boxes onto each axis and keep track of which gives the lowest cost (Figure 4.15). Each bounding box has two edges on each axis, each of which is represented by an instance of the `BoundEdge` structure. This structure records the position of the edge along the axis, whether it represents the start or end of a bounding box (going from low to high along the axis), and which primitive it is associated with.

```
(KdTreeAccel Local Declarations) +=  
enum class EdgeType { Start, End };
```

---

<sup>4</sup> Many other implementations of this approach seem to use values for these costs that are much closer together, sometimes even approaching equal values (for example, see Hurley et al. 2002). The values used here gave the best performance for a number of test scenes in pbrt. We suspect that this discrepancy is due to the fact that ray-primitive intersection tests in pbrt require two virtual function calls and a ray world-to-object-space transformation, in addition to the cost of performing the actual intersection test. Highly optimized ray tracers that only support triangle primitives don’t pay any of that additional cost. See Section 17.1.1 for further discussion of this design trade-off.



**Figure 4.15:** Given an axis along which we'd like to consider possible splits, the primitives' bounding boxes are projected onto the axis, which leads to an efficient algorithm to track how many primitives would be on each side of a particular splitting plane. Here, for example, a split at  $a_1$  would leave  $A$  completely below the splitting plane,  $B$  straddling it, and  $C$  completely above it. Each point on the axis,  $a_0, a_1, b_0, b_1, c_0$ , and  $c_1$ , is represented by an instance of the `BoundEdge` structure.

*(KdTreeAccel Local Declarations)* +≡

```
struct BoundEdge {
    (BoundEdge Public Methods 292)
    float t;
    int primNum;
    EdgeType type;
};
```

*(BoundEdge Public Methods)* ≡

292

```
BoundEdge(float t, int primNum, bool starting)
: t(t), primNum(primNum) {
    type = starting ? EdgeType::Start : EdgeType::End;
}
```

At most,  $2 * \text{primitives.size()}$  `BoundEdges` are needed for computing costs for any tree node, so the memory for the edges for all three axes is allocated once and then reused for each node that is created.

*(Allocate working memory for kd-tree construction)* ≡

289

```
std::unique_ptr<BoundEdge[]> edges[3];
for (int i = 0; i < 3; ++i)
    edges[i].reset(new BoundEdge[2 * primitives.size()]);
```

After determining the estimated cost for creating a leaf, `KdTreeAccel::buildTree()` chooses an axis to try to split along and computes the cost function for each candidate split. `bestAxis` and `bestOffset` record the axis and bounding box edge index that have given the lowest cost so far, `bestCost`. `invTotalSA` is initialized to the reciprocal of the node's surface area; its value will be used when computing the probabilities of rays passing through each of the candidate children nodes.

BoundEdge 292  
`BoundEdge::primNum` 292  
`BoundEdge::t` 292  
`BoundEdge::type` 292  
`EdgeType` 291  
`EdgeType::End` 291  
`EdgeType::Start` 291  
`Float` 1062  
`KdTreeAccel::buildTree()` 290

```

<Choose split axis position for interior node> ≡ 290
    int bestAxis = -1, bestOffset = -1;
    Float bestCost = Infinity;
    Float oldCost = isectCost * Float(nPrimitives);
    Float totalSA = nodeBounds.SurfaceArea();
    Float invTotalSA = 1 / totalSA;
    Vector3f d = nodeBounds.pMax - nodeBounds.pMin;
    <Choose which axis to split along 293>
    int retries = 0;
    retrySplit:
    <Initialize edges for axis 293>
    <Compute cost of all splits for axis to find best 294>

```

This method first tries to find a split along the axis with the largest spatial extent; if successful, this choice helps to give regions of space that tend toward being square in shape. This is an intuitively sensible approach. Later, if it was unsuccessful in finding a good split along this axis, it will go back and try the others in turn.

```

<Choose which axis to split along> ≡ 293
    int axis = nodeBounds.MaximumExtent();

```

First the edges array for the axis is initialized using the bounding boxes of the overlapping primitives. The array is then sorted from low to high along the axis so that it can sweep over the box edges from first to last.

```

<Initialize edges for axis> ≡ 293
    for (int i = 0; i < nPrimitives; ++i) {
        int pn = primNums[i];
        const Bounds3f &bounds = allPrimBounds[pn];
        edges[axis][2 * i] = BoundEdge(bounds.pMin[axis], pn, true);
        edges[axis][2 * i + 1] = BoundEdge(bounds.pMax[axis], pn, false);
    }
    <Sort edges for axis 293>

```

The C++ standard library routine `sort()` requires that the structure being sorted define an ordering; this is done using the `BoundEdge::t` values. However, one subtlety is that if the `BoundEdge::t` values match, it is necessary to try to break the tie by comparing the node's types; this is necessary since `sort()` depends on the fact that the only time `a < b` and `b < a` are both `false` is when `a == b`.

```

BoundEdge 292
BoundEdge::t 292
BoundEdge::type 292
Bounds3::MaximumExtent() 80
Bounds3::SurfaceArea() 80
Bounds3f 76
Float 1062
Infinity 210
KdTreeAccel::isectCost 286
Vector3f 60

```

```

<Sort edges for axis> ≡ 293
    std::sort(&edges[axis][0], &edges[axis][2*nPrimitives],
              [] (const BoundEdge &e0, const BoundEdge &e1) -> bool {
        if (e0.t == e1.t)
            return (int)e0.type < (int)e1.type;
        else return e0.t < e1.t;
    });

```

Given the sorted array of edges, we'd like to quickly compute the cost function for a split at each one of them. The probabilities for a ray passing through each child node are easily

computed using their surface areas, and the number of primitives on each side of the split is tracked by the variables `nBelow` and `nAbove`. We would like to keep their values updated such that if we chose to split at `edgeT` for a particular pass through the loop, `nBelow` will give the number of primitives that would end up below the splitting plane and `nAbove` would give the number above it.<sup>5</sup>

At the first edge, all primitives must be above that edge by definition, so `nAbove` is initialized to `nPrimitives` and `nBelow` is set to 0. When the loop is considering a split at the end of a bounding box's extent, `nAbove` needs to be decremented, since that box, which must have previously been above the splitting plane, will no longer be above it if splitting is done at the point. Similarly, after calculating the split cost, if the split candidate was at the start of a bounding box's extent, then the box will be on the below side for all subsequent splits. The tests at the start and end of the loop body update the primitive counts for these two cases.

*(Compute cost of all splits for axis to find best) ≡*

293

```
int nBelow = 0, nAbove = nPrimitives;
for (int i = 0; i < 2 * nPrimitives; ++i) {
    if (edges[axis][i].type == EdgeType::End) --nAbove;
    float edgeT = edges[axis][i].t;
    if (edgeT > nodeBounds.pMin[axis] &&
        edgeT < nodeBounds.pMax[axis]) {
        (Compute cost for split at ith edge 295)
    }
    if (edges[axis][i].type == EdgeType::Start) ++nBelow;
}
```

`belowSA` and `aboveSA` hold the surface areas of the two candidate child bounds; they are easily computed by adding up the areas of the six faces.

*(Compute child surface areas for split at edget) ≡*

295

```
int otherAxis0 = (axis + 1) % 3, otherAxis1 = (axis + 2) % 3;
float belowSA = 2 * (d[otherAxis0] * d[otherAxis1] +
                     (edget - nodeBounds.pMin[axis]) *
                     (d[otherAxis0] + d[otherAxis1]));
float aboveSA = 2 * (d[otherAxis0] * d[otherAxis1] +
                     (nodeBounds.pMax[axis] - edget) *
                     (d[otherAxis0] + d[otherAxis1]));
```

Given all of this information, the cost for a particular split can be computed.

---

<sup>5</sup> When multiple bounding box faces project to the same point on the axis, this invariant may not be true at those points. However, as implemented here it will only overestimate the counts and, more importantly, will have the correct value for one of the multiple times through the loop at each of those points, so the algorithm functions correctly in the end anyway.

BoundEdge::t 292  
 BoundEdge::type 292  
 Bounds3::pMax 77  
 Bounds3::pMin 77  
 EdgeType::End 291  
 EdgeType::Start 291  
 float 1062

```

⟨Compute cost for split at ith edge⟩ ≡ 294
    ⟨Compute child surface areas for split at edgeT 294⟩
    Float pBelow = belowSA * invTotalSA;
    Float pAbove = aboveSA * invTotalSA;
    Float eb = (nAbove == 0 || nBelow == 0) ? emptyBonus : 0;
    Float cost = traversalCost +
        isectCost * (1 - eb) * (pBelow * nBelow + pAbove * nAbove);
    ⟨Update best split if this is lowest cost so far 295⟩

```

If the cost computed for this candidate split is the best one so far, the details of the split are recorded.

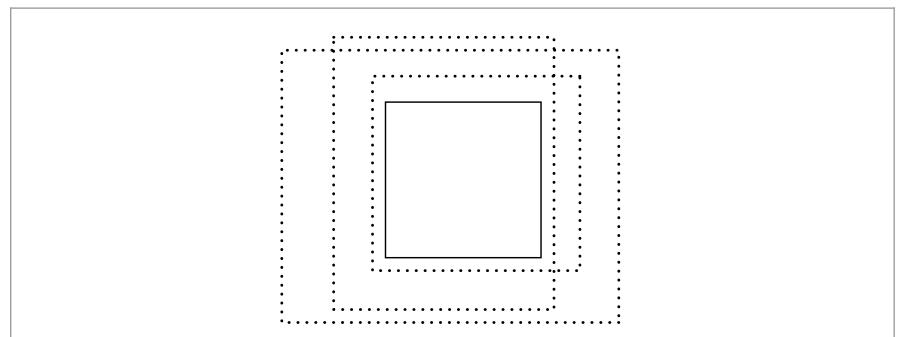
```

⟨Update best split if this is lowest cost so far⟩ ≡ 295
    if (cost < bestCost) {
        bestCost = cost;
        bestAxis = axis;
        bestOffset = i;
    }

```

It may happen that there are no possible splits found in the previous tests (Figure 4.16 illustrates a case where this may happen). In this case, there isn't a single candidate position at which to split the node along the current axis. At this point, splitting is tried for the other two axes in turn. If neither of them can find a split (when retries is equal to 2), then there is no useful way to refine the node, since both children will still have the same number of overlapping primitives. When this condition occurs, all that can be done is to give up and make a leaf node.

It is also possible that the best split will have a cost that is still higher than the cost for not splitting the node at all. If it is substantially worse and there aren't too many primitives, a leaf node is made immediately. Otherwise, badRefines keeps track of how many bad



**Figure 4.16:** If multiple bounding boxes (dotted lines) overlap a kd-tree node (solid lines) as shown here, there is no possible split position that can result in fewer than all of the primitives being on both sides of it.

splits have been made so far above the current node of the tree. It's worth allowing a few slightly poor refinements since later splits may be able to find better ones given a smaller subset of primitives to consider.

*(Create leaf if no good splits were found) ≡* 290

```

if (bestAxis == -1 && retries < 2) {
    ++retries;
    axis = (axis + 1) % 3;
    goto retrySplit;
}
if (bestCost > oldCost) ++badRefines;
if ((bestCost > 4 * oldCost && nPrimitives < 16) ||
    bestAxis == -1 || badRefines == 3) {
    nodes[nodeNum].InitLeaf(primNums, nPrimitives, &primitiveIndices);
    return;
}
```

Having chosen a split position, the bounding box edges can be used to classify the primitives as being above, below, or on both sides of the split in the same way as was done to keep track of `nBelow` and `nAbove` in the earlier code. Note that the `bestOffset` entry in the arrays is skipped in the loops below; this is necessary so that the primitive whose bounding box edge was used for the split isn't incorrectly categorized as being on both sides of the split.

*(Classify primitives with respect to split) ≡* 290

```

int n0 = 0, n1 = 0;
for (int i = 0; i < bestOffset; ++i)
    if (edges[bestAxis][i].type == EdgeType::Start)
        prims0[n0++] = edges[bestAxis][i].primNum;
for (int i = bestOffset + 1; i < 2 * nPrimitives; ++i)
    if (edges[bestAxis][i].type == EdgeType::End)
        prims1[n1++] = edges[bestAxis][i].primNum;
```

Recall that the node number of the “below” child of this node in the kd-tree `nodes` array is the current node number plus one. After the recursion has returned from that side of the tree, the `nextFreeNode` offset is used for the “above” child. The only other important detail here is that the `prims0` memory is passed directly for reuse by both children, while the `prims1` pointer is advanced forward first. This is necessary since the current invocation of `KdTreeAccel::buildTree()` depends on its `prims1` values being preserved over the first recursive call to `KdTreeAccel::buildTree()` in the following, since it must be passed as a parameter to the second call. However, there is no corresponding need to preserve the `edges` values or to preserve `prims0` beyond its immediate use in the first recursive call.

`BoundEdge::primNum` [292](#)  
`BoundEdge::type` [292](#)  
`EdgeType::End` [291](#)  
`EdgeType::Start` [291](#)  
`KdAccelNode::InitLeaf()` [287](#)  
`KdTreeAccel::buildTree()` [290](#)  
`KdTreeAccel::primitiveIndices` [287](#)

```
<Recursively initialize children nodes> ≡ 290
    Float tSplit = edges[bestAxis][bestOffset].t;
    Bounds3f bounds0 = nodeBounds, bounds1 = nodeBounds;
    bounds0.pMax[bestAxis] = bounds1.pMin[bestAxis] = tSplit;
    buildTree(nodeNum + 1, bounds0, allPrimBounds, prims0, n0,
              depth - 1, edges, prims0, prims1 + nPrimitives, badRefines);
    int aboveChild = nextFreeNode;
    nodes[nodeNum].InitInterior(bestAxis, aboveChild, tSplit);
    buildTree(aboveChild, bounds1, allPrimBounds, prims1, n1,
              depth - 1, edges, prims0, prims1 + nPrimitives, badRefines);
```

Thus, much more space is needed for the `prims1` array of integers for storing the worst-case possible number of overlapping primitive numbers than for the `prims0` array, which only needs to handle the primitives at a single level at a time.

```
<Allocate working memory for kd-tree construction> +≡ 289
    std::unique_ptr<int[]> prims0(new int[primitives.size()]);
    std::unique_ptr<int[]> prims1(new int[(maxDepth+1) * primitives.size()]);
```

#### 4.4.3 TRAVERSAL

Figure 4.17 shows the basic process of ray traversal through the tree. Intersecting the ray with the tree’s overall bounds gives initial `tMin` and `tMax` values, marked with points in the figure. As with the `BVHAccel` in this chapter, if the ray misses the overall primitive bounds, this method can immediately return `false`. Otherwise, it starts to descend into the tree, starting at the root. At each interior node, it determines which of the two children the ray enters first and processes both children in order. Traversal ends either when the ray exits the tree or when the closest intersection is found.

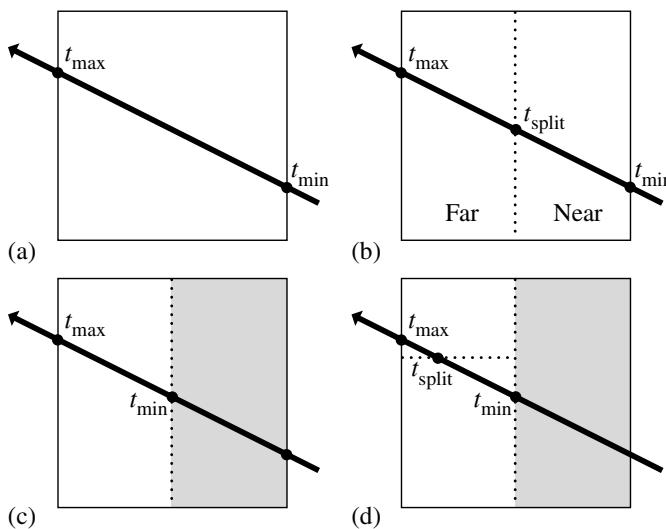
```
<KdTreeAccel Method Definitions> +≡
    bool KdTreeAccel::Intersect(const Ray &ray,
                                SurfaceInteraction *isect) const {
        <Compute initial parametric range of ray inside kd-tree extent 297>
        <Prepare to traverse kd-tree for ray 298>
        <Traverse kd-tree nodes in order for ray 299>
    }
```

BoundEdge::t 292  
 Bounds3::IntersectP() 127  
 Bounds3::pMax 77  
 Bounds3::pMin 76  
 BVHAccel 256  
 Float 1062  
 KdAccelNode::InitInterior() 288  
 KdTreeAccel 285  
 KdTreeAccel::buildTree() 290  
 KdTreeAccel::nextFreeNode 289  
 KdTreeAccel::primitives 286  
 Ray 73  
 SurfaceInteraction 116

The algorithm starts by finding the overall parametric range  $[t_{\min}, t_{\max}]$  of the ray’s overlap with the tree, exiting immediately if there is no overlap.

```
<Compute initial parametric range of ray inside kd-tree extent> ≡ 297
    Float tMin, tMax;
    if (!bounds.IntersectP(ray, &tMin, &tMax))
        return false;
```

The array of `KdToDo` structures is used to record the nodes yet to be processed for the ray; it is ordered so that the last active entry in the array is the next node that should be considered. The maximum number of entries needed in this array is the maximum



**Figure 4.17: Traversal of a Ray through the kd-Tree.** (a) The ray is intersected with the bounds of the tree, giving an initial parametric  $[t_{\min}, t_{\max}]$  range to consider. (b) Because this range is nonempty, it is necessary to consider the two children of the root node here. The ray first enters the child on the right, labeled “near,” where it has a parametric range  $[t_{\min}, t_{\text{split}}]$ . If the near node is a leaf with primitives in it, ray-primitive intersection tests are performed; otherwise, its children nodes are processed. (c) If no hit is found in the node, or if a hit is found beyond  $[t_{\min}, t_{\text{split}}]$ , then the far node, on the left, is processed. (d) This sequence continues—processing tree nodes in a depth-first, front-to-back traversal—until the closest intersection is found or the ray exits the tree.

depth of the kd-tree; the array size used in the following should be more than enough in practice.

*(Prepare to traverse kd-tree for ray) ≡*

297

```
Vector3f invDir(1 / ray.d.x, 1 / ray.d.y, 1 / ray.d.z);
constexpr int maxTodo = 64;
KdToDo todo[maxTodo];
int todoPos = 0;
```

*(KdTreeAccel Declarations) +≡*

```
struct KdToDo {
    const KdAccelNode *node;
    Float tMin, tMax;
};
```

Float 1062  
KdAccelNode 286  
KdToDo 298  
Ray::d 73  
Vector3f 60

The traversal continues through the nodes, processing a single leaf or interior node each time through the loop. The values  $t_{\min}$  and  $t_{\max}$  will always hold the parametric range for the ray’s overlap with the current node.

```

⟨Traverse kd-tree nodes in order for ray⟩ ≡ 297
    bool hit = false;
    const KdAccelNode *node = &nodes[0];
    while (node != nullptr) {
        ⟨Bail out if we found a hit closer than the current node 299⟩
        if (!node->IsLeaf()) {
            ⟨Process kd-tree interior node 299⟩
        } else {
            ⟨Check for intersections inside leaf node 301⟩
            ⟨Grab next node to process from todo list 302⟩
        }
    }
    return hit;

```

An intersection may have been previously found in a primitive that overlaps multiple nodes. If the intersection was outside the current node when first detected, it is necessary to keep traversing the tree until we come to a node where `tMin` is beyond the intersection. Only then is it certain that there is no closer intersection with some other primitive.

```

⟨Bail out if we found a hit closer than the current node⟩ ≡ 299
    if (ray.tMax < tMin) break;

```

For interior tree nodes the first thing to do is to intersect the ray with the node's splitting plane; given the intersection point, we can determine if one or both of the children nodes need to be processed and in what order the ray passes through them.

```

⟨Process kd-tree interior node⟩ ≡ 299
    ⟨Compute parametric distance along ray to split plane 299⟩
    ⟨Get node children pointers for ray 300⟩
    ⟨Advance to next child node, possibly enqueue other child 301⟩

```

The parametric distance to the split plane is computed in the same manner as was done in computing the intersection of a ray and an axis-aligned plane for the ray–bounding box test. We use the precomputed `invDir` value to save a divide each time through the loop.

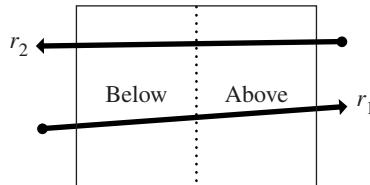
```

⟨Compute parametric distance along ray to split plane⟩ ≡ 299
    int axis = node->SplitAxis();
    float tPlane = (node->SplitPos() - ray.o[axis]) * invDir[axis];

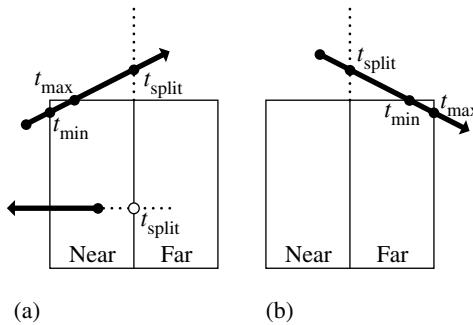
float 1062
KdAccelNode 286
KdAccelNode::IsLeaf() 288
KdAccelNode::SplitAxis() 288
KdAccelNode::SplitPos() 288
KdTreeAccel::nodes 289
Ray::o 73
Ray::tMax 73

```

Now it is necessary to determine the order in which the ray encounters the children nodes so that the tree is traversed in front-to-back order along the ray. Figure 4.18 shows the geometry of this computation. The position of the ray's origin with respect to the splitting plane is enough to distinguish between the two cases, ignoring for now the case where the ray doesn't actually pass through one of the two nodes. The rare case when the ray's origin lies on the splitting plane requires careful handling in this case, as its direction needs to be used instead to discriminate between the two cases.



**Figure 4.18:** The position of the origin of the ray with respect to the splitting plane can be used to determine which of the node's children should be processed first. If the origin of a ray like  $r_1$  is on the “below” side of the splitting plane, we should process the below child before the above child, and vice versa.



**Figure 4.19:** Two cases where both children of a node don't need to be processed because the ray doesn't overlap them. (a) The top ray intersects the splitting plane beyond the ray's  $t_{\max}$  position and thus doesn't enter the far child. The bottom ray is facing away from the splitting plane, indicated by a negative  $t_{\text{split}}$  value. (b) The ray intersects the plane before the ray's  $t_{\min}$  value, indicating that the near child doesn't need processing.

*(Get node children pointers for ray)*  $\equiv$

299

```
const KdAccelNode *firstChild, *secondChild;
int belowFirst = (ray.o[axis] < node->SplitPos()) ||
    (ray.o[axis] == node->SplitPos() && ray.d[axis] <= 0);
if (belowFirst) {
    firstChild = node + 1;
    secondChild = &nodes[node->AboveChild()];
} else {
    firstChild = &nodes[node->AboveChild()];
    secondChild = node + 1;
}
```

KdAccelNode 286

KdAccelNode::SplitPos() 288

Ray::o 73

It may not be necessary to process both children of this node. Figure 4.19 shows some configurations where the ray only passes through one of the children. The ray will never

miss both children, since otherwise the current interior node should never have been visited.

The first `if` test in the following code corresponds to Figure 4.19(a): only the near node needs to be processed if it can be shown that the ray doesn't overlap the far node because it faces away from it or doesn't overlap it because  $t_{\text{split}} > t_{\text{max}}$ . Figure 4.19(b) shows the similar case tested in the second `if` test: the near node may not need processing if the ray doesn't overlap it. Otherwise, the `else` clause handles the case of both children needing processing; the near node will be processed next, and the far node goes on the todo list.

*(Advance to next child node, possibly enqueue other child)* ≡

299

```

if (tPlane > tMax || tPlane <= 0)
    node = firstChild;
else if (tPlane < tMin)
    node = secondChild;
else {
    (Enqueue secondChild in todo list 301)
    node = firstChild;
    tMax = tPlane;
}

```

*(Enqueue secondChild in todo list)* ≡

301

```

todo[todoPos].node = secondChild;
todo[todoPos].tMin = tPlane;
todo[todoPos].tMax = tMax;
++todoPos;

```

If the current node is a leaf, intersection tests are performed against the primitives in the leaf.

*(Check for intersections inside leaf node)* ≡

299

```

int nPrimitives = node->nPrimitives();
if (nPrimitives == 1) {
    const std::shared_ptr<Primitive> &p = primitives[node->onePrimitive];
    (Check one primitive inside leaf node 301)
} else {
    for (int i = 0; i < nPrimitives; ++i) {
        int index = primitiveIndices[node->primitiveIndicesOffset + i];
        const std::shared_ptr<Primitive> &p = primitives[index];
        (Check one primitive inside leaf node 301)
    }
}

```

Processing an individual primitive is just a matter of passing the intersection request on to the primitive.

*(Check one primitive inside leaf node)* ≡

301

```

if (p->Intersect(ray, isect))
    hit = true;

```

After doing the intersection tests at the leaf node, the next node to process is loaded from the todo array. If no more nodes remain, then the ray has passed through the tree without hitting anything.

```
(Grab next node to process from todo list) ≡ 299
if (todoPos > 0) {
    --todoPos;
    node = todo[todoPos].node;
    tMin = todo[todoPos].tMin;
    tMax = todo[todoPos].tMax;
}
else
    break;
```

Like the `BVHAccel`, the `KdTreeAccel` has a specialized intersection method for shadow rays that is not shown here. It is similar to the `Intersect()` method but calls the `Primitive::IntersectP()` method and returns `true` as soon as it finds any intersection without worrying about finding the closest one.

```
(KdTreeAccel Public Methods) ≡ 285
bool IntersectP(const Ray &ray) const;
```

## FURTHER READING

After the introduction of the ray-tracing algorithm, an enormous amount of research was done to try to find effective ways to speed it up, primarily by developing improved ray-tracing acceleration structures. Arvo and Kirk's chapter in *An Introduction to Ray Tracing* (Glassner 1989a) summarizes the state of the art as of 1989 and still provides an excellent taxonomy for categorizing different approaches to ray intersection acceleration.

Kirk and Arvo (1988) introduced the unifying principle of *meta-hierarchies*. They showed that by implementing acceleration data structures to conform to the same interface as is used for primitives in the scene, it's easy to mix and match different intersection acceleration schemes. `pbrt` follows this model since the `Aggregate` inherits from the `Primitive` base class.

### Grids

Fujimoto, Tanaka, and Iwata (1986) introduced uniform grids, a spatial subdivision approach where the scene bounds are decomposed into equally sized grid cells. More efficient grid traversal methods were described by Amanatides and Woo (1987) and Cleary and Wyvill (1988). Snyder and Barr (1987) described a number of key improvements to this approach and showed the use of grids for rendering extremely complex scenes. Hierarchical grids, where grid cells with many primitives in them are themselves refined into grids, were introduced by Jevans and Wyvill (1989). More complex techniques for hierarchical grids were developed by Cazals, Drettakis, and Puech (1995) and Klimaszewski and Sederberg (1997).

`Aggregate` 255  
`BVHAccel` 256  
`KdToDo::node` 298  
`KdToDo::tMax` 298  
`KdToDo::tMin` 298  
`KdTreeAccel` 285  
`Primitive` 248  
`Primitive::IntersectP()` 249  
`Ray` 73

Ize et al. (2006) developed an efficient algorithm for parallel construction of grids. One of their interesting findings was that grid construction performance quickly became limited by available memory bandwidth as the number of processing cores used increased.

Choosing an optimal grid resolution is important for getting good performance from grids. A good paper on this topic is by Ize et al. (2007), who provided a solid foundation for fully automatically selecting the resolution and for deciding when to refine into subgrids when using hierarchical grids. They derived theoretical results using a number of simplifying assumptions and then showed the applicability of the results to rendering real-world scenes. Their paper also includes a good selection of pointers to previous work in this area.

Lagae and Dutré (2008a) described an innovative representation for uniform grids based on hashing that has the desirable properties that not only does each primitive have a single index into a grid cell but also each cell has only a single primitive index. They show that this representation has very low memory usage and is still quite efficient.

Hunt and Mark (2008a) showed that building grids in perspective space, where the center of projection is the camera or a light source, can make tracing rays from the camera or light substantially more efficient. Although this approach requires multiple acceleration structures, the performance benefits from multiple specialized structures for different classes of rays can be substantial. Their approach is also notable in that it is in some ways a middle ground between rasterization and ray tracing.

### Bounding Volume Hierarchies

Clark (1976) first suggested using bounding volumes to cull collections of objects for standard visible-surface determination algorithms. Building on this work, Rubin and Whitted (1980) developed the first hierarchical data structures for scene representation for fast ray tracing, although their method depended on the user to define the hierarchy. Kay and Kajiya (1986) implemented one of the first practical object subdivision approaches based on bounding objects with collections of slabs. Goldsmith and Salmon (1987) described the first algorithm for automatically computing bounding volume hierarchies. Although their algorithm was based on estimating the probability of a ray intersecting a bounding volume based on the volume's surface area, it was much less effective than modern SAH BVH approaches.

The BVHAccel implementation in this chapter is based on the construction algorithm described by Wald (2007) and Günther et al. (2007). The bounding box test is the one introduced by Williams et al. (2005). An even more efficient bounding box test that does additional precomputation in exchange for higher performance when the same ray is tested for intersection against many bounding boxes was developed by Eisemann et al. (2007); we leave implementing their method for an exercise.

The BVH traversal algorithm used in pbrt was concurrently developed by a number of researchers; see the notes by Boulos and Haines (2006) for more details and background. Another option for tree traversal is that of Kay and Kajiya (1986); they maintained a heap of nodes ordered by ray distance. On GPUs, which have relatively limited amounts of on-chip memory, maintaining a stack of to-be-visited nodes for each ray may have a prohibitive memory cost. Foley and Sugerman (2005) introduced a “stackless” kd-tree traversal algorithm that periodically backtracks and searches starting from the tree

root to find the next node to visit rather than storing all nodes to visit explicitly. Laine (2010) made a number of improvements to this approach, reducing the frequency of retraversals from the tree root and applying the approach to BVHs.

A number of researchers have developed techniques for improving the quality of BVHs after construction. Yoon et al. (2007) and Kensler (2008) presented algorithms that make local adjustments to the BVH, and Kopta et al. (2012) reused BVHs over multiple frames of an animation, maintaining their quality by updating the parts that bound moving objects. See also Bittner et al. (2013), Karras and Aila (2013), and Bittner et al. (2014) for recent work in this area.

Most current methods for building BVHs are based on top-down construction of the tree, first creating the root node and then partitioning the primitives into children and continuing recursively. An alternative approach was demonstrated by Walter et al. (2008), who showed that bottom-up construction, where the leaves are created first and then agglomerated into parent nodes, is a viable option. Gu et al. (2013b) developed a much more efficient implementation of this approach and showed its suitability for parallel implementation.

One shortcoming of BVHs is that even a small number of relatively large primitives that have overlapping bounding boxes can substantially reduce the efficiency of the BVH: many of the nodes of the tree will be overlapping, solely due to the overlapping bounding boxes of geometry down at the leaves. Ernst and Greiner (2007) proposed “split clipping” as a solution; the restriction that each primitive only appears once in the tree is lifted, and the bounding boxes of large input primitives are subdivided into a set of tighter sub-bounds that are then used for tree construction. Dammertz and Keller (2008a) observed that the problematic primitives are the ones with a large amount of empty space in their bounding box relative to their surface area, so they subdivided the most egregious triangles and reported substantial performance improvements. Stich et al. (2009) developed an approach that splits primitives during BVH construction, making it possible to only split primitives when an SAH cost reduction was found. See also Popov et al.’s paper (2009) on a theoretically optimum BVH partitioning algorithm and its relationship to previous approaches, and Karras and Aila (2013) for improved criteria for deciding when to split triangles. Woop et al. (2014) developed an approach to building BVHs for long, thin geometry like hair and fur; because this sort of geometry is quite thin with respect to the volume of its bounding boxes, it normally has poor performance with most acceleration structures.

The memory requirements for BVHs can be significant. In our implementation, each node is 32 bytes. With up to 2 BVH tree nodes needed per primitive in the scene, the total overhead may be as high as 64 bytes per primitive. Cline et al. (2006) suggested a more compact representation for BVH nodes, at some expense of efficiency. First, they quantized the bounding box stored in each node using 8 or 16 bytes to encode its position with respect to the node’s parent’s bounding box. Second, they used *implicit indexing*, where the node  $i$ ’s children are at positions  $2i$  and  $2i + 1$  in the node array (assuming a  $2 \times$  branching factor). They showed substantial memory savings, with moderate performance impact. Bauszat et al. (2010) developed another space-efficient BVH representation. See also Segovia and Ernst (2010), who developed compact representations of both BVH nodes and triangle meshes.

Yoon and Manocha (2006) described algorithms for cache-efficient layout of BVHs and kd-trees and demonstrated performance improvements from using them. See also Ericson’s book (2004) for extensive discussion of this topic.

The linear BVH was introduced by Lauterbach et al. (2009). Pantaleoni and Luebke (2010) developed the HLBVH generalization, using the SAH at the upper levels of the tree. They also noted that the upper bits of the Morton coded values can be used to efficiently find clusters of primitives—both of these ideas are used in our HLBVH implementation. Garanzha et al. (2011) introduced further improvements to the HLBVH, most of them targeting GPU implementations.

Other than the HLBVH path, the BVH construction implementations in the BVHAccel here haven’t been parallelized. See Wald (2012) for an approach for high-performance parallel BVH construction using the SAH throughout.

### kd-trees

Glassner (1984) introduced the use of octrees for ray intersection acceleration. Use of the kd-tree for ray tracing was first described by Kaplan (1985). Kaplan’s tree construction algorithm always split nodes down their middle; MacDonald and Booth (1990) introduced the SAH approach, estimating ray–node traversal probabilities using relative surface areas. Naylor (1993) has also written on general issues of constructing good kd-trees. Havran and Bittner (2002) revisited many of these issues and introduced useful improvements. Adding a bonus factor to the SAH for tree nodes that are completely empty, as is done in our implementation, was suggested by Hurley et al. (2002). See Havran’s Ph.D. thesis (2000) for an excellent overview of high-performance kd-construction and traversal algorithms.

Jansen (1986) first developed the efficient ray traversal algorithm for kd-trees. Arvo (1988) also investigated this problem and discussed it in a note in *Ray Tracing News*. Sung and Shirley (1992) described a ray traversal algorithm’s implementation for a BSP-tree accelerator; our KdTreeAccel traversal code is loosely based on theirs.

The asymptotic complexity of the kd-tree construction algorithm in pbrt is  $O(n \log^2 n)$ . Wald and Havran (2006) showed that it’s possible to build kd-trees in  $(n \log n)$  time with some additional implementation complexity; they reported a 2 to 3× speedup in construction time for typical scenes.

The best kd-trees for ray tracing are built using “perfect splits,” where the primitive being inserted into the tree is clipped to the bounds of the current node at each step. This eliminates the issue that, for example, an object’s bounding box may intersect a node’s bounding box and thus be stored in it, even though the object itself doesn’t intersect the node’s bounding box. This approach was introduced by Havran and Bittner (2002) and discussed further by Hurley et al. (2002) and Wald and Havran (2006). See also Soupikov et al. (2008). Even with perfect splits, large primitives may still be stored in many kd-tree leaves; Choi et al. (2013) suggest storing some primitives in interior nodes to address this issue.

work in this area includes that of Shevtsov et al. (2007b) and Choi et al. (2010), who presented efficient parallel kd-tree construction algorithms with good scalability to multiple processors.

### The Surface Area Heuristic

A number of researchers have investigated improvements to the SAH since its introduction to ray tracing by MacDonald and Booth (1990). Fabianowski et al. (2009) derived a version that replaces the assumption that rays are uniformly distributed throughout space with the assumption that ray origins are uniformly distributed inside the scene’s bounding box. Hunt and Mark (2008b) introduced a new SAH that accounts for the fact that rays generally aren’t uniformly distributed but rather that many of them originate from a single point or a set of nearby points (cameras and light sources, respectively). Hunt (2008) showed how the SAH should be modified when the “mailboxing” optimization is being used, and Vinkler et al. (2012) used assumptions about the visibility of primitives to adjust their SAH cost. Ize and Hansen (2011) derived a “ray termination surface area heuristic” (RTSAH), which they use to adjust BVH traversal order for shadow rays in order to more quickly find intersections with occluders. See also Moulin et al. (2015), who adapted the SAH to account for shadow rays being occluded during kd-tree traversal.

Evaluating the SAH can be costly, particularly when many different splits or primitive partitions are being considered. One solution to this problem is to only compute it at a subset of the candidate points—for example, along the lines of the bucketing approach used in the BVHAccel in pbrt. Hurley et al. (2002) suggested this approach for building kd-trees, and Popov et al. (2006) discusses it in detail. Shevtsov et al. (2007b) introduced the improvement of binning the full extents of triangles, not just their centroids.

Hunt et al. (2006) noted that if you only have to evaluate the SAH at one point, for example, you don’t need to sort the primitives but only need to do a linear scan over them to compute primitive counts and bounding boxes at the point. They showed that approximating the SAH with a piecewise quadratic based on evaluating it at a number of individual positions and using that to choose a good split leads to effective trees. A similar approximation was used by Popov et al. (2006).

While the SAH has led to very effective kd-trees and BVHs, it has become clear that it isn’t perfect: a number of researchers have noted that it’s not unusual to encounter cases where a kd-tree or BVH with a higher SAH-estimated cost gives better performance than one with lower estimated cost. Aila et al. (2013) survey some of these results and propose two additional heuristics that help address them; one accounts for the fact that most rays start on surfaces—ray origins aren’t actually randomly distributed throughout the scene, and another accounts for SIMD divergence when multiple rays traverse the hierarchy together. While these new heuristics are effective at explaining why a given tree delivers the performance that it does, it’s not yet clear how to incorporate them into tree construction algorithms.

### Other Topics in Acceleration Structures

Weghorst, Hooper, and Greenberg (1984) discussed the trade-offs of using various shapes for bounding volumes and suggested projecting objects to the screen and using a z-buffer rendering to accelerate finding intersections for camera rays.

A number of researchers have investigated the applicability of general BSP trees, where the splitting planes aren't necessarily axis aligned, as they are with kd-trees. Kammaje and Mora (2007) built BSP trees using a preselected set of candidate splitting planes. Budge et al. (2008) developed a number of improvements to their approach, though their results only approached kd-tree performance in practice due to a slower construction stage and slower traversal than kd-trees. Ize et al. (2008) showed a BSP implementation that renders scenes faster than modern kd-trees but at the cost of extremely long construction times.

There are many techniques for traversing a collection of rays through the acceleration structure together, rather than just one at a time. This approach ("packet tracing") is an important component of high-performance ray tracing; it's discussed in more depth in Section 17.2.2.

Animated primitives present two challenges to ray tracers: first, renderers that try to reuse acceleration structures over multiple frames of an animation must update the acceleration structures if objects are moving. Wald et al. (2007a) showed how to incrementally update BVHs in this case, and Garanzha (2009) suggested creating clusters of nearby primitives and then building BVHs of those clusters (thus lightening the load on the BVH construction algorithm). A second problem is that for primitives that are moving quickly, the bounding boxes of their full motion over the frame time may be quite large, leading to many unnecessary ray-primitive intersection tests. Notable work on this issue includes Glassner (1988), who generalized ray tracing (and an octree for acceleration) to four dimensions, adding time. More recently, Grünschloß et al. (2011) developed improvements to BVHs for moving primitives. See also Wald et al.'s (2007b) survey paper on ray tracing animated scenes.

An innovative approach to acceleration structures was suggested by Arvo and Kirk (1987), who introduced a 5D data structure that subdivided based on both 3D spatial and 2D ray directions. Another interesting approach for scenes described with triangle meshes was developed by Lagae and Dutré (2008b): they computed a constrained tetrahedralization, where all triangle faces of the model are represented in the tetrahedralization. Rays are then stepped through tetrahedra until they intersect a triangle from the scene description. This approach is still a few times slower than the state-of-the-art in kd-trees and BVHs but is an interesting new way to think about the problem.

There is an interesting middle ground between kd-trees and BVHs, where the tree node holds a splitting plane for each child rather than just a single splitting plane. For example, this refinement makes it possible to do object subdivision in a kd-tree-like acceleration structure, putting each primitive in just one subtree and allowing the subtrees to overlap, while still preserving many of the benefits of efficient kd-tree traversal. Ooi et al. (1987) first introduced this refinement to kd-trees for storing spatial data, naming it the "spatial kd-tree" (skd-tree). Skd-trees have recently been applied to ray tracing by a number of researchers, including Zachmann (2002), Woop et al. (2006), Wächter and Keller (2006), Havran et al. (2006), and Zuniga and Uhlmann (2006).

When spatial subdivision approaches like grids or kd-trees are used, primitives may overlap multiple nodes of the structure and a ray may be tested for intersection with the same primitive multiple times as it passes through the structure. Arnaldi, Priol, and Bouatouch (1987) and Amanatides and Woo (1987) developed the "mailboxing"

technique to address this issue: each ray is given a unique integer identifier, and each primitive records the id of the last ray that was tested against it. If the ids match, then the intersection test is unnecessary and can be skipped.

While effective, this approach doesn't work well with a multi-threaded ray tracer. To address this issue, Benthin (2006) suggested storing a small per-ray hash table to record ids of recently intersected primitives. Shevtsov et al. (2007a) maintained a small array of the last  $n$  intersected primitive ids and searched it linearly before performing intersection tests. Although some primitives may still be checked multiple times with both of these approaches, they usually eliminate most redundant tests.

## EXERCISES

- ② 4.1 What kinds of scenes are worst-case scenarios for the two acceleration structures in pbrt? (Consider specific geometric configurations that the approaches will respectively be unable to handle well.) Construct scenes with these characteristics, and measure the performance of pbrt as you add more primitives. How does the worst case for one behave when rendered with the other?
- ② 4.2 Implement a hierarchical grid accelerator where you refine cells that have an excessive number of primitives overlapping them to instead hold a finer subgrid to store its geometry. (See, for example, Jevans and Wyvill (1989) for one approach to this problem and Ize et al. (2007) for effective methods for deciding when refinement is worthwhile.) Compare both accelerator construction performance and rendering performance to a non-hierarchical grid as well as to the accelerators in this chapter.
- ② 4.3 Implement smarter overlap tests for building accelerators. Using objects' bounding boxes to determine which sides of a kd-tree split they overlap can hurt performance by causing unnecessary intersection tests. Therefore, add a `bool Shape::Overlaps(const Bounds3f &)` const method to the shape interface that takes a world space bounding box and determines if the shape truly overlaps the given bound.

A default implementation could get the world bound from the shape and use that for the test, and specialized versions could be written for frequently used shapes. Implement this method for `Spheres` and `Triangles`, and modify `KdTreeAccel` to call it. You may find it helpful to read Akenine-Möller's paper (2001) on fast triangle-box overlap testing. Measure the change in pbrt's overall performance caused by this change, separately accounting for increased time spent building the acceleration structure and reduction in ray-object intersection time due to fewer intersections. For a variety of scenes, determine how many fewer intersection tests are performed thanks to this improvement.

- ② 4.4 Implement "split clipping" in pbrt's BVH implementation. Read the papers by Ernst and Greiner (2007), Dammertz and Keller (2008a), Stich et al. (2009), and Karras and Aila (2013), and implement one of their approaches to subdivide primitives with large bounding boxes relative to their surface area into multiple subprimitives for tree construction. (Doing so will probably require

modification to the Shape interface; you will probably want to design a new interface that allows some shapes to indicate that they are unable to subdivide themselves, so that you only need to implement this method for triangles, for example.) Measure the improvement for rendering actual scenes; a compelling way to gather this data is to do the experiment that Dammertz and Keller did, where a scene is rotated around an axis over progressive frames of an animation. Typically, many triangles that are originally axis aligned will have very loose bounding boxes as they rotate more, leading to a substantial performance degradation if split clipping isn't used.

- ② 4.5 The 30-bit Morton codes used for the HLBVH construction algorithm in the BVHAccel may be insufficient for large scenes (note that they can only represent  $2^{10} = 1024$  steps in each dimension.) Modify the BVHAccel to use 64-bit integers with 63-bit Morton codes for HLBVHs. Compare the performance of your approach to the original one with a variety of scenes. Are there scenes where performance is substantially improved? Are there any where there is a loss of performance?
- ② 4.6 Investigate alternative SAH cost functions for building BVHs or kd-trees. How much can a poor cost function hurt its performance? How much improvement can be had compared to the current one? (See the discussion in the “Further Reading” section for ideas about how the SAH may be improved.)
- ③ 4.7 Construction time for the BVHAccel and particularly the KdTreeAccel can be a meaningful portion of overall rendering time yet, other than HLBVHs, the implementations in this chapter do not parallelize building the acceleration structures. Investigate techniques for parallel construction of accelerators such as described by Wald (2007) and Shevtsov et al. (2007b), and implement one of them in pbrt. How much of a speedup do you achieve in accelerator construction? How does the speedup scale with additional processors? Measure how much of a speedup your changes translate to for overall rendering. For what types of scenes does your implementation have the greatest impact?
- ③ 4.8 The idea of using spatial data structures for ray intersection acceleration can be generalized to include spatial data structures that themselves hold other spatial data structures rather than just primitives. Not only could we have a grid that has subgrids inside the grid cells that have many primitives in them, but we could also have the scene organized into a hierarchical bounding volume where the leaf nodes are grids that hold smaller collections of spatially nearby primitives. Such hybrid techniques can bring the best of a variety of spatial data structure-based ray intersection acceleration methods. In pbrt, because both geometric primitives and intersection accelerators inherit from the Primitive base class and thus provide the same interface, it's easy to mix and match in this way.

BVHAccel 256  
KdTreeAccel 285  
Primitive 248  
Shape 123

Modify pbrt to build hybrid acceleration structures—for example, using a BVH to coarsely sort the scene geometry and then uniform grids at the leaves of the

tree to manage dense, spatially local collections of geometry. Measure the running time and memory use for rendering schemes with this method compared to the current accelerators.

- ② 4.9 Eisemann et al. (2007) described an even more efficient ray–box intersection test than is used in the BVHAccel. It does more computation at the start for each ray but makes up for this work with fewer computations to do tests for individual bounding boxes. Implement their method in pbrt, and measure the change in rendering time for a variety of scenes. Are there simple scenes where the additional upfront work doesn’t pay off? How does the improvement for highly complex scenes compare to the improvement for simpler scenes?
- ② 4.10 Read the paper by Segovia and Ernst (2010) on memory-efficient BVHs, and implement their approach in pbrt. How does memory usage with their approach compare to that for the BVHAccel? Compare rendering performance with your approach to pbrt’s current performance. Discuss how your results compare to the results reported in their paper.
- ② 4.11 Modify pbrt to use the “mailboxing” optimization in the KdTreeAccel to avoid repeated intersections with primitives that overlap multiple kd-tree nodes. Given that pbrt is multi-threaded, you will probably do best to consider either the hashed mailing approach suggested by Benthin (2006) or the inverse mailing algorithm of Shevtsov et al. (2007a). Measure the performance change compared to the current implementation for a variety of scenes. How does the change in running time relates to changes in reported statistics about the number of ray–primitive intersection tests?
- ③ 4.12 It is often possible to introduce some approximation into the computation of shadows from very complex geometry (consider, e.g., the branches and leaves of a tree casting a shadow). Lacewell et al. (2008) suggested augmenting the acceleration structure with a prefiltered directionally varying representation of occlusion for regions of space. As shadow rays pass through these regions, an approximate visibility probability can be returned rather than a binary result, and the cost of tree traversal and object intersection tests is reduced. Implement such an approach in pbrt, and compare its performance to the current implementation. Do you see any changes in rendered images?

This page intentionally left blank



# CHAPTER FIVE

## 05 COLOR AND RADIOMETRY

In order to precisely describe how light is represented and sampled to compute images, we must first establish some background in *radiometry*—the study of the propagation of electromagnetic radiation in an environment. Of particular interest in rendering are the wavelengths ( $\lambda$ ) of electromagnetic radiation between approximately 380 nm and 780 nm, which account for light visible to humans.<sup>1</sup> The lower wavelengths ( $\lambda \approx 400$  nm) are the bluish colors, the middle wavelengths ( $\lambda \approx 550$  nm) are the greens, and the upper wavelengths ( $\lambda \approx 650$  nm) are the reds.

In this chapter, we will introduce four key quantities that describe electromagnetic radiation: flux, intensity, irradiance, and radiance. These radiometric quantities are each described by their *spectral power distribution* (SPD)—a distribution function of wavelength that describes the amount of light at each wavelength. The `Spectrum` class, which is defined in Section 5.1, is used to represent SPDs in `pbrt`.

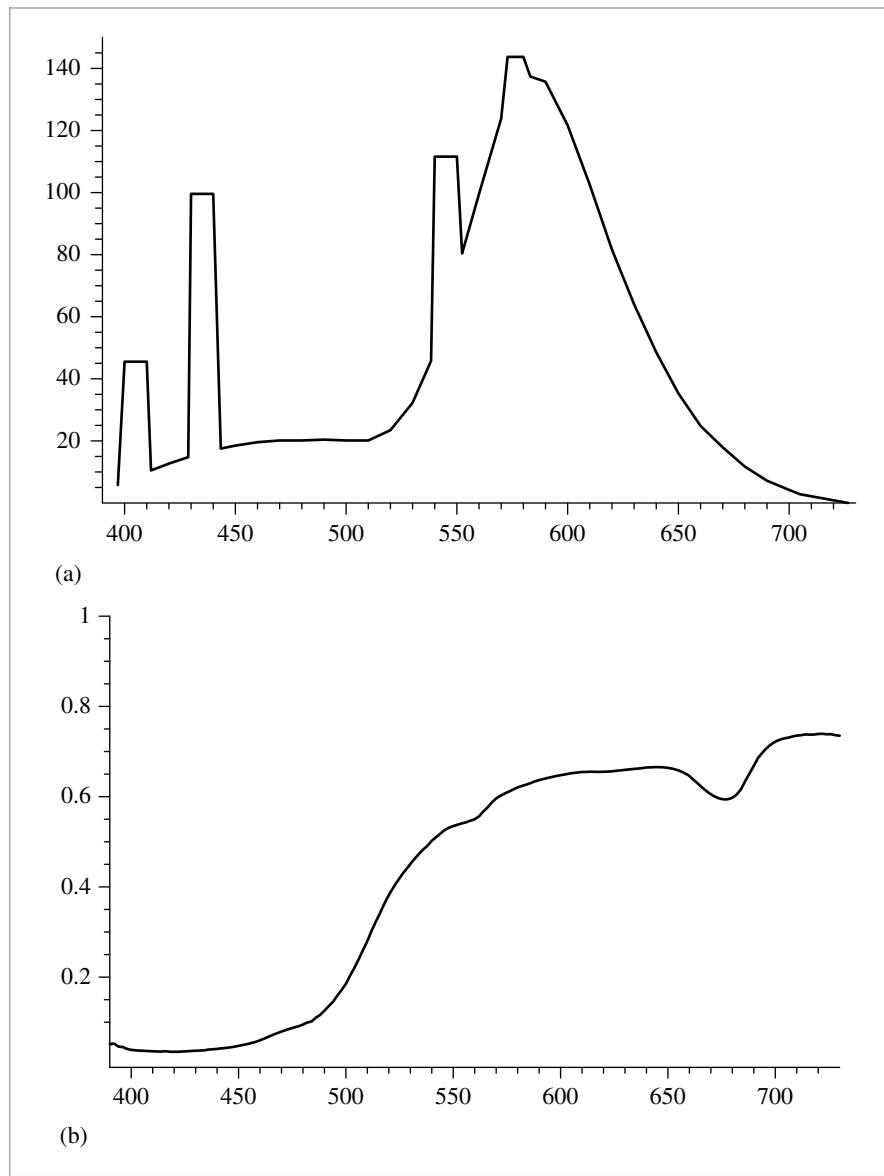
### 5.1 SPECTRAL REPRESENTATION

The SPDs of real-world objects can be quite complicated; Figure 5.1 shows graphs of the spectral distribution of emission from a fluorescent light and the spectral distribution of the reflectance of lemon skin. A renderer doing computations with SPDs needs a compact, efficient, and accurate way to represent functions like these. In practice, some trade-off needs to be made between these qualities.

A general framework for investigating these issues can be developed based on the problem of finding good *basis functions* to represent SPDs. The idea behind basis functions

---

<sup>1</sup> The full range of perceptible wavelengths slightly extends beyond this interval, though the eye's sensitivity at these wavelengths is lower by many orders of magnitude. The range 360–830 nm is often used as a conservative bound when tabulating spectral curves.



**Figure 5.1:** (a) Spectral power distributions of a fluorescent light and (b) the reflectance of lemon skin. Wavelengths around 400 nm are bluish colors, greens and yellows are in the middle range of wavelengths, and reds have wavelengths around 700 nm. The fluorescent light's SPD is even spikier than shown here, where the SPDs have been binned into 10-nm ranges; it actually emits much of its illumination at single discrete frequencies.

is to map the infinite-dimensional space of possible SPD functions to a low-dimensional space of coefficients  $c_i \in \mathbb{R}$ . For example, a trivial basis function is the constant function  $B(\lambda) = 1$ . An arbitrary SPD would be represented in this basis by a single coefficient  $c$  equal to its average value, so that its approximation would be  $cB(\lambda) = c$ . This is obviously a poor approximation, since most SPDs are much more complex than this single basis function is capable of representing accurately.

Many different basis functions have been investigated for spectral representation in computer graphics; the “Further Reading” section cites a number of papers and further resources on this topic. Different sets of basis functions can offset substantially different trade-offs in the complexity of the key operations like converting an arbitrary SPD into a set of coefficients (projecting it into the basis), computing the coefficients for the SPD given by the product of two SPDs expressed in the basis, and so on. In this chapter, we’ll introduce two representations that can be used for spectra in pbrt: `RGBSpectrum`, which follows the typical computer graphics practice of representing SPDs with coefficients representing a mixture of red, green, and blue colors, and `SampledSpectrum`, which represents the SPD as a set of point samples over a range of wavelengths.

### 5.1.1 THE Spectrum TYPE

Throughout pbrt, we have been careful to implement all computations involving SPDs in terms of the `Spectrum` type, using a specific set of built-in operators (addition, multiplication, etc.). The `Spectrum` type hides the details of the particular spectral representation used, so that changing this detail of the system only requires changing the `Spectrum` implementation; other code can remain unchanged. The implementations of the `Spectrum` type are in the files `core/spectrum.h` and `core/spectrum.cpp`.

The selection of which spectrum representation is used for the `Spectrum` type in pbrt is done with a `typedef` in the file `core/pbrt.h`. By default, pbrt uses the more efficient but less accurate RGB representation.

```
(Global Forward Declarations) ≡
    typedef RGBSpectrum Spectrum;
    // typedef SampledSpectrum Spectrum;
```

We have not written the system such that the selection of which `Spectrum` implementation to use could be resolved at run time; to switch to a different representation, the entire system must be recompiled. One advantage to this design is that many of the various `Spectrum` methods can be implemented as short functions that can be inlined by the compiler, rather than being left as stand-alone functions that have to be invoked through the relatively slow virtual method call mechanism. Inlining frequently used short functions like these can give a substantial improvement in performance. A second advantage is that structures in the system that hold instances of the `Spectrum` type can hold them directly rather than needing to allocate them dynamically based on the spectral representation chosen at run time.

`CoefficientSpectrum` 316  
`RGBSpectrum` 332  
`SampledSpectrum` 319  
`Spectrum` 315

### 5.1.2 CoefficientSpectrum IMPLEMENTATION

Both of the representations implemented in this chapter are based on storing a fixed number of samples of the SPD. Therefore, we’ll start by defining the `CoefficientSpectrum`

template class, which represents a spectrum as a particular number of samples given as the `nSpectrumSamples` template parameter. Both `RGBSpectrum` and `SampledSpectrum` are partially implemented by inheriting from `CoefficientSpectrum`.

*(Spectrum Declarations) ≡*

```
template <int nSpectrumSamples> class CoefficientSpectrum {
public:
    (CoefficientSpectrum Public Methods 316)
    (CoefficientSpectrum Public Data 318)
protected:
    (CoefficientSpectrum Protected Data 316)
};
```

One `CoefficientSpectrum` constructor is provided; it initializes a spectrum with a constant value across all wavelengths.

*(CoefficientSpectrum Public Methods) ≡*

316

```
CoefficientSpectrum(Float v = 0.f) {
    for (int i = 0; i < nSpectrumSamples; ++i)
        c[i] = v;
}
```

*(CoefficientSpectrum Protected Data) ≡*

316

```
Float c[nSpectrumSamples];
```

A variety of arithmetic operations on `Spectrum` objects are needed; the implementations in `CoefficientSpectrum` are all straightforward. First, we define operations to add pairs of spectral distributions. For the sampled representation, it's easy to show that each sample value for the sum of two SPDs is equal to the sum of the corresponding sample values.

*(CoefficientSpectrum Public Methods) +≡*

316

```
CoefficientSpectrum &operator+=(const CoefficientSpectrum &s2) {
    for (int i = 0; i < nSpectrumSamples; ++i)
        c[i] += s2.c[i];
    return *this;
}
```

*(CoefficientSpectrum Public Methods) +≡*

316

```
CoefficientSpectrum operator+(const CoefficientSpectrum &s2) const {
    CoefficientSpectrum ret = *this;
    for (int i = 0; i < nSpectrumSamples; ++i)
        ret.c[i] += s2.c[i];
    return ret;
}
```

Similarly, subtraction, multiplication, division, and unary negation are defined component-wise. These methods are very similar to the ones already shown, so we won't include them here. `prbt` also provides equality and inequality tests, also not included here.

It is often useful to know if a spectrum represents an SPD with value zero everywhere. If, for example, a surface has zero reflectance, the light transport routines can avoid

`CoefficientSpectrum` 316  
`CoefficientSpectrum::c` 316  
`Float` 1062  
`nSpectrumSamples` 316  
`RGBSpectrum` 332  
`SampledSpectrum` 319  
`Spectrum` 315

the computational cost of casting reflection rays that have contributions that would eventually be multiplied by zeros and thus do not need to be traced.

```
(CoefficientSpectrum Public Methods) +≡ 316
    bool IsBlack() const {
        for (int i = 0; i < nSpectrumSamples; ++i)
            if (c[i] != 0.) return false;
        return true;
    }
```

The `Spectrum` implementation (and thus the `CoefficientSpectrum` implementation) must also provide implementations of a number of slightly more esoteric methods, including those that take the square root of an SPD or raise the function it represents to a given power. These are needed for some of the computations performed by the `Fresnel` classes in Chapter 8, for example. The implementation of `Sqrt()` takes the square root of each component to give the square root of the SPD. The implementations of `Pow()` and `Exp()` are analogous and won't be included here.

```
(CoefficientSpectrum Public Methods) +≡ 316
    friend CoefficientSpectrum Sqrt(const CoefficientSpectrum &s) {
        CoefficientSpectrum ret;
        for (int i = 0; i < nSpectrumSamples; ++i)
            ret.c[i] = std::sqrt(s.c[i]);
        return ret;
    }
```

It's frequently useful to be able to linearly interpolate between two SPDs with a parameter  $t$ .

*(Spectrum Inline Functions) ≡*

```
inline Spectrum Lerp(Float t, const Spectrum &s1, const Spectrum &s2) {
    return (1 - t) * s1 + t * s2;
}
```

Some portions of the image processing pipeline will want to clamp a spectrum to ensure that the function it represents is within some allowable range.

```
(CoefficientSpectrum Public Methods) +≡ 316
    CoefficientSpectrum Clamp(Float low = 0, Float high = Infinity) const {
        CoefficientSpectrum ret;
        for (int i = 0; i < nSpectrumSamples; ++i)
            ret.c[i] = ::Clamp(c[i], low, high);
        return ret;
    }
```

Finally, we provide a debugging routine to check if any of the sample values of the SPD is the not-a-number (NaN floating-point value). This situation can happen due to an accidental division by 0; `Assert()`s throughout the system use this method to catch this case close to where it happens.

```
(CoefficientSpectrum Public Methods) +≡ 316
    bool HasNaNs() const {
        for (int i = 0; i < nSpectrumSamples; ++i)
            if (std::isnan(c[i])) return true;
        return false;
    }
```

Most of the spectral computations in `pbrt` can be implemented using the basic operations we have defined so far. However, in some cases it's necessary to be able to iterate over a set of spectral samples that represent an SPD—for example to perform a spectral sample-based table lookup or to evaluate a piecewise function over wavelengths. Classes that need this functionality in `pbrt` include the `TabulatedBSSRDF` class, which is used for subsurface scattering, and the `HomogeneousMedium` and `GridDensityMedium` classes.

For these uses, `CoefficientSpectrum` provides a public constant, `nSamples`, that gives the number of samples used to represent the SPD and an `operator[]` method to access individual sample values.

```
(CoefficientSpectrum Public Data) ≡ 316
    static const int nSamples = nSpectrumSamples;
```

```
(CoefficientSpectrum Public Methods) +≡ 316
    Float &operator[](int i) {
        return c[i];
    }
```

Note that the presence of this sample accessor imposes the implicit assumption that the spectral representation is a set of coefficients that linearly scale a fixed set of basis functions. If, for example, a `Spectrum` implementation instead represented SPDs as a sum of Gaussians where the coefficients  $c_i$  alternately scaled the Gaussians and set their width,

$$S(\lambda) = \sum_i^N c_{2i} e^{-c_{2i+1}},$$

then the code that currently uses this accessor would need to be modified, perhaps to instead operate on a version of the SPD that had been converted to a set of linear coefficients. While this crack in the `Spectrum` abstraction is not ideal, it simplifies other parts of the current system and isn't too hard to clean up if one adds spectral representations, where this assumption isn't correct.

## 5.2 THE SampledSpectrum CLASS

`SampledSpectrum` uses the `CoefficientSpectrum` infrastructure to represent an SPD with uniformly spaced samples between a starting and an ending wavelength. The wavelength range covers from 400 nm to 700 nm—the range of the visual spectrum where the human visual system is most sensitive. The number of samples, 60, is generally more than enough to accurately represent complex SPDs for rendering. (See the “Further Reading” section for background on sampling rate requirements for SPDs.) Thus, the first sample

`CoefficientSpectrum` [316](#)  
`CoefficientSpectrum::c` [316](#)  
`Float` [1062](#)  
`GridDensityMedium` [690](#)  
`HomogeneousMedium` [689](#)  
`nSpectrumSamples` [316](#)  
`TabulatedBSSRDF` [696](#)

represents the wavelength range [400, 405), the second represents [405, 410), and so forth. These values can easily be changed here as needed.

```
(Spectrum Utility Declarations) ≡
    static const int sampledLambdaStart = 400;
    static const int sampledLambdaEnd = 700;
    static const int nSpectralSamples = 60;

(Spectrum Declarations) +≡
class SampledSpectrum : public CoefficientSpectrum<nSpectralSamples> {
public:
    (SampledSpectrum Public Methods 319)
private:
    (SampledSpectrum Private Data 324)
};
```

By inheriting from the `CoefficientSpectrum` class, `SampledSpectrum` automatically has all of the basic spectrum arithmetic operations defined earlier. The main methods left to define for it are those that initialize it from spectral data and convert the SPD it represents to other spectral representations (such as RGB). The constructor for initializing it with a constant SPD is straightforward.

```
(SampledSpectrum Public Methods) ≡ 319
SampledSpectrum(Float v = 0.f) : CoefficientSpectrum(v) { }
```

We will often be provided spectral data as a set of  $(\lambda_i, v_i)$  samples, where the  $i$ th sample has some value  $v_i$  at wavelength  $\lambda_i$ . In general, the samples may have an irregular spacing and there may be more or fewer of them than the `SampledSpectrum` stores. (See the directory `scenes/spds` in the `pbrt` distribution for a variety of SPDs for use with `pbrt`, many of them with irregular sample spacing.)

The `FromSampled()` method takes arrays of SPD sample values `v` at given wavelengths `lambda` and uses them to define a piecewise linear function to represent the SPD. For each SPD sample in the `SampledSpectrum`, it uses the `AverageSpectrumSamples()` utility function, defined below, to compute the average of the piecewise linear function over the range of wavelengths that each SPD sample is responsible for.

```
(SampledSpectrum Public Methods) +≡ 319
static SampledSpectrum FromSampled(const Float *lambda,
                                    const Float *v, int n) {
    (Sort samples if unordered, use sorted for returned spectrum 320)
    SampledSpectrum r;
    for (int i = 0; i < nSpectralSamples; ++i) {
        (Compute average value of given SPD over ith sample's range 320)
    }
    return r;
}
```

AverageSpectrumSamples() 321  
 CoefficientSpectrum 316  
 Float 1062  
 nSpectralSamples 319  
 SampledSpectrum 319

The `AverageSpectrumSamples()` function requires that the  $(\lambda_i, v_i)$  values be sorted by wavelength. The `SpectrumSamplesSorted()` function checks that they are; if not,

`SortSpectrumSamples()` sorts them. Note that we allocate new storage for the sorted samples and do not modify the values passed in by the caller; in general, doing so would likely be unexpected behavior for a user of this function (who shouldn't need to worry about these requirements of its specific implementation). We won't include the implementations of either of these two functions here, as they are straightforward.

*(Sort samples if unordered, use sorted for returned spectrum) ≡* 319, 333

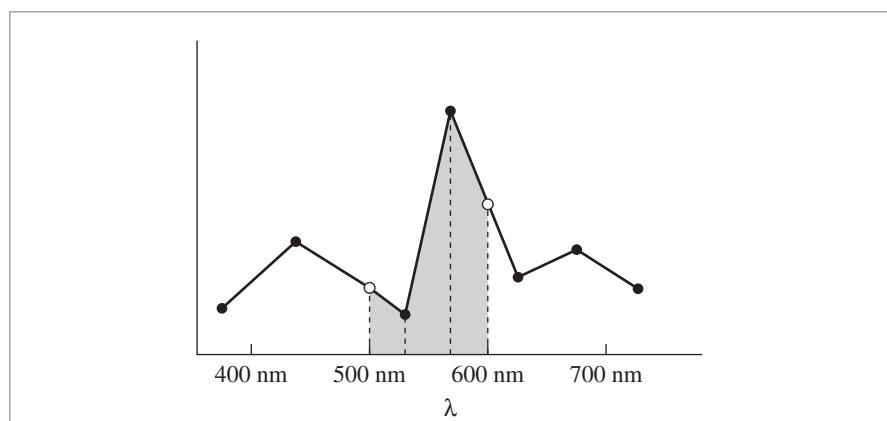
```
if (!SpectrumSamplesSorted(&lambda, v, n)) {
    std::vector<Float> slambda(&lambda[0], &lambda[n]);
    std::vector<Float> sv(&v[0], &v[n]);
    SortSpectrumSamples(&slambda[0], &sv[0], n);
    return FromSampled(&slambda[0], &sv[0], n);
}
```

In order to compute the value for the  $i$ th spectral sample, we compute the range of wavelengths that it's responsible for— $\lambda_{min}$  to  $\lambda_{max}$ —and use the `AverageSpectrumSamples()` function to compute the average value of the given piecewise linear SPD over that range. This is a 1D instance of sampling and reconstruction, a topic that will be discussed in more detail in Chapter 7.

*(Compute average value of given SPD over  $i$ th sample's range) ≡* 319

```
Float lambda0 = Lerp(Float(i) / Float(nSpectralSamples),
                      sampledLambdaStart, sampledLambdaEnd);
Float lambda1 = Lerp(Float(i + 1) / Float(nSpectralSamples),
                      sampledLambdaStart, sampledLambdaEnd);
r.c[i] = AverageSpectrumSamples(lambda, v, n, lambda0, lambda1);
```

Figure 5.2 shows the basic approach taken by `AverageSpectrumSamples()`: it iterates over each of the linear segments between samples that are partially or fully within the range of



**Figure 5.2:** When resampling an irregularly defined SPD, we need to compute the average value of the piecewise linear function defined by the SPD samples. Here, we want to average the value from 500 nm to 600 nm—the shaded region under the plot. The `FromSampled()` function does this by computing the areas of each of the regions denoted by dashed lines in this figure.

[AverageSpectrumSamples\(\) 321](#)  
[CoefficientSpectrum::c 316](#)  
[Float 1062](#)  
[Lerp\(\) 1079](#)  
[nSpectralSamples 319](#)  
[sampledLambdaEnd 319](#)  
[sampledLambdaStart 319](#)  
[SampledSpectrum::  
FromSampled\(\) 319](#)  
[SortSpectrumSamples\(\) 320](#)  
[SpectrumSamplesSorted\(\) 319](#)

wavelengths, `lambdaStart` to `lambdaEnd`. For each such segment, it computes the average value over its range, scales the average by the wavelength range the segment covers, and accumulates a sum of these values. The final average value is this sum divided by the total wavelength range.

*(Spectrum Method Definitions) ≡*

```
Float AverageSpectrumSamples(const Float *lambda, const Float *vals,
    int n, Float lambdaStart, Float lambdaEnd) {
    <Handle cases with out-of-bounds range or single sample only 321>
    Float sum = 0;
    <Add contributions of constant segments before/after samples 321>
    <Advance to first relevant wavelength segment 322>
    <Loop over wavelength sample segments and add contributions 322>
    return sum / (lambdaEnd - lambdaStart);
}
```

The function starts by checking for and handling the edge cases where the range of wavelengths to average over is outside the range of provided wavelengths or the case where there is only a single sample, in which case the average value is trivially computed. We assume that the SPD has a constant value (the values at the two respective endpoints) outside of the provided sample range; if this isn't a reasonable assumption for a particular set of data, the data provided should have explicit values of (for example) 0 at the endpoints.

*(Handle cases with out-of-bounds range or single sample only) ≡*

```
if (lambdaEnd <= lambda[0])      return vals[0];
if (lambdaStart >= lambda[n - 1]) return vals[n - 1];
if (n == 1) return vals[0];
```

321

Having handled these cases, the next step is to check to see if part of the range to average over goes beyond the first and/or last sample value. If so, we accumulate the contribution of the constant segment(s), scaled by the out-of-bounds wavelength range.

*(Add contributions of constant segments before/after samples) ≡*

```
if (lambdaStart < lambda[0])
    sum += vals[0] * (lambda[0] - lambdaStart);
if (lambdaEnd > lambda[n-1])
    sum += vals[n - 1] * (lambdaEnd - lambda[n - 1]);
```

321

And now we advance to the first index  $i$  where the starting wavelength of the interpolation range overlaps the segment from  $\lambda_i$  to  $\lambda_{i+1}$ . A more efficient implementation would use a binary search rather than a linear search here.<sup>2</sup> However, this code is currently only called at scene initialization time, so the lack of these optimizations doesn't currently impact rendering performance.

---

Float 1062

2 An even more efficient implementation would take advantage of the fact that the calling code will generally ask for interpolated values over a series of adjacent wavelength ranges, and possibly take all of the ranges in a single call. It could then incrementally find the starting index for the next interpolation starting from the end of the previous one.

*(Advance to first relevant wavelength segment) ≡*

321

```
int i = 0;
while (lambdaStart > lambda[i + 1]) ++i;
```

The loop below iterates over each of the linear segments that the averaging range overlaps. For each one, it computes the average value over the wavelength range segLambdaStart to segLambdaEnd by averaging the values of the function at those two points. The values in turn are computed by `interp()`, a lambda function that linearly interpolates between the two endpoints at the given wavelength.

The `std::min()` and `std::max()` calls below compute the wavelength range to average over within the segment; note that they naturally handle the cases where `lambdaStart`, `lambdaEnd`, or both of them are within the current segment.

*(Loop over wavelength sample segments and add contributions) ≡*

321

```
auto interp = [lambda, vals](Float w, int i) {
    return Lerp((w - lambda[i]) / (lambda[i + 1] - lambda[i]),
                vals[i], vals[i + 1]);
};

for (; i+1 < n && lambdaEnd >= lambda[i]; ++i) {
    Float segLambdaStart = std::max(lambdaStart, lambda[i]);
    Float segLambdaEnd = std::min(lambdaEnd, lambda[i + 1]);
    sum += 0.5 * (interp(segLambdaStart, i) + interp(segLambdaEnd, i)) *
        (segLambdaEnd - segLambdaStart);
}
```

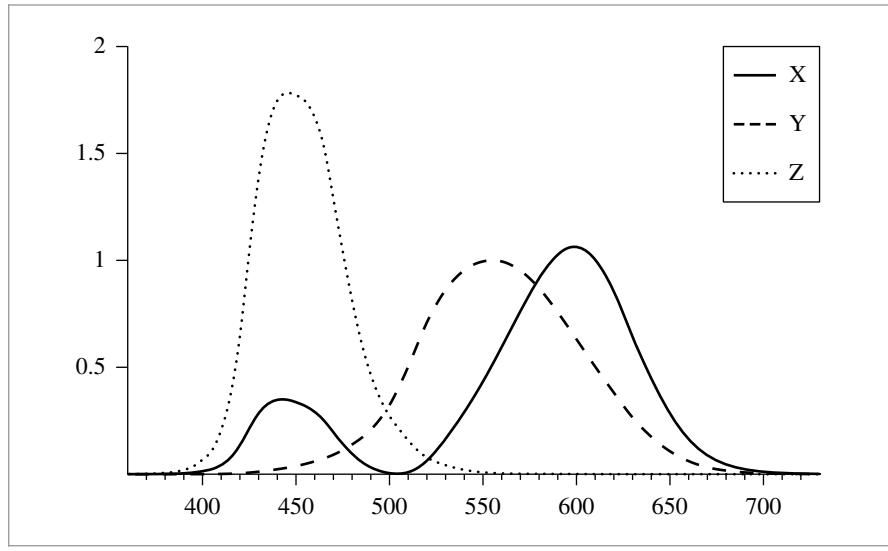
### 5.2.1 XYZ COLOR

A remarkable property of the human visual system makes it possible to represent colors for human perception with just three floating-point numbers. The *tristimulus theory* of color perception says that all visible SPDs can be accurately represented for human observers with three values,  $x_\lambda$ ,  $y_\lambda$ , and  $z_\lambda$ . Given an SPD  $S(\lambda)$ , these values are computed by integrating its product with the *spectral matching curves*  $X(\lambda)$ ,  $Y(\lambda)$ , and  $Z(\lambda)$ :

$$\begin{aligned} x_\lambda &= \frac{1}{\int Y(\lambda)d\lambda} \int_\lambda S(\lambda) X(\lambda) d\lambda \\ y_\lambda &= \frac{1}{\int Y(\lambda)d\lambda} \int_\lambda S(\lambda) Y(\lambda) d\lambda \\ z_\lambda &= \frac{1}{\int Y(\lambda)d\lambda} \int_\lambda S(\lambda) Z(\lambda) d\lambda. \end{aligned} \quad [5.1]$$

These curves were determined by the Commission Internationale de l'Éclairage (CIE) standards body after a series of experiments with human test subjects and are graphed in Figure 5.3. It is believed that these matching curves are generally similar to the responses of the three types of color-sensitive cones in the human retina. Remarkably, SPDs with substantially different distributions may have very similar  $x_\lambda$ ,  $y_\lambda$ , and  $z_\lambda$  values. To the human observer, such SPDs actually appear the same visually. Pairs of such spectra are called *metamers*.

Float 1062



**Figure 5.3: Computing the XYZ Values for an Arbitrary SPD.** The SPD is convolved with each of the three matching curves to compute the values  $x_\lambda$ ,  $y_\lambda$ , and  $z_\lambda$ , using Equation (5.1).

This brings us to a subtle point about representations of spectral power distributions. Most color spaces attempt to model colors that are visible to humans and therefore use only three coefficients, exploiting the tristimulus theory of color perception. Although XYZ works well to represent a given SPD to be displayed for a human observer, it is *not* a particularly good set of basis functions for spectral computation. For example, although XYZ values would work well to describe the perceived color of lemon skin or a fluorescent light individually (recall Figure 5.1), the product of their respective XYZ values is likely to give a noticeably different XYZ color than the XYZ value computed by multiplying more accurate representations of their SPDs and *then* computing the XYZ value.

pbrt provides the values of the standard  $X(\lambda)$ ,  $Y(\lambda)$ , and  $Z(\lambda)$  response curves sampled at 1-nm increments from 360 nm to 830 nm. The wavelengths of the  $i$ th sample in the arrays below are given by the  $i$ th element of `CIE_lambda`; having the wavelengths of the samples explicitly represented in this way makes it easy to pass the XYZ samples into functions like `AverageSpectrumSamples()` that take such an array of wavelengths as a parameter.

```
<Spectral Data Declarations> ≡
    static const int nCIESamples = 471;
    extern const Float CIE_X[nCIESamples];
    extern const Float CIE_Y[nCIESamples];
    extern const Float CIE_Z[nCIESamples];
    extern const Float CIE_lambda[nCIESamples];
```

`AverageSpectrumSamples()` 321

`Float` 1062

`nCIESamples` 323

`SampledSpectrum` 319

`SampledSpectrum` uses these samples to compute the XYZ matching curves in its spectral representation (i.e., themselves as `SampledSpectrums`).

<i>(SampledSpectrum Private Data) ≡</i>	<b>319</b>
static SampledSpectrum X, Y, Z;	

The `SampledSpectrum` XYZ matching curves are computed in the `SampledSpectrum::Init()` method, which is called at system startup time by the `pbrtInit()` function defined in Section B.2.

<i>(SampledSpectrum Public Methods) +≡</i>	<b>319</b>
static void Init() {	
<i>(Compute XYZ matching functions for SampledSpectrum 324)</i>	
<i>(Compute RGB to spectrum functions for SampledSpectrum)</i>	
}	

<i>(General pbrt Initialization) ≡</i>	<b>1109</b>
SampledSpectrum::Init();	

Given the wavelength range and number of samples for `SampledSpectrum`, computing the values of the matching functions for each sample is just a matter of computing the sample's wavelength range and using the `AverageSpectrumSamples()` routine.

<i>(Compute XYZ matching functions for SampledSpectrum) ≡</i>	<b>324</b>
for (int i = 0; i < nSpectralSamples; ++i) {	
Float w10 = Lerp(Float(i) / Float(nSpectralSamples),	
sampledLambdaStart, sampledLambdaEnd);	
Float w11 = Lerp(Float(i + 1) / Float(nSpectralSamples),	
sampledLambdaStart, sampledLambdaEnd);	
X.c[i] = AverageSpectrumSamples(CIE_lambda, CIE_X, nCIESamples,	
w10, w11);	
Y.c[i] = AverageSpectrumSamples(CIE_lambda, CIE_Y, nCIESamples,	
w10, w11);	
Z.c[i] = AverageSpectrumSamples(CIE_lambda, CIE_Z, nCIESamples,	
w10, w11);	
}	

All `Spectrum` implementations in `pbrt` must provide a method that converts their SPD to  $(x_\lambda, y_\lambda, z_\lambda)$  coefficients. This method is called, for example, in the process of updating pixels in the image. When a `Spectrum` representing the light carried along a ray from the camera is provided to the `Film`, the `Film` converts the SPD into XYZ coefficients as a first step in the process of finally turning them into RGB values used for storage and/or display.

To compute XYZ coefficients, `SampledSpectrum` computes the integrals from Equation (5.1) with a Riemann sum:

$$x_\lambda \approx \frac{1}{\int Y(\lambda) d\lambda} \frac{\lambda_{\text{end}} - \lambda_{\text{start}}}{N} \sum_{i=0}^{N-1} X_i c_i,$$

and so forth.

AverageSpectrumSamples() 321  
 CIE\_lambda 323  
 CIE\_X 323  
 CIE\_Y 323  
 CIE\_Z 323  
 Film 484  
 Float 1062  
 Lerp() 1079  
 nCIESamples 323  
 nSpectralSamples 319  
 pbrtInit() 1109  
 sampledLambdaEnd 319  
 sampledLambdaStart 319  
 SampledSpectrum 319  
 SampledSpectrum::Init() 324  
 Spectrum 315

```
(SampledSpectrum Public Methods) +≡
    void ToXYZ(Float xyz[3]) const {
        xyz[0] = xyz[1] = xyz[2] = 0.f;
        for (int i = 0; i < nSpectralSamples; ++i) {
            xyz[0] += X.c[i] * c[i];
            xyz[1] += Y.c[i] * c[i];
            xyz[2] += Z.c[i] * c[i];
        }
        Float scale = Float(sampledLambdaEnd - sampledLambdaStart) /
            Float(CIE_Y_integral * nSpectralSamples);
        xyz[0] *= scale;
        xyz[1] *= scale;
        xyz[2] *= scale;
    }
```

The value of the integral  $\int Y(\lambda)d\lambda$  is used in a number of calculations like these; it's therefore useful to have its value available directly via the `CIE_Y_integral` constant.

```
(Spectral Data Declarations) +≡
    static const Float CIE_Y_integral = 106.856895;
```

The `y` coefficient of XYZ color is closely related to *luminance*, which measures the perceived brightness of a color. Luminance is described in more detail in Section 5.4.3. We provide a method to compute `y` alone in a separate method as often only the luminance of a spectrum is desired. (For example, some of the light transport algorithms in Chapters 14–16 use luminance as a measure of the relative importance of light-carrying paths through the scene.)

```
(SampledSpectrum Public Methods) +≡
    Float y() const {
        Float yy = 0.f;
        for (int i = 0; i < nSpectralSamples; ++i)
            yy += Y.c[i] * c[i];
        return yy * Float(sampledLambdaEnd - sampledLambdaStart) /
            Float(CIE_Y_integral * nSpectralSamples);
    }
```

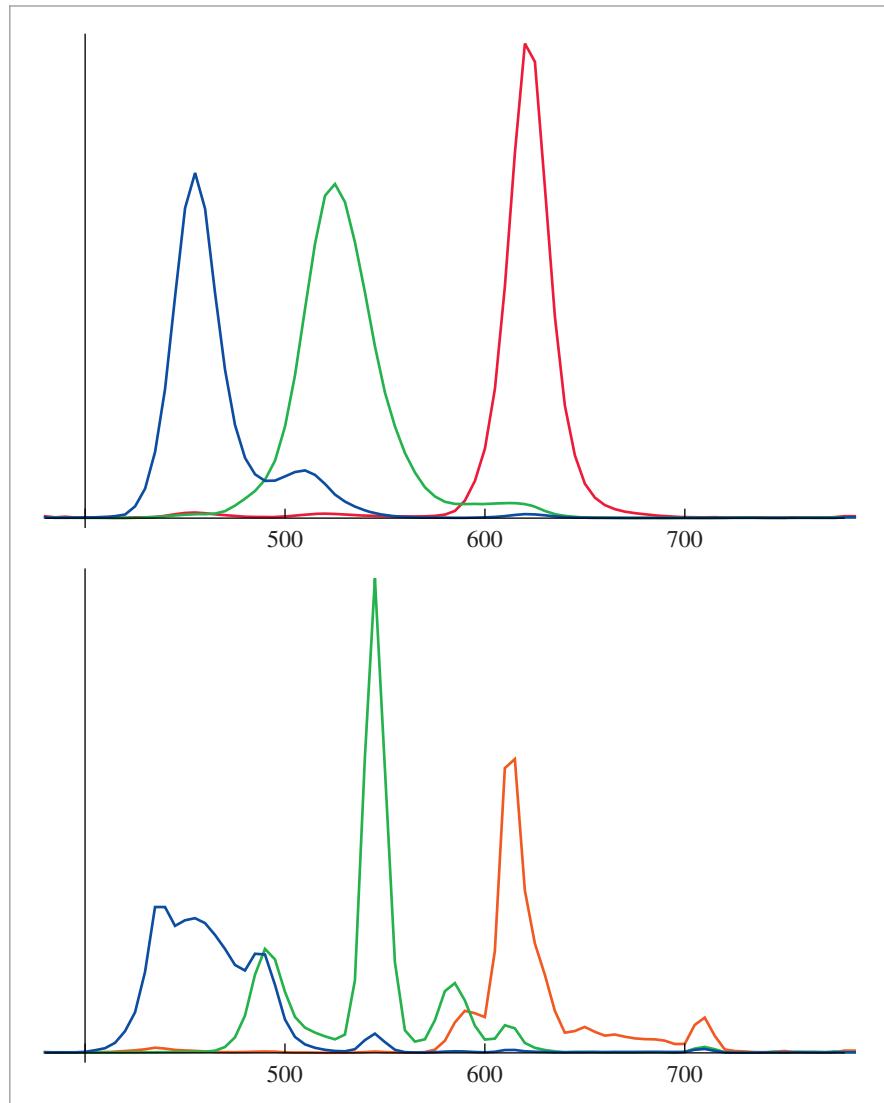
## 5.2.2 RGB COLOR

When we display an RGB color on a display, the spectrum that is actually displayed is basically determined by the weighted sum of three spectral response curves, one for each of red, green, and blue, as emitted by the display's phosphors, LED or LCD elements, or plasma cells.<sup>3</sup> Figure 5.4 plots the red, green, and blue distributions emitted by a LED display and a LCD display; note that they are remarkably different. Figure 5.5 in turn shows the SPDs that result from displaying the RGB color (0.6, 0.3, 0.2) on those displays. Not surprisingly, the resulting SPDs are quite different as well. This example

---

CIE\_Y\_integral 325  
 CoefficientSpectrum::c 316  
 Float 1062  
 nSpectralSamples 319  
 sampledLambdaEnd 319  
 sampledLambdaStart 319  
 SampledSpectrum::Y 324

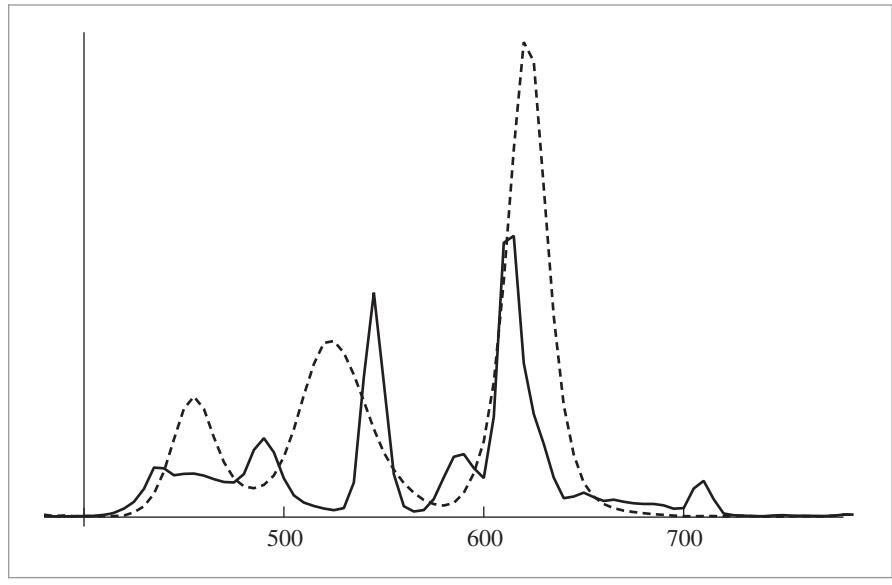
<sup>3</sup> This model is admittedly a simplification in that it neglects any additional processing the display does; in particular, many displays perform nonlinear remappings of the displayed values.



**Figure 5.4: Red, Green, and Blue Emission Curves for an LCD Display and an LED Display.** The top plot shows the curves for an LCD display, and the bottom shows them for an LED. These two displays have quite different emission profiles. (Data courtesy of X-Rite, Inc.)

illustrates that using RGB values provided by the user to describe a particular color is actually only meaningful given knowledge of the characteristics of the display they were using when they selected the RGB values.

Given an  $(x_\lambda, y_\lambda, z_\lambda)$  representation of an SPD, we can convert it to corresponding RGB coefficients, given the choice of a particular set of SPDs that define red, green, and blue for a display of interest. Given the spectral response curves  $R(\lambda)$ ,  $G(\lambda)$ , and  $B(\lambda)$ , for a



**Figure 5.5: SPDs from Displaying the RGB Color (0.6, 0.3, 0.2) on LED and LCD Displays.** The resulting emitted SPDs are remarkably different, even given the same RGB values, due to the different emission curves illustrated in Figure 5.4.

particular display, RGB coefficients can be computed by integrating the response curves with the SPD  $S(\lambda)$ :

$$\begin{aligned} r &= \int R(\lambda)S(\lambda)d\lambda = \int R(\lambda)(x_\lambda X(\lambda) + y_\lambda Y(\lambda) + z_\lambda Z(\lambda)) d\lambda \\ &= x_\lambda \int R(\lambda)X(\lambda) d\lambda + y_\lambda \int R(\lambda)Y(\lambda) d\lambda + z_\lambda \int R(\lambda)Z(\lambda) d\lambda. \end{aligned}$$

The integrals of the products of  $R(\lambda)X(\lambda)$  and so forth can be precomputed for given response curves, making it possible to express the full conversion as a matrix:

$$\begin{bmatrix} r \\ g \\ b \end{bmatrix} = \begin{pmatrix} \int R(\lambda)X(\lambda)d\lambda & \int R(\lambda)Y(\lambda)d\lambda & \int R(\lambda)Z(\lambda)d\lambda \\ \int G(\lambda)X(\lambda)d\lambda & \int G(\lambda)Y(\lambda)d\lambda & \int G(\lambda)Z(\lambda)d\lambda \\ \int B(\lambda)X(\lambda)d\lambda & \int B(\lambda)Y(\lambda)d\lambda & \int B(\lambda)Z(\lambda)d\lambda \end{pmatrix} \begin{bmatrix} x_\lambda \\ y_\lambda \\ z_\lambda \end{bmatrix}.$$

The conversion routines implemented in pbrt are based on a standard set of these RGB spectra that has been defined for high-definition television.

```

⟨Spectrum Utility Declarations⟩ +≡
    inline void XYZToRGB(const Float xyz[3], Float rgb[3]) {
        rgb[0] = 3.240479f*xyz[0] - 1.537150f*xyz[1] - 0.498535f*xyz[2];
        rgb[1] = -0.969256f*xyz[0] + 1.875991f*xyz[1] + 0.041556f*xyz[2];
        rgb[2] = 0.055648f*xyz[0] - 0.204043f*xyz[1] + 1.057311f*xyz[2];
    }

```

Float 1062

The inverse of this matrix gives coefficients to convert given RGB values expressed with respect to a particular set of RGB response curves to  $(x_\lambda, y_\lambda, z_\lambda)$  coefficients.

```
(Spectrum Utility Declarations) +≡
    inline void RGBToXYZ(const Float rgb[3], Float xyz[3]) {
        xyz[0] = 0.412453f*rgb[0] + 0.357580f*rgb[1] + 0.180423f*rgb[2];
        xyz[1] = 0.212671f*rgb[0] + 0.715160f*rgb[1] + 0.072169f*rgb[2];
        xyz[2] = 0.019334f*rgb[0] + 0.119193f*rgb[1] + 0.950227f*rgb[2];
    }
```

Given these functions, a `SampledSpectrum` can convert to RGB coefficients by first converting to XYZ and then using the `XYZToRGB()` utility function.

```
(SampledSpectrum Public Methods) +≡ 319
    void ToRGB(Float rgb[3]) const {
        Float xyz[3];
        ToXYZ(xyz);
        XYZToRGB(xyz, rgb);
    }
```

An `RGBSpectrum` can also be created easily, using the `ToRGB()` method.

```
(SampledSpectrum Public Methods) +≡ 319
    RGBSpectrum ToRGBSpectrum() const;
```

Going the other way and converting from RGB or XYZ values to a SPD isn't as easy: the problem is highly under-constrained. Recall that an infinite number of different SPDs have the same  $(x_\lambda, y_\lambda, z_\lambda)$  (and thus RGB) coefficients. Thus, given an RGB or  $(x_\lambda, y_\lambda, z_\lambda)$  value, there are an infinite number of possible SPDs that could be chosen for it. There are a number of desirable criteria that we'd like a conversion function to have:

- If all of the RGB coefficients have the same value, the resulting SPD should be constant.
- In general, it's desirable that the computed SPD be smooth. Most real-world objects have relatively smooth spectra. (The main source of spiky spectra is light sources, especially fluorescents. Fortunately, actual spectral data are more commonly available for illuminants than they are for reflectances.)

The smoothness goal is one of the reasons why constructing an SPD as a weighted sum of a display's  $R(\lambda)$ ,  $G(\lambda)$ , and  $B(\lambda)$  SPDs is not a good solution: as shown in Figure 5.4, those functions are generally irregular and spiky, and a weighted sum of them will thus not be a very smooth SPD. Although the result will be a metamer of the given RGB values, it's likely not an accurate representation of the SPD of the actual object.

Here we implement a method for converting RGBs to SPDs suggested by Smits (1999) that tries to achieve the goals above. This approach is based on the observation that a good start is to compute individual SPDs for red, green, and blue that are smooth and such that computing the weighted sum of them with the given RGB coefficients and then converting back to RGB give a result that is close to the original RGB coefficients. He found such spectra through a numerical optimization procedure.

Float [1062](#)  
RGBSpectrum [332](#)  
SampledSpectrum [319](#)  
SampledSpectrum::ToXYZ() [325](#)  
XYZToRGB() [327](#)

Smits observed that two additional improvements could be made to this basic approach. First, rather than representing constant spectra by the sums of the computed red, green, and blue SPDs, the sum of which isn't exactly constant, it's better to represent constant spectra with constant SPDs. Second, mixtures of colors like yellow (a mixture of red and green) that are a mixture of two of the primaries are better represented by their own precomputed smooth SPDs rather than the sum of SPDs for the two corresponding primaries.

The following arrays store SPDs that meet these criteria, with their samples' wavelengths in `RGB2SpectLambda[]` (these data were generated by Karl vom Berge).

```
(Spectral Data Declarations) +≡
    static const int nRGB2SpectSamples = 32;
    extern const Float RGB2SpectLambda[nRGB2SpectSamples];
    extern const Float RGBRef12SpectWhite[nRGB2SpectSamples];
    extern const Float RGBRef12SpectCyan[nRGB2SpectSamples];
    extern const Float RGBRef12SpectMagenta[nRGB2SpectSamples];
    extern const Float RGBRef12SpectYellow[nRGB2SpectSamples];
    extern const Float RGBRef12SpectRed[nRGB2SpectSamples];
    extern const Float RGBRef12SpectGreen[nRGB2SpectSamples];
    extern const Float RGBRef12SpectBlue[nRGB2SpectSamples];
```

If a given RGB color describes illumination from a light source, better results are achieved if the conversion tables are computed using the spectral power distribution of a representative illumination source to define “white” rather than using a constant spectrum as they are for the tables above that are used for reflectances. The `RGBIllum2Spect` arrays use the D65 spectral power distribution, which has been standardized by the CIE to represent midday sunlight. (The D65 illuminant will be discussed more in Section 12.1.2.)

```
(Spectral Data Declarations) +≡
    extern const Float RGBIllum2SpectWhite[nRGB2SpectSamples];
    extern const Float RGBIllum2SpectCyan[nRGB2SpectSamples];
    extern const Float RGBIllum2SpectMagenta[nRGB2SpectSamples];
    extern const Float RGBIllum2SpectYellow[nRGB2SpectSamples];
    extern const Float RGBIllum2SpectRed[nRGB2SpectSamples];
    extern const Float RGBIllum2SpectGreen[nRGB2SpectSamples];
    extern const Float RGBIllum2SpectBlue[nRGB2SpectSamples];
```

The fragment `<Compute RGB to spectrum functions for SampledSpectrum>`, which is called from `SampledSpectrum::Init()`, isn't included here; it initializes the following `SampledSpectrum` values by resampling the `RGBRef12Spect` and `RGBIllum2Spect` distributions using the `AverageSpectrumSamples()` function.

`AverageSpectrumSamples()` 321  
`Float` 1062  
`nRGB2SpectSamples` 329  
`SampledSpectrum` 319  
`SampledSpectrum::Init()` 324

```
(SampledSpectrum Private Data) +≡
    static SampledSpectrum rgbRef12SpectWhite, rgbRef12SpectCyan;
    static SampledSpectrum rgbRef12SpectMagenta, rgbRef12SpectYellow;
    static SampledSpectrum rgbRef12SpectRed, rgbRef12SpectGreen;
    static SampledSpectrum rgbRef12SpectBlue;
```

```
(SampledSpectrum Private Data) +≡ 319
  static SampledSpectrum rgbillum2SpectWhite, rgbillum2SpectCyan;
  static SampledSpectrum rgbillum2SpectMagenta, rgbillum2SpectYellow;
  static SampledSpectrum rgbillum2SpectRed, rgbillum2SpectGreen;
  static SampledSpectrum rgbillum2SpectBlue;
```

The `SampledSpectrum::FromRGB()` method converts from the given RGB values to a full SPD. In addition to the RGB values, it takes an enumeration value that denotes whether the RGB value represents surface reflectance or an illuminant; the corresponding `rgbillum2Spect` or `rgbRef12Spect` values are used for the conversion.

```
(Spectrum Utility Declarations) +≡
enum class SpectrumType { Reflectance, Illuminant };

(Spectrum Method Definitions) +≡
SampledSpectrum SampledSpectrum::FromRGB(const Float rgb[3],
                                           SpectrumType type) {
    SampledSpectrum r;
    if (type == SpectrumType::Reflectance) {
        <Convert reflectance spectrum to RGB 330
    } else {
        <Convert illuminant spectrum to RGB>
    }
    return r.Clamp();
}
```

Here we'll show the conversion process for reflectances. The computation for illuminants is the same, just using the different conversion values. First, the implementation determines whether the red, green, or blue channel is the smallest.

```
(Convert reflectance spectrum to RGB) ≡ 330
if (rgb[0] <= rgb[1] && rgb[0] <= rgb[2]) {
    (Compute reflectance SampledSpectrum with rgb[0] as minimum 331)
} else if (rgb[1] <= rgb[0] && rgb[1] <= rgb[2]) {
    (Compute reflectance SampledSpectrum with rgb[1] as minimum)
} else {
    (Compute reflectance SampledSpectrum with rgb[2] as minimum)
}
```

Here is the code for the case of a red component being the smallest. (The cases for green and blue are analogous and not included in the book here.) If red is the smallest, we know that green and blue have greater values than red. As such, we can start to convert the final SPD to return by assigning to it the value of the red component times the white spectrum in `rgbRef12SpectWhite`. Having done this, the remaining RGB value left to process is  $(0, g - r, b - r)$ . The code in turn determines which of the remaining two components is the smallest. This value, times the cyan (green and blue) spectrum, is added to the result and we're left with either  $(0, g - b, 0)$  or  $(0, 0, b - g)$ . Based on whether the green or blue channel is non-zero, the green or blue SPD is scaled by the remainder and the conversion is complete.

Float 1062  
 SampledSpectrum 319  
`Spectrum::Clamp()` 317  
 SpectrumType 330  
`SpectrumType::Reflectance`  
 330

```
<Compute reflectance SampledSpectrum with rgb[0] as minimum> ≡ 330
    r += rgb[0] * rgbRef12SpectWhite;
    if (rgb[1] <= rgb[2]) {
        r += (rgb[1] - rgb[0]) * rgbRef12SpectCyan;
        r += (rgb[2] - rgb[1]) * rgbRef12SpectBlue;
    } else {
        r += (rgb[2] - rgb[0]) * rgbRef12SpectCyan;
        r += (rgb[1] - rgb[2]) * rgbRef12SpectGreen;
    }
```

Given the method to convert from RGB, converting from XYZ color is easy. We first convert from XYZ to RGB and then use the `FromRGB()` method.

```
<SampledSpectrum Public Methods> +≡ 319
    static SampledSpectrum FromXYZ(const Float xyz[3],
                                    SpectrumType type = SpectrumType::Reflectance) {
        Float rgb[3];
        XYZToRGB(xyz, rgb);
        return FromRGB(rgb, type);
    }
```

Finally, we provide a constructor that converts from an instance of the `RGBSpectrum` class, again using the infrastructure above.

```
<Spectrum Method Definitions> +≡
    SampledSpectrum::SampledSpectrum(const RGBSpectrum &r, SpectrumType t) {
        Float rgb[3];
        r.ToRGB(rgb);
        *this = SampledSpectrum::FromRGB(rgb, t);
    }
```

CoefficientSpectrum 316  
 Float 1062  
 RGBSpectrum 332  
 RGBSpectrum::ToRGB() 332  
 SampledSpectrum 319  
 SampledSpectrum::FromRGB()  
 330  
 SampledSpectrum::  
 rgbRef12SpectBlue  
 329  
 SampledSpectrum::  
 rgbRef12SpectCyan  
 329  
 SampledSpectrum::  
 rgbRef12SpectGreen  
 329  
 SampledSpectrum::  
 rgbRef12SpectWhite  
 329  
 SpectrumType 330  
 XYZToRGB() 327

## 5.3 RGBSpectrum IMPLEMENTATION

The `RGBSpectrum` implementation here represents SPDs with a weighted sum of red, green, and blue components. Recall that this representation is ill defined: given two different computer displays, having them display the same RGB value won't cause them to emit the same SPD. Thus, in order for a set of RGB values to specify an actual SPD, we must know the monitor primaries that they are defined in terms of; this information is generally not provided along with RGB values.

The RGB representation is nevertheless convenient: almost all 3D modeling and design tools use RGB colors, and most 3D content is specified in terms of RGB. Furthermore, it's computationally and storage efficient, requiring just three floating-point values to represent. Our implementation of `RGBSpectrum` inherits from `CoefficientSpectrum`, specifying three components to store. Thus, all of the arithmetic operations defined earlier are automatically available for the `RGBSpectrum`.

```
(Spectrum Declarations) +≡
  class RGBSpectrum : public CoefficientSpectrum<3> {
    public:
      (RGBSpectrum Public Methods 332}
  };
```

*(RGBSpectrum Public Methods) ≡*

```
RGBSpectrum(Float v = 0.f) : CoefficientSpectrum<3>(v) { }
RGBSpectrum(const CoefficientSpectrum<3> &v)
  : CoefficientSpectrum<3>(v) { }
```

332

Beyond the basic arithmetic operators, the `RGBSpectrum` needs to provide methods to convert to and from `XYZ` and `RGB` representations. For the `RGBSpectrum` these are trivial. Note that `FromRGB()` takes a `SpectrumType` parameter like the `SampledSpectrum` instance of this method. Although it's unused here, the `FromRGB()` methods of these two classes must have matching signatures so that the rest of the system can call them consistently regardless of which spectral representation is being used.

*(RGBSpectrum Public Methods) +≡*

```
static RGBSpectrum FromRGB(const Float rgb[3],
                           SpectrumType type = SpectrumType::Reflectance) {
  RGBSpectrum s;
  s.c[0] = rgb[0];
  s.c[1] = rgb[1];
  s.c[2] = rgb[2];
  return s;
}
```

332

Similarly, spectrum representations must be able to convert themselves to `RGB` values. For the `RGBSpectrum`, the implementation can sidestep the question of what particular `RGB` primaries are used to represent the spectral distribution and just return the `RGB` coefficients directly, assuming that the primaries are the same as the ones already being used to represent the color.

*(RGBSpectrum Public Methods) +≡*

```
void ToRGB(Float *rgb) const {
  rgb[0] = c[0];
  rgb[1] = c[1];
  rgb[2] = c[2];
}
```

332

All spectrum representations must also be able to convert themselves to an `RGBSpectrum` object. This is again trivial here.

*(RGBSpectrum Public Methods) +≡*

```
const RGBSpectrum &ToRGBSpectrum() const {
  return *this;
}
```

`CoefficientSpectrum` 316  
`CoefficientSpectrum::c` 316  
`Float` 1062  
`RGBSpectrum` 332  
`SpectrumType` 330

332

The implementations of the `RGBSpectrum::ToXYZ()`, `RGBSpectrum::FromXYZ()`, and `RGBSpectrum::y()` methods are based on the `RGBToXYZ()` and `XYZToRGB()` functions defined above and are not included here.

To create an RGB spectrum from an arbitrary sampled SPD, `FromSampled()` converts the spectrum to XYZ and then to RGB. It evaluates the piecewise linear sampled spectrum at 1-nm steps, using the `InterpolateSpectrumSamples()` utility function, at each of the wavelengths where there is a value for the CIE matching functions. It then uses this value to compute the Riemann sum to approximate the XYZ integrals.

*(RGBSpectrum Public Methods) +≡*

```
static RGBSpectrum FromSampled(const Float *lambda, const Float *v,
                                 int n) {
    <Sort samples if unordered, use sorted for returned spectrum 320>
    Float xyz[3] = { 0, 0, 0 };
    for (int i = 0; i < nCIESamples; ++i) {
        Float val = InterpolateSpectrumSamples(lambda, v, n,
                                                CIE_lambda[i]);
        xyz[0] += val * CIE_X[i];
        xyz[1] += val * CIE_Y[i];
        xyz[2] += val * CIE_Z[i];
    }
    Float scale = Float(CIE_lambda[nCIESamples-1] - CIE_lambda[0]) /
                  Float(CIE_Y_integral * nCIESamples);
    xyz[0] *= scale;
    xyz[1] *= scale;
    xyz[2] *= scale;
    return FromXYZ(xyz);
}
```

332

`CIE_lambda` 323  
`CIE_X` 323  
`CIE_Y` 323  
`CIE_Y_integral` 325  
`CIE_Z` 323  
`FindInterval()` 1065  
`Float` 1062  
`InterpolateSpectrumSamples()` 333  
`Lerp()` 1079  
`nCIESamples` 323  
`RGBSpectrum` 332  
`RGBSpectrum::FromXYZ()` 333  
`RGBToXYZ()` 328  
`XYZToRGB()` 327

`InterpolateSpectrumSamples()` takes a possibly irregularly sampled set of wavelengths and SPD values ( $\lambda_i, v_i$ ) and returns the value of the SPD at the given wavelength  $\lambda$ , linearly interpolating between the two sample values that bracket  $\lambda$ . The `FindInterval()` function defined in Appendix A performs a binary search through the sorted wavelength array `lambda` to find the interval containing  $\lambda$ .

*(Spectrum Method Definitions) +≡*

```
Float InterpolateSpectrumSamples(const Float *lambda, const Float *vals,
                                  int n, Float l) {
    if (l <= lambda[0])      return vals[0];
    if (l >= lambda[n - 1]) return vals[n - 1];
    int offset = FindInterval(n,
                               [&](int index) { return lambda[index] <= l; });
    Float t = (l - lambda[offset]) / (lambda[offset+1] - lambda[offset]);
    return Lerp(t, vals[offset], vals[offset + 1]);
}
```

## 5.4 RADIOMETRY

Radiometry provides a set of ideas and mathematical tools to describe light propagation and reflection. It forms the basis of the derivation of the rendering algorithms that will be used throughout the rest of this book. Interestingly enough, radiometry wasn't originally derived from first principles using the physics of light but was built on an abstraction of light based on particles flowing through space. As such, effects like polarization of light do not naturally fit into this framework, although connections have since been made between radiometry and Maxwell's equations, giving radiometry a solid basis in physics.

*Radiative transfer* is the phenomenological study of the transfer of radiant energy. It is based on radiometric principles and operates at the *geometric optics* level, where macroscopic properties of light suffice to describe how light interacts with objects much larger than the light's wavelength. It is not uncommon to incorporate phenomena from wave optics models of light, but these results need to be expressed in the language of radiative transfer's basic abstractions.<sup>4</sup> In this manner, it is possible to describe interactions of light with objects of approximately the same size as the wavelength of the light, and thereby model effects like dispersion and interference. At an even finer level of detail, quantum mechanics is needed to describe light's interaction with atoms. Fortunately, direct simulation of quantum mechanical principles is unnecessary for solving rendering problems in computer graphics, so the intractability of such an approach is avoided.

In pbrt, we will assume that geometric optics is an adequate model for the description of light and light scattering. This leads to a few basic assumptions about the behavior of light that will be used implicitly throughout the system:

- *Linearity*: The combined effect of two inputs to an optical system is always equal to the sum of the effects of each of the inputs individually.
- *Energy conservation*: When light scatters from a surface or from participating media, the scattering events can never produce more energy than they started with.
- *No polarization*: We will ignore polarization of the electromagnetic field; therefore, the only relevant property of light is its distribution by wavelength (or, equivalently, frequency).
- *No fluorescence or phosphorescence*: The behavior of light at one wavelength is completely independent of light's behavior at other wavelengths or times. As with polarization, it is not too difficult to include these effects, but they would add relatively little practical value to the system.
- *Steady state*: Light in the environment is assumed to have reached equilibrium, so its radiance distribution isn't changing over time. This happens nearly instantaneously with light in realistic scenes, so it is not a limitation in practice. Note that phosphorescence also violates the steady-state assumption.

The most significant loss from adopting a geometric optics model is that diffraction and interference effects cannot easily be accounted for. As noted by Preisendorfer (1965), this

---

<sup>4</sup> Preisendorfer (1965) has connected radiative transfer theory to Maxwell's classical equations describing electromagnetic fields. His framework both demonstrates their equivalence and makes it easier to apply results from one worldview to the other. More recent work was done in this area by Fante (1981).

is a hard problem to fix because, for example, the total flux over two areas isn't necessarily equal to the sum of the power received over each individual area in the presence of those effects (p. 24).

### 5.4.1 BASIC QUANTITIES

There are four radiometric quantities that are central to rendering: flux, irradiance/radiant exitance, intensity, and radiance. They can each be derived from energy (measured in joules) by successively taking limits over time, area, and directions. All of these radiometric quantities are in general wavelength dependent. For the remainder of this chapter, we will not make this dependence explicit, but this property is important to keep in mind.

#### Energy

Our starting point is energy, which is measured in joules (J). Sources of illumination emit photons, each of which is at a particular wavelength and carries a particular amount of energy. All of the basic radiometric quantities are effectively different ways of measuring photons. A photon at wavelength  $\lambda$  carries energy

$$Q = \frac{hc}{\lambda},$$

where  $c$  is the speed of light, 299,472,458 m/s, and  $h$  is Planck's constant,  $h \approx 6.626 \times 10^{-34} \text{ m}^2 \text{ kg/s}$ .

#### Flux

Energy measures work over some period of time, though under the steady-state assumption generally used in rendering, we're mostly interested in measuring light at an instant. *Radiant flux*, also known as *power*, is the total amount of energy passing through a surface or region of space per unit time. Radiant flux can be found by taking the limit of differential energy per differential time:

$$\Phi = \lim_{\Delta t \rightarrow 0} \frac{\Delta Q}{\Delta t} = \frac{dQ}{dt}.$$

Its units are joules/second (J/s), or more commonly, watts (W).

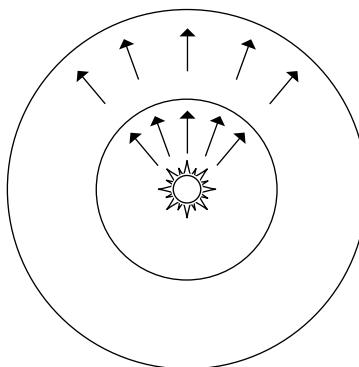
For example, given a light that emitted  $Q = 200,000 \text{ J}$  over the course of an hour, if the same amount of energy was emitted at all times over the hour, we can find that the light source's flux was

$$\Phi = 200,000 \text{ J}/3600 \text{ s} \approx 55.6 \text{ W}.$$

Conversely, given flux as a function of time, we can integrate over a range of times to compute the total energy:

$$Q = \int_{t_0}^{t_1} \Phi(t) dt.$$

Note that our notation here is slightly informal: among other issues, because photons are actually discrete quanta means that it's not really meaningful to take limits that go to zero for differential time. For the purposes of rendering, where the number of



**Figure 5.6:** Radiant flux,  $\Phi$ , measures energy passing through a surface or region of space. Here, flux from a point light source is measured at spheres that surround the light.

photons is enormous with respect to the measurements we're interested in, this detail isn't problematic in practice.

Total emission from light sources is generally described in terms of flux. Figure 5.6 shows flux from a point light source measured by the total amount of energy passing through imaginary spheres around the light. Note that the total amount of flux measured on either of the two spheres in Figure 5.6 is the same—although less energy is passing through any local part of the large sphere than the small sphere, the greater area of the large sphere means that the total flux is the same.

### Irradiance and Radiant Exitance

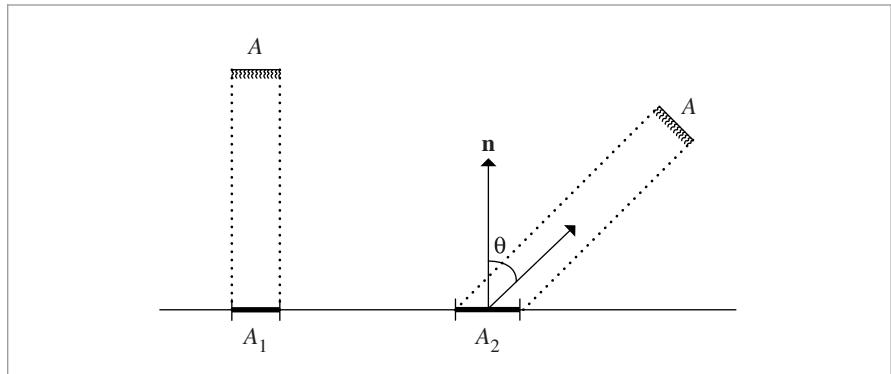
Any measurement of flux requires an area over which photons per time is being measured. Given a finite area  $A$ , we can define the average density of power over the area by  $E = \Phi/A$ . This quantity is either *irradiance* ( $E$ ), the area density of flux arriving at a surface, or *radiant exitance* ( $M$ ), the area density of flux leaving a surface. These measurements have units of  $\text{W}/\text{m}^2$ . (The term *irradiance* is sometimes also used to refer to flux leaving a surface, but for clarity we'll use different terms for the two cases.)

For the point light source example in Figure 5.6, irradiance at a point on the outer sphere is less than the irradiance at a point on the inner sphere, since the surface area of the outer sphere is larger. In particular, if the point source is illuminating the same amount of illumination in all directions, then for a sphere in this configuration that has radius  $r$ ,

$$E = \frac{\Phi}{4\pi r^2}.$$

This fact explains why the amount of energy received from a light at a point falls off with the squared distance from the light.

More generally, we can define irradiance and radiant exitance by taking the limit of differential power per differential area at a point  $p$ :



**Figure 5.7: Lambert’s Law.** Irradiance arriving at a surface varies according to the cosine of the angle of incidence of illumination, since illumination is over a larger area at larger incident angles.

$$E(p) = \lim_{\Delta A \rightarrow 0} \frac{\Delta\Phi(p)}{\Delta A} = \frac{d\Phi(p)}{dA}.$$

We can also integrate irradiance over an area to find power:

$$\Phi = \int_A E(p) dA.$$

The irradiance equation can also help us understand the origin of *Lambert’s law*, which says that the amount of light energy arriving at a surface is proportional to the cosine of the angle between the light direction and the surface normal (Figure 5.7). Consider a light source with area  $A$  and flux  $\Phi$  that is illuminating a surface. If the light is shining directly down on the surface (as on the left side of the figure), then the area on the surface receiving light  $A_1$  is equal to  $A$ . Irradiance at any point inside  $A_1$  is then

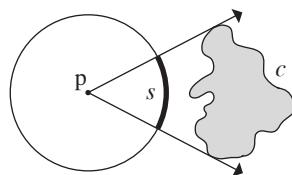
$$E_1 = \frac{\Phi}{A}.$$

However, if the light is at an angle to the surface, the area on the surface receiving light is larger. If  $A$  is small, then the area receiving flux,  $A_2$ , is roughly  $A/\cos \theta$ . For points inside  $A_2$ , the irradiance is therefore

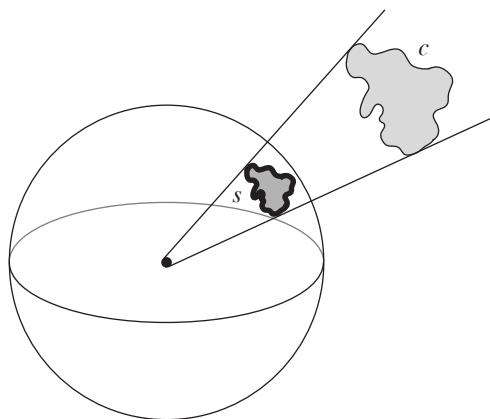
$$E_2 = \frac{\Phi \cos \theta}{A}.$$

### Solid Angle and Intensity

In order to define intensity, we first need to define the notion of a *solid angle*. Solid angles are just the extension of 2D angles in a plane to an angle on a sphere. The *planar angle* is the total angle subtended by some object with respect to some position (Figure 5.8). Consider the unit circle around the point  $p$ ; if we project the shaded object onto that circle, some length of the circle  $s$  will be covered by its projection. The arc length of  $s$  (which is the same as the angle  $\theta$ ) is the angle subtended by the object. Planar angles are measured in *radians*.



**Figure 5.8: Planar Angle.** The planar angle of an object  $c$  as seen from a point  $p$  is equal to the angle it subtends as seen from  $p$  or, equivalently, as the length of the arc  $s$  on the unit sphere.



**Figure 5.9: Solid Angle.** The solid angle  $s$  subtended by an object  $c$  in three dimensions is computed by projecting  $c$  onto the unit sphere and measuring the area of its projection.

The solid angle extends the 2D unit circle to a 3D unit sphere (Figure 5.9). The total area  $s$  is the solid angle subtended by the object. Solid angles are measured in *steradians* (sr). The entire sphere subtends a solid angle of  $4\pi$  sr, and a hemisphere subtends  $2\pi$  sr.

The set of points on the unit sphere centered at a point  $p$  can be used to describe the vectors anchored at  $p$ . We will usually use the symbol  $\omega$  to indicate these directions, and we will use the convention that  $\omega$  is a normalized vector.

Consider now an infinitesimal light source emitting photons. If we center this light source within the unit sphere, we can compute the angular density of emitted power. *Intensity*, denoted by  $I$ , is this quantity; it has units W/sr. Over the entire sphere of directions, we have

$$I = \frac{\Phi}{4\pi},$$

but more generally we're interested in taking the limit of a differential cone of directions:

$$I = \lim_{\Delta\omega \rightarrow 0} \frac{\Delta\Phi}{\Delta\omega} = \frac{d\Phi}{d\omega}.$$

As usual, we can go back to power by integrating intensity: given intensity as a function of direction  $I(\omega)$ , we can integrate over a finite set of directions  $\Omega$  to recover the intensity:

$$\Phi = \int_{\Omega} I(\omega) d\omega.$$

Intensity describes the directional distribution of light, but it is only meaningful for point light sources.

### Radiance

The final, and most important, radiometric quantity is *radiance*,  $L$ . Irradiance and radiant exitance give us differential power per differential area at a point  $p$ , but they don't distinguish the directional distribution of power. Radiance takes this last step and measures irradiance or radiant exitance with respect to solid angles. It is defined by

$$L(p, \omega) = \lim_{\Delta\omega \rightarrow 0} \frac{\Delta E_{\omega}(p)}{\Delta\omega} = \frac{dE_{\omega}(p)}{d\omega},$$

where we have used  $E_{\omega}$  to denote irradiance at the surface that is perpendicular to the direction  $\omega$ . In other words, radiance is not measured with respect to the irradiance incident at the surface  $p$  lies on. In effect, this change of measurement area serves to eliminate the  $\cos\theta$  term from Lambert's law in the definition of radiance.

Radiance is the flux density per unit area, per unit solid angle. In terms of flux, it is defined by

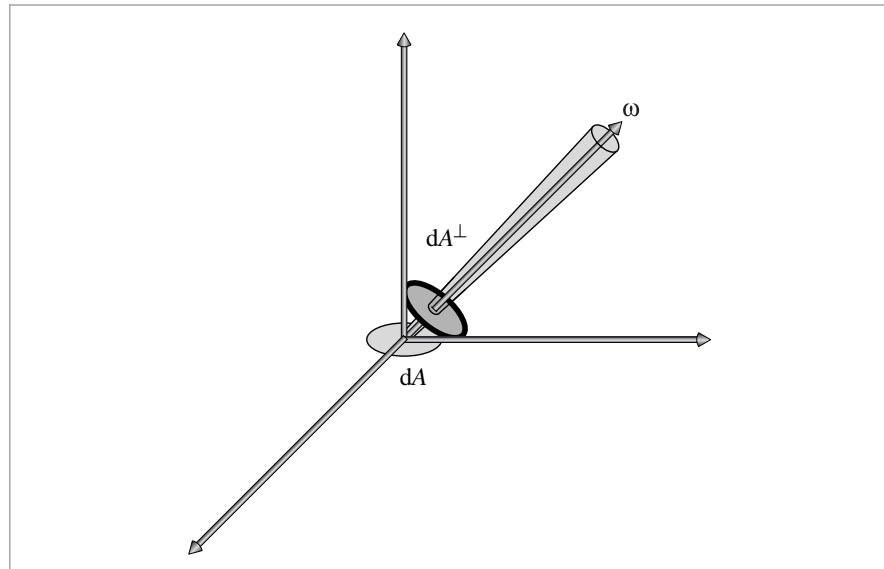
$$L = \frac{d\Phi}{d\omega dA^{\perp}}, \quad [5.2]$$

where  $dA^{\perp}$  is the projected area of  $dA$  on a hypothetical surface perpendicular to  $\omega$  (Figure 5.10). Thus, it is the limit of the measurement of incident light at the surface as a cone of incident directions of interest  $d\omega$  becomes very small and as the local area of interest on the surface  $dA$  also becomes very small.

Of all of these radiometric quantities, radiance will be the one used most frequently throughout the rest of the book. An intuitive reason for this is that in some sense it's the most fundamental of all the radiometric quantities; if radiance is given, then all of the other values can be computed in terms of integrals of radiance over areas and directions. Another nice property of radiance is that it remains constant along rays through empty space. It is thus a natural quantity to compute with ray tracing.

#### 5.4.2 INCIDENT AND EXITANT RADIANCE FUNCTIONS

When light interacts with surfaces in the scene, the radiance function  $L$  is generally not continuous across the surface boundaries. In the most extreme case of a fully opaque surface (e.g., a mirror), the radiance function slightly above and slightly below a surface could be completely unrelated.



**Figure 5.10:** Radiance  $L$  is defined as flux per unit solid angle  $d\omega$  per unit projected area  $dA^\perp$ .

It therefore makes sense to take one-sided limits at the discontinuity to distinguish between the radiance function just above and below

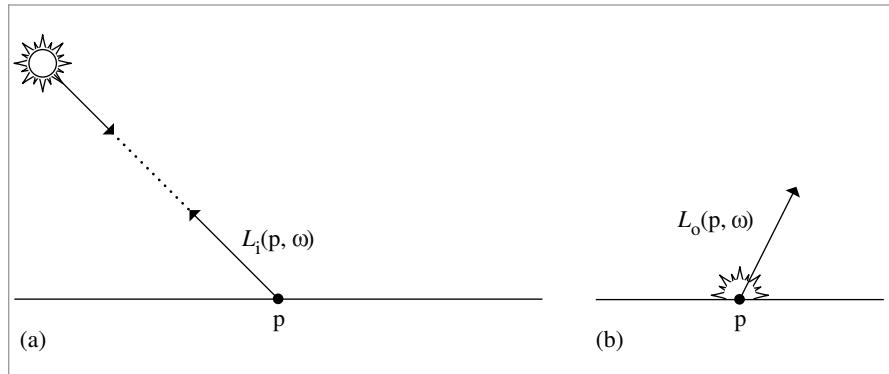
$$\begin{aligned} L^+(p, \omega) &= \lim_{t \rightarrow 0^+} L(p + t\mathbf{n}_p, \omega), \\ L^-(p, \omega) &= \lim_{t \rightarrow 0^-} L(p + t\mathbf{n}_p, \omega), \end{aligned} \quad [5.3]$$

where  $\mathbf{n}_p$  is the surface normal at  $p$ . However, keeping track of one-sided limits throughout the text is unnecessarily cumbersome.

We prefer to solve this ambiguity by making a distinction between radiance arriving at the point (e.g., due to illumination from a light source) and radiance leaving that point (e.g., due to reflection from a surface).

Consider a point  $p$  on the surface of an object. There is some distribution of radiance arriving at the point that can be described mathematically by a function of position and direction. This function is denoted by  $L_i(p, \omega)$  (Figure 5.11). The function that describes the outgoing reflected radiance from the surface at that point is denoted by  $L_o(p, \omega)$ . Note that in both cases the direction vector  $\omega$  is oriented to point away from  $p$ , but be aware that some authors use a notation where  $\omega$  is reversed for  $L_i$  terms so that it points toward  $p$ .

There is a simple relation between these more intuitive incident and exitant radiance functions and the one-sided limits from Equation (5.3):



**Figure 5.11:** (a) The incident radiance function  $L_i(p, \omega)$  describes the distribution of radiance arriving at a point as a function of position and direction. (b) The exitant radiance function  $L_o(p, \omega)$  gives the distribution of radiance leaving the point. Note that for both functions,  $\omega$  is oriented to point away from the surface, and, thus, for example,  $L_i(p, -\omega)$  gives the radiance arriving on the other side of the surface than the one where  $\omega$  lies.

$$L_i(p, \omega) = \begin{cases} L^+(p, -\omega), & \omega \cdot n_p > 0 \\ L^-(p, -\omega), & \omega \cdot n_p < 0 \end{cases}$$

$$L_o(p, \omega) = \begin{cases} L^+(p, \omega), & \omega \cdot n_p > 0 \\ L^-(p, \omega), & \omega \cdot n_p < 0 \end{cases}$$

Throughout the book, we will use the idea of incident and exitant radiance functions to resolve ambiguity in the radiance function at boundaries.

Another property to keep in mind is that at a point in space where there is no surface (i.e. in free space),  $L$  is continuous, so  $L^+ = L^-$ , which means

$$L_o(p, \omega) = L_i(p, -\omega) = L(p, \omega).$$

In other words,  $L_i$  and  $L_o$  only differ by a direction reversal.

### 5.4.3 LUMINANCE AND PHOTOMETRY

All of the radiometric measurements like flux, radiance, and so forth have corresponding photometric measurements. *Photometry* is the study of visible electromagnetic radiation in terms of its perception by the human visual system. Each spectral radiometric quantity can be converted to its corresponding photometric quantity by integrating against the spectral response curve  $V(\lambda)$ , which describes the relative sensitivity of the human eye to various wavelengths.<sup>5</sup>

*Luminance* measures how bright a spectral power distribution appears to a human observer. For example, luminance accounts for the fact that an SPD with a particular

<sup>5</sup> The spectral response curve model is based on experiments done in a normally illuminated indoor environment. Because sensitivity to color decreases in dark environments, it doesn't model the human visual system's response well under all lighting situations. Nonetheless, it forms the basis for the definition of luminance and other related photometric properties.

**Table 5.1: Representative Luminance Values for a Number of Lighting Conditions.**

Condition	Luminance (cd/m <sup>2</sup> , or nits)
Sun at horizon	600,000
60-watt lightbulb	120,000
Clear sky	8,000
Typical office	100–1,000
Typical computer display	1–100
Street lighting	1–10
Cloudy moonlight	0.25

**Table 5.2: Radiometric Measurements and Their Photometric Analogs.**

Radiometric	Unit	Photometric	Unit
Radiant energy	joule (J)	Luminous energy	talbot (T)
Radiant flux	watt (W)	Luminous flux	lumen (lm)
Intensity	W/sr	Luminous intensity	lm/sr = candela (cd)
Irradiance	W/m <sup>2</sup>	Illuminance	lm/m <sup>2</sup> = lux (lx)
Radiance	W/(m <sup>2</sup> sr)	Luminance	lm/(m <sup>2</sup> sr) = cd/m <sup>2</sup> = nit

amount of energy in the green wavelengths will appear brighter to a human than an SPD with the same amount of energy in blue.

We will denote luminance by  $Y$ ; it related to spectral radiance  $L(\lambda)$  by

$$Y = \int_{\lambda} L(\lambda) V(\lambda) d\lambda.$$

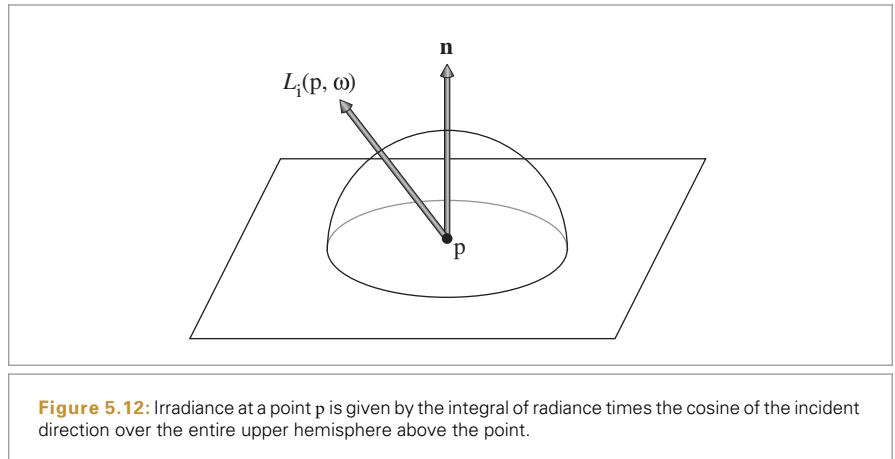
Luminance and the spectral response curve  $V(\lambda)$  are closely related to the XYZ representation of color (Section 5.2.1). The CIE  $Y(\lambda)$  tristimulus curve was chosen to be proportional to  $V(\lambda)$  so that

$$Y = 683 \int_{\lambda} L(\lambda) Y(\lambda) d\lambda.$$

The units of luminance are candelas per meter squared (cd/m<sup>2</sup>), where the candela is the photometric equivalent of radiant intensity. Some representative luminance values are given in Table 5.1.

All of the other radiometric quantities that we have introduced in this chapter have photometric equivalents; they are summarized in Table 5.2.<sup>6</sup>

<sup>6</sup> The various photometric quantities have fairly unusual names; the somewhat confusing state of affairs was nicely summarized by Jim Kajiya: “Thus one nit is one lux per steradian is one candela per square meter is one lumen per square meter per steradian. Got it?”



## 5.5 WORKING WITH RADIOMETRIC INTEGRALS

One of the most frequent tasks in rendering is the evaluation of integrals of radiometric quantities. In this section, we will present some tricks that can make this task easier. To illustrate the use of these techniques, we will use the computation of irradiance at a point as an example. Irradiance at a point  $p$  with surface normal  $n$  due to radiance over a set of directions  $\Omega$  is

$$E(p, n) = \int_{\Omega} L_i(p, \omega) |\cos \theta| d\omega, \quad [5.4]$$

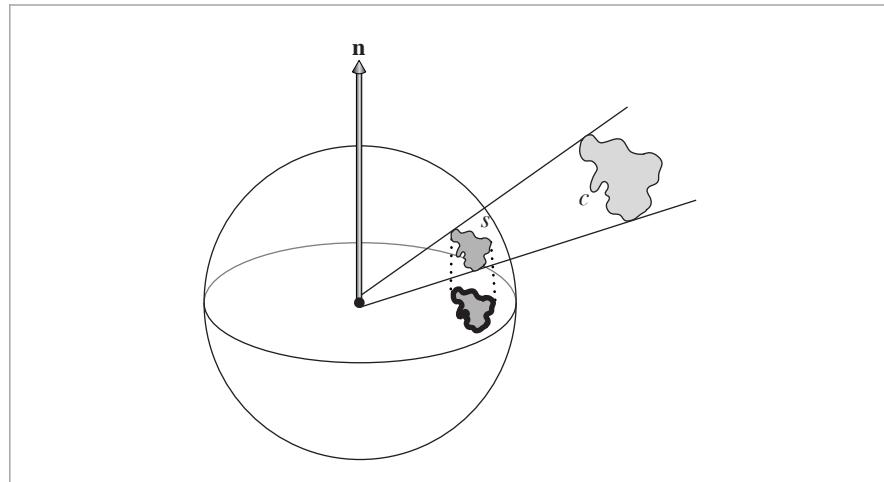
where  $L_i(p, \omega)$  is the incident radiance function (Figure 5.12) and the  $\cos \theta$  term in this integral is due to the  $dA^\perp$  term in the definition of radiance.  $\theta$  is measured as the angle between  $\omega$  and the surface normal  $n$ . Irradiance is usually computed over the hemisphere  $\mathcal{H}^2(n)$  of directions about a given surface normal  $n$ .

### 5.5.1 INTEGRALS OVER PROJECTED SOLID ANGLE

The various cosine terms in the integrals for radiometric quantities can often distract from what is being expressed in the integral. This problem can be avoided using *projected solid angle* rather than solid angle to measure areas subtended by objects being integrated over. The projected solid angle subtended by an object is determined by projecting the object onto the unit sphere, as was done for the solid angle, but then projecting the resulting shape down onto the unit disk that is perpendicular to the surface normal (Figure 5.13). Integrals over hemispheres of directions with respect to cosine-weighted solid angle can be rewritten as integrals over projected solid angle.

The projected solid angle measure is related to the solid angle measure by

$$d\omega^\perp = |\cos \theta| d\omega,$$



**Figure 5.13:** The projected solid angle subtended by an object  $c$  is the cosine-weighted solid angle that it subtends. It can be computed by finding the object's solid angle  $s$ , projecting it down to the plane perpendicular to the surface normal, and measuring its area there. Thus, the projected solid angle depends on the surface normal where it is being measured, since the normal orients the plane of projection.

so the irradiance-from-radiance integral over the hemisphere can be written more simply as

$$E(p, \mathbf{n}) = \int_{\mathcal{H}^2(\mathbf{n})} L_i(p, \omega) d\omega^\perp.$$

For the rest of this book, we will write integrals over directions in terms of solid angle, rather than projected solid angle. In other sources, however, projected solid angle may be used, and so it is always important to be aware of the integrand's actual measure.

Just as we found irradiance in terms of incident radiance, we can also compute the total flux emitted from some object over the hemisphere surrounding the normal by integrating over the object's surface area  $A$ :

$$\begin{aligned} \Phi &= \int_A \int_{\mathcal{H}^2(\mathbf{n})} L_o(p, \omega) \cos \theta d\omega dA \\ &= \int_A \int_{\mathcal{H}^2(\mathbf{n})} L_o(p, \omega) d\omega^\perp dA. \end{aligned}$$

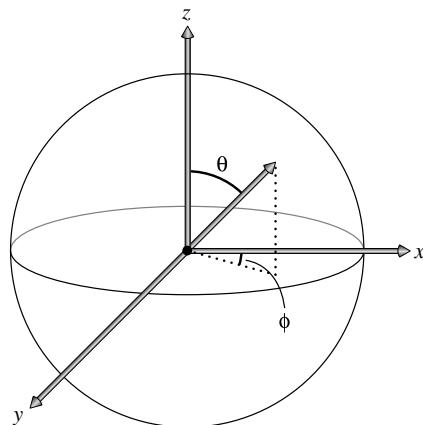
### 5.5.2 INTEGRALS OVER SPHERICAL COORDINATES

It is often convenient to transform integrals over solid angle into integrals over spherical coordinates  $(\theta, \phi)$ . Recall that an  $(x, y, z)$  direction vector can also be written in terms of spherical angles (Figure 5.14):

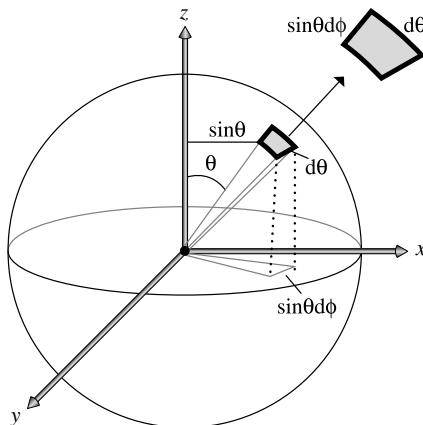
$$x = \sin \theta \cos \phi$$

$$y = \sin \theta \sin \phi$$

$$z = \cos \theta.$$



**Figure 5.14:** A direction vector can be written in terms of spherical coordinates  $(\theta, \phi)$  if the  $x$ ,  $y$ , and  $z$  basis vectors are given as well. The spherical angle formulae make it easy to convert between the two representations.



**Figure 5.15:** The differential area  $dA$  subtended by a differential solid angle is the product of the differential lengths of the two edges  $\sin \theta d\phi$  and  $d\theta$ . The resulting relationship,  $d\omega = \sin \theta d\theta d\phi$ , is the key to converting between integrals over solid angles and integrals over spherical angles.

In order to convert an integral over a solid angle to an integral over  $(\theta, \phi)$ , we need to be able to express the relationship between the differential area of a set of directions  $d\omega$  and the differential area of a  $(\theta, \phi)$  pair (Figure 5.15). The differential area  $d\omega$  is the product of the differential lengths of its sides,  $\sin \theta d\phi$  and  $d\theta$ . Therefore,

$$d\omega = \sin \theta d\theta d\phi. \quad [5.5]$$

We can thus see that the irradiance integral over the hemisphere, Equation (5.4) with  $\Omega = \mathcal{H}^2(\mathbf{n})$ , can equivalently be written as

$$E(p, \mathbf{n}) = \int_0^{2\pi} \int_0^{\pi/2} L_i(p, \theta, \phi) \cos \theta \sin \theta d\theta d\phi.$$

If the radiance is the same from all directions, the equation simplifies to  $E = \pi L_i$ .

For convenience, we'll define two functions that convert  $\theta$  and  $\phi$  values into  $(x, y, z)$  direction vectors. The first function applies the earlier equations directly. Notice that these functions are passed the sine and cosine of  $\theta$ , rather than  $\theta$  itself. This is because the sine and cosine of  $\theta$  are often already available to the caller. This is not normally the case for  $\phi$ , however, so  $\phi$  is passed in as is.

*(Geometry Inline Functions) +≡*

```
inline Vector3f SphericalDirection(Float sinTheta,
    Float cosTheta, Float phi) {
    return Vector3f(sinTheta * std::cos(phi),
        sinTheta * std::sin(phi),
        cosTheta);
}
```

The second function takes three basis vectors representing the  $x$ ,  $y$ , and  $z$  axes and returns the appropriate direction vector with respect to the coordinate frame defined by them:

*(Geometry Inline Functions) +≡*

```
inline Vector3f SphericalDirection(Float sinTheta, Float cosTheta,
    Float phi, const Vector3f &x, const Vector3f &y,
    const Vector3f &z) {
    return sinTheta * std::cos(phi) * x +
        sinTheta * std::sin(phi) * y + cosTheta * z;
}
```

The conversion of a direction  $(x, y, z)$  to spherical angles can be found by

$$\begin{aligned}\theta &= \arccos z \\ \phi &= \arctan \frac{y}{x}.\end{aligned}$$

The corresponding functions follow. Note that `SphericalTheta()` assumes that the vector  $v$  has been normalized before being passed in; the clamp is purely to avoid errors from `std::acos()` if  $|v.z|$  is slightly greater than 1 due to floating-point round-off error.

*(Geometry Inline Functions) +≡*

```
inline Float SphericalTheta(const Vector3f &v) {
    return std::acos(Clamp(v.z, -1, 1));
}
```

Float 1062

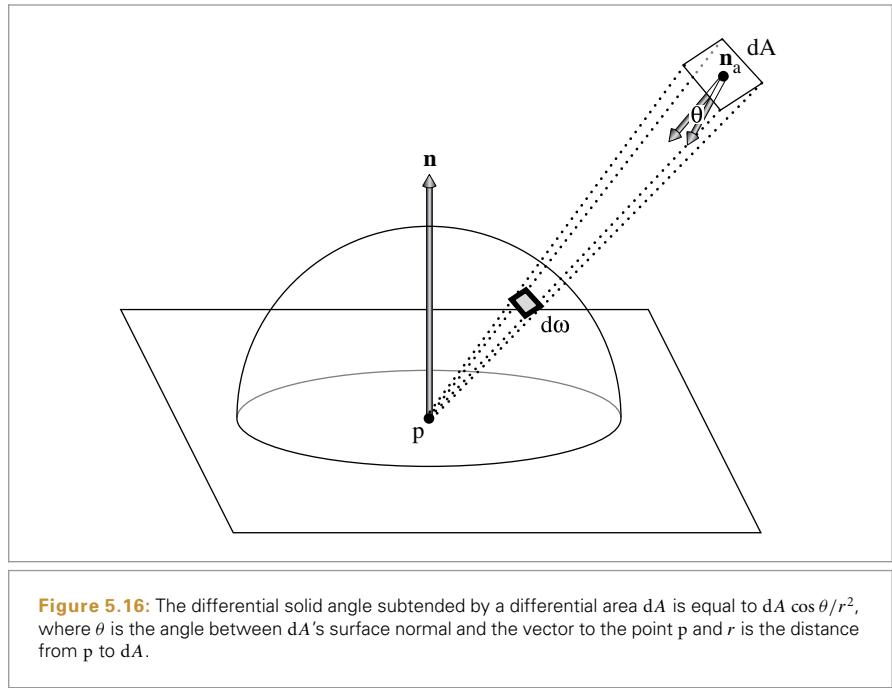
Pi 1063

SphericalTheta() 346

Vector3f 60

*(Geometry Inline Functions) +≡*

```
inline Float SphericalPhi(const Vector3f &v) {
    Float p = std::atan2(v.y, v.x);
    return (p < 0) ? (p + 2 * Pi) : p;
}
```



### 5.5.3 INTEGRALS OVER AREA

One last transformation of integrals that can simplify computation is to turn integrals over directions into integrals over area. Consider the irradiance integral in Equation (5.4) again, and imagine there is a quadrilateral with constant outgoing radiance and we'd like to compute the resulting irradiance at a point  $p$ . Computing this value as an integral over directions is not straightforward, since given a particular direction it is nontrivial to determine if the quadrilateral is visible in that direction. It's much easier to compute the irradiance as an integral over the area of the quadrilateral.

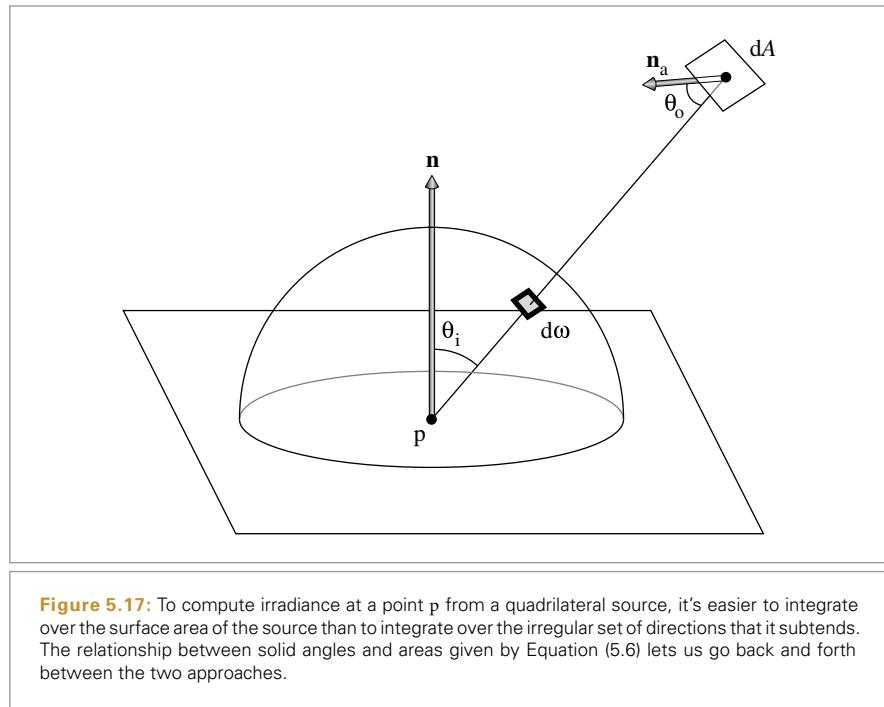
Differential area is related to differential solid angle (as viewed from a point  $p$ ) by

$$d\omega = \frac{dA \cos \theta}{r^2}, \quad [5.6]$$

where  $\theta$  is the angle between the surface normal of  $dA$  and the vector to  $p$ , and  $r$  is the distance from  $p$  to  $dA$  (Figure 5.16). We will not derive this result here, but it can be understood intuitively: if  $dA$  is at distance 1 from  $p$  and is aligned exactly so that it is perpendicular to  $d\omega$ , then  $d\omega = dA$ ,  $\theta = 0$ , and Equation (5.6) holds. As  $dA$  moves farther away from  $p$ , or as it rotates so that it's not aligned with the direction of  $d\omega$ , the  $r^2$  and  $\cos \theta$  terms compensate accordingly to reduce  $d\omega$ .

Therefore, we can write the irradiance integral for the quadrilateral source as

$$E(p, n) = \int_A L \cos \theta_i \frac{\cos \theta_o dA}{r^2},$$



**Figure 5.17:** To compute irradiance at a point  $p$  from a quadrilateral source, it's easier to integrate over the surface area of the source than to integrate over the irregular set of directions that it subtends. The relationship between solid angles and areas given by Equation (5.6) lets us go back and forth between the two approaches.

where  $L$  is the emitted radiance from the surface of the quadrilateral,  $\theta_i$  is the angle between the surface normal at  $p$  and the direction from  $p$  to the point  $p'$  on the light, and  $\theta_o$  is the angle between the surface normal at  $p'$  on the light and the direction from  $p'$  to  $p$  (Figure 5.17).

## 5.6 SURFACE REFLECTION

When light is incident on a surface, the surface scatters the light, reflecting some of it back into the environment. There are two main effects that need to be described to model this reflection: the spectral distribution of the reflected light and its directional distribution. For example, the skin of a lemon mostly absorbs light in the blue wavelengths but reflects most of the light in the red and green wavelengths (recall the lemon skin reflectance SPD in Figure 5.1). Therefore, when it is illuminated with white light, its color is yellow. The skin has pretty much the same color no matter what direction it's being observed from, although for some directions a highlight—a brighter area that is more white than yellow—is visible. In contrast, the light reflected from a point in a mirror depends almost entirely on the viewing direction. At a fixed point on the mirror, as the viewing angle changes, the object that is reflected in the mirror changes accordingly.

Reflection from translucent surfaces is more complex; a variety of materials ranging from skin and leaves to wax and liquids exhibit *subsurface light transport*, where light that enters the surface at one point exits it some distance away. (Consider, for example, how

shining a flashlight in one's mouth makes one's cheeks light up, as light that enters the inside of the cheeks passes through the skin and exits the face.)

There are two abstractions for describing these mechanisms for light reflection: the BRDF and the BSSRDF, described in Sections 5.6.1 and 5.6.2, respectively. The BRDF describes surface reflection at a point neglecting the effect of subsurface light transport; for materials where this transport mechanism doesn't have a significant effect, this simplification introduces little error and makes the implementation of rendering algorithms much more efficient. The BSSRDF generalizes the BRDF and describes the more general setting of light reflection from translucent materials.

### 5.6.1 THE BRDF

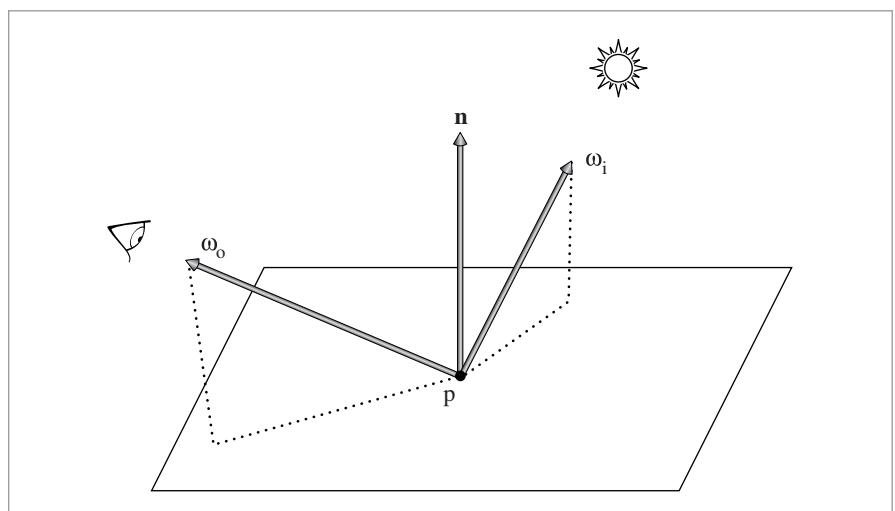
The *bidirectional reflectance distribution function* (BRDF) gives a formalism for describing reflection from a surface. Consider the setting in Figure 5.18: we'd like to know how much radiance is leaving the surface in the direction  $\omega_o$  toward the viewer,  $L_o(p, \omega_o)$ , as a result of incident radiance along the direction  $\omega_i$ ,  $L_i(p, \omega_i)$ .

If the direction  $\omega_i$  is considered as a differential cone of directions, the differential irradiance at  $p$  is

$$dE(p, \omega_i) = L_i(p, \omega_i) \cos \theta_i d\omega_i. \quad [5.7]$$

A differential amount of radiance will be reflected in the direction  $\omega_o$  due to this irradiance. Because of the linearity assumption from geometric optics, the reflected differential radiance is proportional to the irradiance

$$dL_o(p, \omega_o) \propto dE(p, \omega_i).$$



**Figure 5.18: The BRDF.** The bidirectional reflectance distribution function is a 4D function over pairs of directions  $\omega_i$  and  $\omega_o$  that describes how much incident light along  $\omega_i$  is scattered from the surface in the direction  $\omega_o$ .

The constant of proportionality defines the surface's BRDF for the particular pair of directions  $\omega_i$  and  $\omega_o$ :

$$f_r(p, \omega_o, \omega_i) = \frac{dL_o(p, \omega_o)}{dE(p, \omega_i)} = \frac{dL_o(p, \omega_o)}{L_i(p, \omega_i) \cos \theta_i d\omega_i}. \quad (5.8)$$

Physically based BRDFs have two important qualities:

1. *Reciprocity*: For all pairs of directions  $\omega_i$  and  $\omega_o$ ,  $f_r(p, \omega_i, \omega_o) = f_r(p, \omega_o, \omega_i)$ .
2. *Energy conservation*: The total energy of light reflected is less than or equal to the energy of incident light. For all directions  $\omega_o$ ,

$$\int_{\mathcal{H}^2(\mathbf{n})} f_r(p, \omega_o, \omega') \cos \theta' d\omega' \leq 1.$$

The surface's *bidirectional transmittance distribution function* (BTDF), which describes the distribution of transmitted light, can be defined in a manner similar to that for the BRDF. The BTDF is generally denoted by  $f_t(p, \omega_o, \omega_i)$ , where  $\omega_i$  and  $\omega_o$  are in opposite hemispheres around  $p$ . Remarkably, the BTDF does not obey reciprocity as defined above; we will discuss this issue in detail in Sections 8.2 and 16.1.3.

For convenience in equations, we will denote the BRDF and BTDF when considered together as  $f(p, \omega_o, \omega_i)$ ; we will call this the *bidirectional scattering distribution function* (BSDF). Chapter 8 is entirely devoted to describing a variety of BSDFs that are useful for rendering.

Using the definition of the BSDF, we have

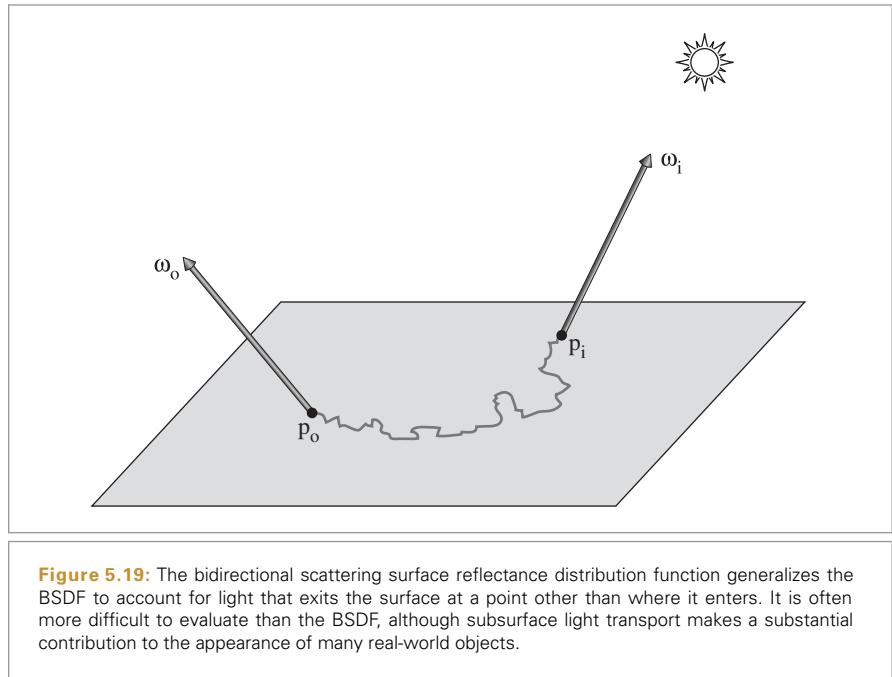
$$dL_o(p, \omega_o) = f(p, \omega_o, \omega_i) L_i(p, \omega_i) |\cos \theta_i| d\omega_i.$$

Here an absolute value has been added to the  $\cos \theta_i$  term. This is done because surface normals in pbrt are not reoriented to lie on the same side of the surface as  $\omega_i$  (many other rendering systems do this, although we find it more useful to leave them in their natural orientation as given by the Shape). Doing so makes it easier to consistently apply conventions like “the surface normal is assumed to point outside the surface” elsewhere in the system. Thus, applying the absolute value to  $\cos \theta$  terms like these ensures that the desired quantity is actually calculated.

We can integrate this equation over the sphere of incident directions around  $p$  to compute the outgoing radiance in direction  $\omega_o$  due to the incident illumination at  $p$  from all directions:

$$L_o(p, \omega_o) = \int_{S^2} f(p, \omega_o, \omega_i) L_i(p, \omega_i) |\cos \theta_i| d\omega_i. \quad (5.9)$$

This is a fundamental equation in rendering; it describes how an incident distribution of light at a point is transformed into an outgoing distribution, based on the scattering properties of the surface. It is often called the *scattering equation* when the sphere  $S^2$  is the domain (as it is here), or the *reflection equation* when just the upper hemisphere  $\mathcal{H}^2(\mathbf{n})$  is being integrated over. One of the key tasks of the integration routines in Chapters 14 and 16 is to evaluate this integral at points on surfaces in the scene.



### 5.6.2 THE BSSRDF

The *bidirectional scattering surface reflectance distribution function* (BSSRDF) is the formalism that describes scattering from materials that exhibit a significant amount of subsurface light transport. It is a distribution function  $S(p_o, \omega_o, p_i, \omega_i)$  that describes the ratio of exitant differential radiance at point  $p_o$  in direction  $\omega_o$  to the incident differential flux at  $p_i$  from direction  $\omega_i$  (Figure 5.19):

$$S(p_o, \omega_o, p_i, \omega_i) = \frac{dL_o(p_o, \omega_o)}{d\Phi(p_i, \omega_i)}. \quad [5.10]$$

The generalization of the scattering equation for the BSSRDF requires integration over surface area *and* incoming direction, turning the 2D scattering Equation (5.9) into a 4D integral. With two more dimensions to integrate over, it is substantially more complex to use in rendering algorithms.

$$L_o(p_o, \omega_o) = \int_A \int_{H^2(n)} S(p_o, \omega_o, p_i, \omega_i) L_i(p_i, \omega_i) |\cos \theta_i| d\omega_i dA. \quad [5.11]$$

As the distance between points  $p_i$  and  $p_o$  increases, the value of  $S$  generally diminishes. This fact can be a substantial help in implementations of subsurface scattering algorithms.

Light transport beneath a surface is described by the same principles as volume light transport in participating media and is described by the equation of transfer, which is introduced in Section 15.1. Subsurface scattering is thus based on the same effects as light scattering in clouds and smoke, just at a smaller scale.

## FURTHER READING

Meyer was one of the first researchers to closely investigate spectral representations in graphics (Meyer and Greenberg 1980; Meyer et al. 1986). Hall (1989) summarized the state of the art in spectral representations through 1989, and Glassner's *Principles of Digital Image Synthesis* (1995) covers the topic through the mid-1990s. Survey articles by Hall (1999), Johnson and Fairchild (1999), and Devlin et al. (2002) are good resources on this topic.

Borges (1991) analyzed the error introduced from the tristimulus representation when used for spectral computation. Peercy (1993) developed a technique based on choosing basis functions in a scene-dependent manner: by looking at the SPDs of the lights and reflecting objects in the scene, a small number of spectral basis functions that could accurately represent the scene's SPDs were found using characteristic vector analysis. Rougeron and Péroche (1997) projected all SPDs in the scene onto a hierarchical basis (the Haar wavelets), and showed that this adaptive representation can be used to stay within a desired error bound. Ward and Eydelberg-Vileshin (2002) developed a method for improving the spectral results from a regular RGB-only rendering system by carefully adjusting the color values provided to the system before rendering.

Another approach to spectral representation was investigated by Sun et al. (2001), who partitioned SPDs into a smooth base SPD and a set of spikes. Each part was represented differently, using basis functions that worked well for each of these parts of the distribution. Drew and Finlayson (2003) applied a "sharp" basis, which is adaptive but has the property that computing the product of two functions in the basis doesn't require a full matrix multiplication as many other basis representations do.

When using a point-sampled representation (like `SampledSpectrum`), it can be difficult to know how many samples are necessary for accurate results. Lehtonen et al. (2006) studied this issue and determined that a 5-nm sample spacing was sufficient for real-world SPDs.

Evans and McCool (1999) introduced stratified wavelength clusters for representing SPDs: the idea is that each spectral computation uses a small fixed number of samples at representative wavelengths, chosen according to the spectral distribution of the light source. Subsequent computations use different wavelengths, such that individual computations are relatively efficient (being based on a small number of samples), but, in the aggregate over a large number of computations, the overall range of wavelengths is well covered. Related to this approach is the idea of computing the result for just a single wavelength in each computation and averaging the results together: this was the method used by Walter et al. (1997) and Morley et al. (2006).

Radziszewski et al. (2009) proposed a technique that generates light-carrying paths according to a single wavelength, while tracking their contribution at several additional wavelengths using efficient SIMD instructions. Combining these contributions using multiple importance sampling led to reduced variance when simulating dispersion through rough refractive boundaries. Wilkie et al. (2014) used equally spaced point samples in the wavelength domain and showed how this approach can also be used for photon mapping and rendering of participating media.

Glassner (1989b) has written an article on the underconstrained problem of converting RGB values (e.g., as selected by the user from a display) to an SPD. Smits (1999) developed an improved method that is the one we have implemented in Section 5.2.2. See Meng et al. (2015) for recent work in this area, including thorough discussion of the complexities involved in doing these conversions accurately.

McCluney's book on radiometry is an excellent introduction to the topic (McCluney 1994). Preisendorfer (1965) also covered radiometry in an accessible manner and delved into the relationship between radiometry and the physics of light. Nicodemus et al. (1977) carefully defined the BRDF, BSSRDF, and various quantities that can be derived from them. See Moon and Spencer (1936, 1948) and Gershun (1939) for classic early introductions to radiometry. Finally, Lambert's seminal early writings about photometry from the mid-18th century have been translated into English by DiLaura (Lambert 1760).

Correctly implementing radiometric computations can be tricky: one missed cosine term and one is computing a completely different quantity than expected. Debugging these sorts of issues can be quite time-consuming. Ou and Pellacini (2010) showed how to use C++'s type system to associate units with each term of these sorts of computations so that, for example, trying to add a radiance value to another value that represents irradiance would trigger a compile time error.

## EXERCISES

- ① 5.1 How many photons would a 50-W lightbulb that emits light at the single wavelength  $\lambda = 600$  nm emit in 1 second?
- ② 5.2 Implement a new representation for spectral basis functions in pbrt. Compare both image quality and rendering time to the `RGBSpectrum` and `SampledSpectrum` representations implemented in this chapter. Be sure to include tricky situations like fluorescent lighting.
- ③ 5.3 Compute the irradiance at a point due to a unit-radius disk  $h$  units directly above its normal with constant outgoing radiance of  $10 \text{ W/m}^2 \text{ sr}$ . Do the computation twice, once as an integral over solid angle and once as an integral over area. (Hint: If the results don't match at first, see Section 13.6.2.)
- ④ 5.4 Similarly, compute the irradiance at a point due to a square quadrilateral with outgoing radiance of  $10 \text{ W/m}^2 \text{ sr}$  that has sides of length 1 and is 1 unit directly above the point in the direction of its surface normal.

# CHAPTER SIX



## CAMERA MODELS

In Chapter 1, we described the pinhole camera model that is commonly used in computer graphics. This model is easy to describe and simulate, but it neglects important effects that lenses have on light passing through them that occur with real cameras. For example, everything rendered with a pinhole camera is in sharp focus—a state of affairs not possible with real lens systems. Such images often look computer generated. More generally, the distribution of radiance leaving a lens system is quite different from the distribution entering it; modeling this effect of lenses is important for accurately simulating the radiometry of image formation.

Camera lens systems also introduce various aberrations that affect the images that they form; for example, *vignetting* causes a darkening toward the edges of images due to less light making it through to the edges of the film or sensor than to the center. Lenses can also cause *pincushion* or *barrel* distortion, which causes straight lines to be imaged as curves. Although lens designers work to minimize aberrations in their designs, they can still have a meaningful effect on images.

Like the Shapes from Chapter 3, cameras in pbrt are represented by an abstract base class. This chapter describes the Camera class and two of its key methods: Camera::GenerateRay() and Camera::GenerateRayDifferential(). The first method computes the world space ray corresponding to a sample position on the film plane. By generating these rays in different ways based on different models of image formation, the cameras in pbrt can create many types of images of the same 3D scene. The second method not only generates this ray but also computes information about the image area that the ray is sampling; this information is used for anti-aliasing computations in Chapter 10, for example. In Section 16.1.1, a few additional Camera methods will be introduced to support bidirectional light transport algorithms.

In this chapter, we will show a few implementations of the Camera interface, starting by implementing the ideal pinhole model with some generalizations and finishing with

a fairly realistic model that simulates light passing through a collection of glass lens elements to form an image, similar to real-world cameras.

## 6.1 CAMERA MODEL

The abstract Camera base class holds generic camera options and defines the interface that all camera implementations must provide. It is defined in the files core/camera.h and core/camera.cpp.

```
(Camera Declarations) ≡
class Camera {
public:
    (Camera Interface 356)
    (Camera Public Data 356)
};
```

The base Camera constructor takes several parameters that are appropriate for all camera types. One of the most important is the transformation that places the camera in the scene, which is stored in the CameraToWorld member variable. The Camera stores an AnimatedTransform (rather than just a regular Transform) so that the camera itself can be moving over time.

Real-world cameras have a shutter that opens for a short period of time to expose the film to light. One result of this nonzero exposure time is *motion blur*: objects that are in motion relative to the camera during the exposure are blurred. All Cameras therefore store a shutter open and shutter close time and are responsible for generating rays with associated times at which to sample the scene. Given an appropriate distribution of ray times between the shutter open time and the shutter close time, it is possible to compute images that exhibit motion blur.

Cameras also contain a pointer to an instance of the Film class to represent the final image (Film is described in Section 7.9), and a pointer to a Medium instance to represent the scattering medium that the camera lies in (Medium is described in Section 11.3).

Camera implementations must pass along parameters that set these values to the Camera constructor. We will only show the constructor's prototype here because its implementation just copies the parameters to the corresponding member variables.

<i>(Camera Interface) ≡</i>	356
Camera(const AnimatedTransform &CameraToWorld, Float shutterOpen,	
Float shutterClose, Film *film, const Medium *medium);	
<i>(Camera Public Data) ≡</i>	356
AnimatedTransform CameraToWorld;	
const Float shutterOpen, shutterClose;	
Film *film;	
const Medium *medium;	

AnimatedTransform 103  
 Camera 356  
 Camera::GenerateRay() 357  
 Film 484  
 Float 1062  
 Medium 684  
 Transform 83

The first method that camera subclasses need to implement is Camera::GenerateRay(), which should compute the ray corresponding to a given sample. It is important that the

direction component of the returned ray be normalized—many other parts of the system will depend on this behavior.

```
<Camera Interface> +≡
    virtual Float GenerateRay(const CameraSample &sample,
                               Ray *ray) const = 0;
```

356

The CameraSample structure holds all of the sample values needed to specify a camera ray. Its pFilm member gives the point on the film to which the generated ray carries radiance. The point on the lens the ray passes through is in pLens (for cameras that include the notion of lenses), and CameraSample::time gives the time at which the ray should sample the scene; implementations should use this value to linearly interpolate within the shutterOpen–shutterClose time range. (Choosing these various sample values carefully can greatly increase the quality of final images; this is the topic of much of Chapter 7.)

GenerateRay() also returns a floating-point value that affects how much the radiance arriving at the film plane along the generated ray will contribute to the final image. Simple camera models can just return a value of 1, but cameras that simulate real physical lens systems like the one in Section 6.4 to set this value to indicate how much light the ray carries through the lenses based on their optical properties. (See Sections 6.4.7 and 13.6.6 for more information about how exactly this weight is computed and used.)

```
<Camera Declarations> +≡
    struct CameraSample {
        Point2f pFilm;
        Point2f pLens;
        Float time;
    };
```

The GenerateRayDifferential() method computes a main ray like GenerateRay() but also computes the corresponding rays for pixels shifted one pixel in the  $x$  and  $y$  directions on the film plane. This information about how camera rays change as a function of position on the film helps give other parts of the system a notion of how much of the film area a particular camera ray's sample represents, which is particularly useful for anti-aliasing texture map lookups.

```
<Camera Method Definitions> ≡
    Float Camera::GenerateRayDifferential(const CameraSample &sample,
                                            RayDifferential *rd) const {
        Float wt = GenerateRay(sample, rd);
        <Find camera ray after shifting one pixel in the x direction 358>
        <Find camera ray after shifting one pixel in the y direction>
        rd->hasDifferentials = true;
        return wt;
    }
```

Camera::GenerateRay() 357  
 CameraSample 357  
 Float 1062  
 Point2f 68  
 Ray 73  
 RayDifferential 75  
 RayDifferential::  
 hasDifferentials  
 75

Finding the ray for one pixel over in  $x$  is just a matter of initializing a new CameraSample and copying the appropriate values returned by calling GenerateRay() into the Ray Differential structure. The implementation of the fragment *(Find ray after shifting one pixel in the y direction)* follows similarly and isn't included here.

*(Find camera ray after shifting one pixel in the x direction) ≡* **357**

```
CameraSample sshift = sample;
sshift.pFilm.x++;
Ray rx;
Float wtx = GenerateRay(sshift, &rx);
if (wtx == 0) return 0;
rd->rxOrigin = rx.o;
rd->rxDirection = rx.d;
```

### 6.1.1 CAMERA COORDINATE SPACES

We have already made use of two important modeling coordinate spaces, object space and world space. We will now introduce an additional coordinate space, *camera space*, which has the camera at its origin. We have:

- *Object space*: This is the coordinate system in which geometric primitives are defined. For example, spheres in pbrt are defined to be centered at the origin of their object space.
- *World space*: While each primitive may have its own object space, all objects in the scene are placed in relation to a single world space. Each primitive has an object-to-world transformation that determines where it is located in world space. World space is the standard frame that all other spaces are defined in terms of.
- *Camera space*: A camera is placed in the scene at some world space point with a particular viewing direction and orientation. This camera defines a new coordinate system with its origin at the camera’s location. The z axis of this coordinate system is mapped to the viewing direction, and the y axis is mapped to the up direction. This is a handy space for reasoning about which objects are potentially visible to the camera. For example, if an object’s camera space bounding box is entirely behind the  $z = 0$  plane (and the camera doesn’t have a field of view wider than 180 degrees), the object will not be visible to the camera.

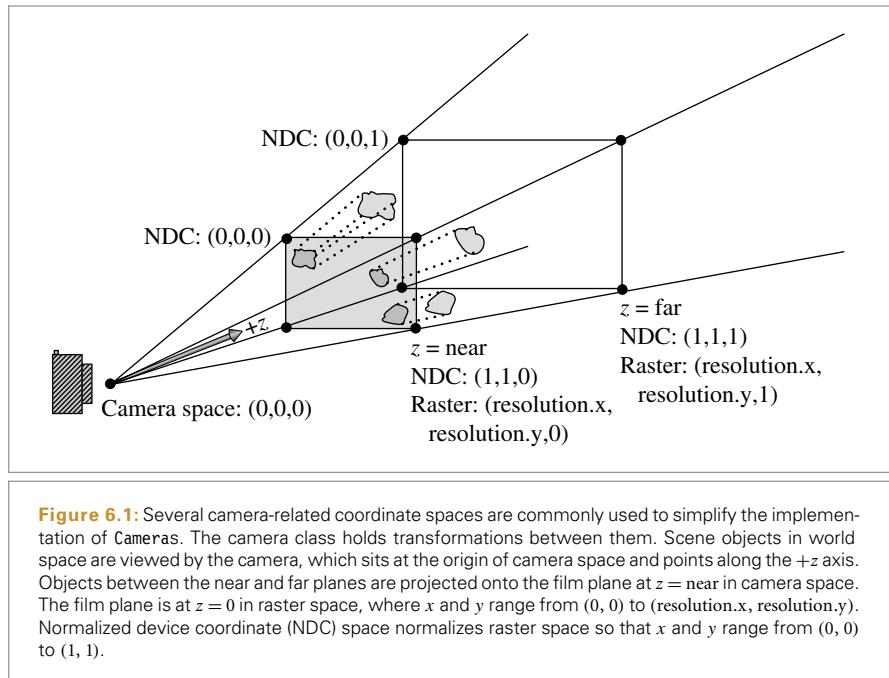
## 6.2 PROJECTIVE CAMERA MODELS

One of the fundamental issues in 3D computer graphics is the *3D viewing problem*: how to project a 3D scene onto a 2D image for display. Most of the classic approaches can be expressed by a  $4 \times 4$  projective transformation matrix. Therefore, we will introduce a projection matrix camera class, `ProjectiveCamera`, and then define two camera models based on it. The first implements an orthographic projection, and the other implements a perspective projection—two classic and widely used projections.

*(Camera Declarations) +≡*

```
class ProjectiveCamera : public Camera {
public:
    (ProjectiveCamera Public Methods 360)
protected:
    (ProjectiveCamera Protected Data 360)
};
```

`Camera` **356**  
`Camera::GenerateRay()` **357**  
`CameraSample` **357**  
`CameraSample::pFilm` **357**  
`Float` **1062**  
`Ray` **73**  
`Ray::d` **73**  
`Ray::o` **73**  
`RayDifferential::rxDirection` **75**  
`RayDifferential::rxOrigin` **75**



Three more coordinate systems (summarized in Figure 6.1) are useful for defining and discussing projective cameras:

- *Screen space:* Screen space is defined on the film plane. The camera projects objects in camera space onto the film plane; the parts inside the *screen window* are visible in the image that is generated. Depth  $z$  values in screen space range from 0 to 1, corresponding to points at the near and far clipping planes, respectively. Note that, although this is called “screen” space, it is still a 3D coordinate system, since  $z$  values are meaningful.
- *Normalized device coordinate (NDC) space:* This is the coordinate system for the actual image being rendered. In  $x$  and  $y$ , this space ranges from  $(0, 0)$  to  $(1, 1)$ , with  $(0, 0)$  being the upper-left corner of the image. Depth values are the same as in screen space, and a linear transformation converts from screen to NDC space.
- *Raster space:* This is almost the same as NDC space, except the  $x$  and  $y$  coordinates range from  $(0, 0)$  to  $(\text{resolution.x}, \text{resolution.y})$ .

Projective cameras use  $4 \times 4$  matrices to transform among all of these spaces, but cameras with unusual imaging characteristics can't necessarily represent all of these transformations with matrices.

[Camera 356](#)

[ProjectiveCamera 358](#)

In addition to the parameters required by the Camera base class, the ProjectiveCamera takes the projective transformation matrix, the screen space extent of the image, and additional parameters related to depth of field. Depth of field, which will be described

and implemented at the end of this section, simulates the blurriness of out-of-focus objects that occurs in real lens systems.

*(ProjectiveCamera Public Methods) ≡* 358

```
ProjectiveCamera(const AnimatedTransform &CameraToWorld,
                 const Transform &CameraToScreen, const Bounds2f &screenWindow,
                 Float shutterOpen, Float shutterClose, Float lensr, Float focald,
                 Film *film, const Medium *medium)
: Camera(CameraToWorld, shutterOpen, shutterClose, film, medium),
  CameraToScreen(CameraToScreen) {
  (Initialize depth of field parameters 374)
  (Compute projective camera transformations 360)
}
```

ProjectiveCamera implementations pass the projective transformation up to the base class constructor shown here. This transformation gives the camera-to-screen projection; from that, the constructor can easily compute the other transformation that will be needed, to go all the way from raster space to camera space.

*(Compute projective camera transformations) ≡* 360

```
(Compute projective camera screen transformations 360)
RasterToCamera = Inverse(CameraToScreen) * RasterToScreen;
```

*(ProjectiveCamera Protected Data) ≡* 358

```
Transform CameraToScreen, RasterToCamera;
```

The only nontrivial transformation to compute in the constructor is the screen-to-raster projection. In the following code, note the composition of transformations where (reading from bottom to top), we start with a point in screen space, translate so that the upper-left corner of the screen is at the origin, and then scale by the reciprocal of the screen width and height, giving us a point with  $x$  and  $y$  coordinates between 0 and 1 (these are NDC coordinates). Finally, we scale by the raster resolution, so that we end up covering the entire raster range from  $(0, 0)$  up to the overall raster resolution. An important detail here is that the  $y$  coordinate is inverted by this transformation; this is necessary because increasing  $y$  values move up the image in screen coordinates but down in raster coordinates.

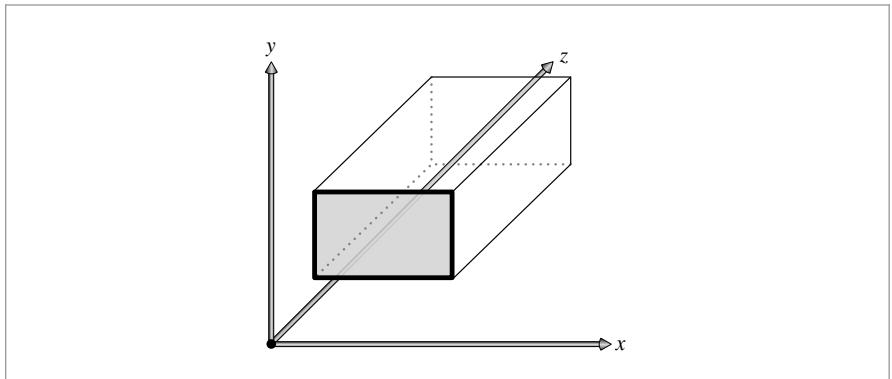
*(Compute projective camera screen transformations) ≡* 360

```
ScreenToRaster = Scale(film->fullResolution.x,
                      film->fullResolution.y, 1) *
  Scale(1 / (screenWindow.pMax.x - screenWindow.pMin.x),
        1 / (screenWindow.pMin.y - screenWindow.pMax.y), 1) *
  Translate(Vector3f(-screenWindow.pMin.x, -screenWindow.pMax.y, 0));
RasterToScreen = Inverse(ScreenToRaster);
```

*(ProjectiveCamera Protected Data) +≡* 358

```
Transform ScreenToRaster, RasterToScreen;
```

AnimatedTransform 103  
 Bounds2f 76  
 Camera 356  
 Film 484  
 Film::fullResolution 485  
 Float 1062  
 Inverse() 1081  
 Medium 684  
 ProjectiveCamera 358  
 ProjectiveCamera::  
 CameraToScreen  
 360  
 ProjectiveCamera::  
 RasterToCamera  
 360  
 ProjectiveCamera::  
 RasterToScreen  
 360  
 ProjectiveCamera::  
 ScreenToRaster  
 360  
 Scale() 87  
 Transform 83  
 Translate() 87  
 Vector3f 60



**Figure 6.2:** The orthographic view volume is an axis-aligned box in camera space, defined such that objects inside the region are projected onto the  $z = \text{near}$  face of the box.

### 6.2.1 ORTHOGRAPHIC CAMERA

*(OrthographicCamera Declarations)* ≡

```
class OrthographicCamera : public ProjectiveCamera {
public:
    (OrthographicCamera Public Methods 361)
private:
    (OrthographicCamera Private Data 363)
};
```

The orthographic camera, defined in the files `cameras/orthographic.h` and `cameras/orthographic.cpp`, is based on the orthographic projection transformation. The orthographic transformation takes a rectangular region of the scene and projects it onto the front face of the box that defines the region. It doesn't give the effect of *foreshortening*—objects becoming smaller on the image plane as they get farther away—but it does leave parallel lines parallel, and it preserves relative distance between objects. Figure 6.2 shows how this rectangular volume defines the visible region of the scene. Figure 6.3 compares the result of using the orthographic projection for rendering to the perspective projection defined in the next section.

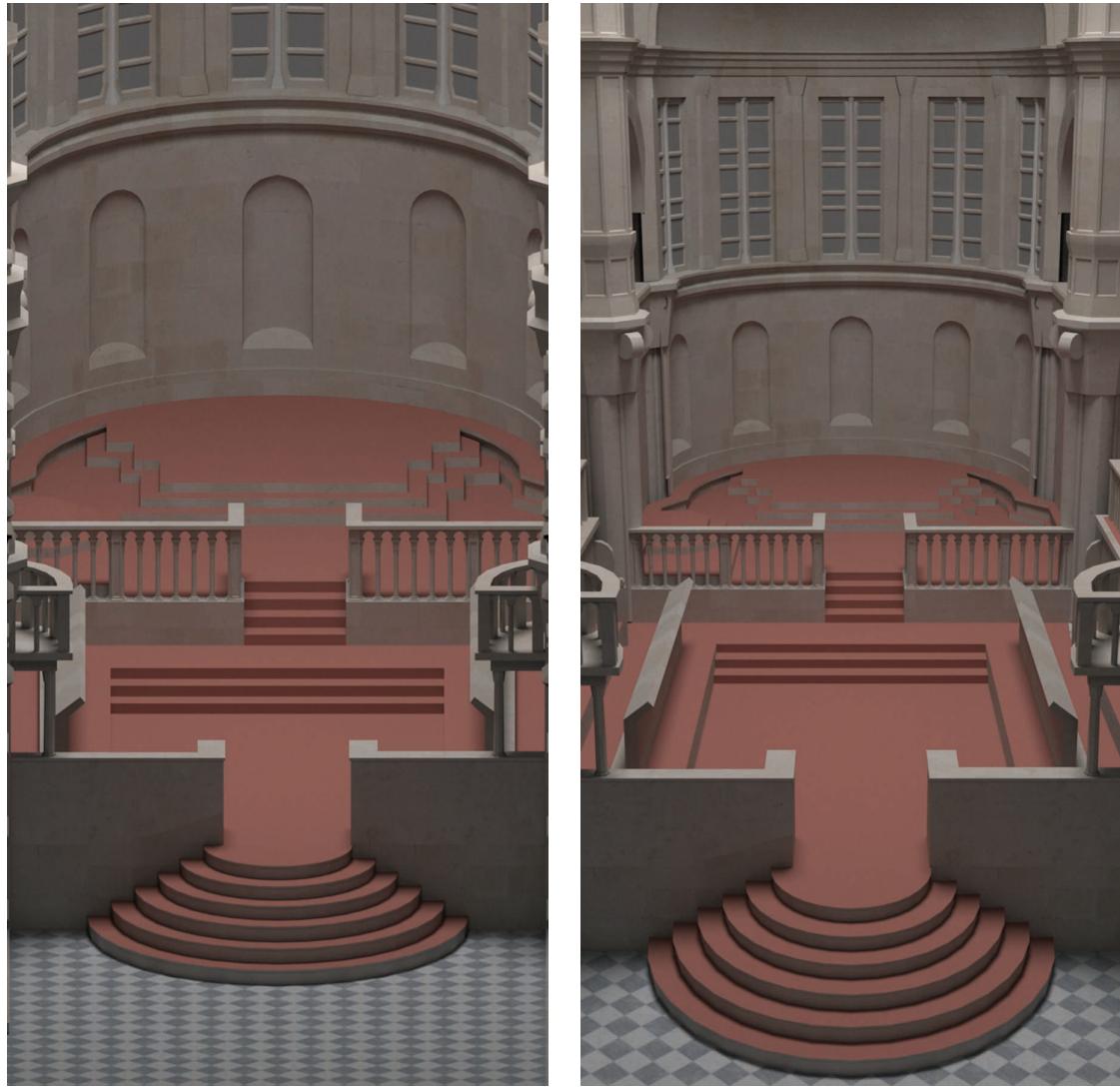
The orthographic camera constructor generates the orthographic transformation matrix with the `Orthographic()` function, which will be defined shortly.

*(OrthographicCamera Public Methods)* ≡

```
OrthographicCamera(const AnimatedTransform &CameraToWorld,
                    const Bounds2f &screenWindow, Float shutterOpen,
                    Float shutterClose, Float lensRadius, Float focalDistance,
                    Film *film, const Medium *medium)
: ProjectiveCamera(CameraToWorld, Orthographic(0, 1),
                  screenWindow, shutterOpen, shutterClose,
                  lensRadius, focalDistance, film, medium) {
    (Compute differential changes in origin for orthographic camera rays 363)
}
```

`AnimatedTransform` 103  
`Bounds2f` 76  
`Film` 484  
`Float` 1062  
`Medium` 684  
`Orthographic()` 363  
`OrthographicCamera` 361  
`ProjectiveCamera` 358

361



**Figure 6.3: Images of the Church Model.** Rendered with (left) orthographic and (right) perspective cameras. Note that features like the stairs, checks on the floor, and back windows are rendered quite differently with the two models. The lack of foreshortening makes the orthographic view feel like it has less depth, although it does preserve parallel lines, which can be a useful property.

The orthographic viewing transformation leaves  $x$  and  $y$  coordinates unchanged but maps  $z$  values at the near plane to 0 and  $z$  values at the far plane to 1. To do this, the scene is first translated along the  $z$  axis so that the near plane is aligned with  $z = 0$ . Then, the scene is scaled in  $z$  so that the far plane maps to  $z = 1$ . The composition of these two transformations gives the overall transformation. (For a ray tracer like pbrt, we'd like the near plane to be at 0 so that rays start at the plane that goes through the camera's position; the far plane offset doesn't particularly matter.)

```
(Transform Method Definitions) +≡
    Transform Orthographic(Float zNear, Float zFar) {
        return Scale(1, 1, 1 / (zFar - zNear)) *
            Translate(Vector3f(0, 0, -zNear));
    }
```

Thanks to the simplicity of the orthographic projection, it's easy to directly compute the differential rays in the  $x$  and  $y$  directions in the `GenerateRayDifferential()` method. The directions of the differential rays will be the same as the main ray (as they are for all rays generated by an orthographic camera), and the difference in origins will be the same for all rays. Therefore, the constructor here precomputes how much the ray origins shift in camera space coordinates due to a single pixel shift in the  $x$  and  $y$  directions on the film plane.

*(Compute differential changes in origin for orthographic camera rays) ≡*

361

```
dxCamera = RasterToCamera(Vector3f(1, 0, 0));
dyCamera = RasterToCamera(Vector3f(0, 1, 0));
```

*(OrthographicCamera Private Data) ≡*

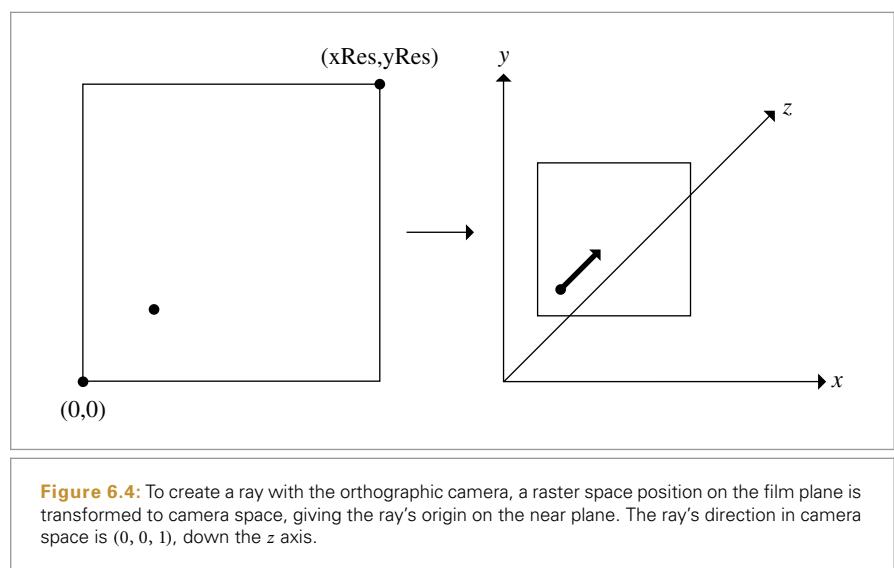
361

```
Vector3f dxCamera, dyCamera;
```

We can now go through the code to take a sample point in raster space and turn it into a camera ray. The process is summarized in Figure 6.4. First, the raster space sample position is transformed into a point in camera space, giving a point located on the near plane, which is the origin of the camera ray. Because the camera space viewing direction points down the  $z$  axis, the camera space ray direction is  $(0, 0, 1)$ .

If depth of field has been enabled for this scene, the ray's origin and direction are modified so that depth of field is simulated. Depth of field will be explained later in this section. The ray's time value is set by linearly interpolating between the shutter open and shutter

```
Float 1062
ProjectiveCamera::
    RasterToCamera
360
Scale() 87
Transform 83
Translate() 87
Vector3f 60
```



close times by the `CameraSample::time` offset (which is in the range [0, 1]). Finally, the ray is transformed into world space before being returned.

*(OrthographicCamera Definitions) ≡*

```
Float OrthographicCamera::GenerateRay(const CameraSample &sample,
                                       Ray *ray) const {
    (Compute raster and camera sample positions 364)
    *ray = Ray(pCamera, Vector3f(0, 0, 1));
    (Modify ray for depth of field 374)
    ray->time = Lerp(sample.time, shutterOpen, shutterClose);
    ray->medium = medium;
    *ray = CameraToWorld(*ray);
    return 1;
}
```

Once all of the transformation matrices have been set up, it's easy to transform the raster space sample point to camera space.

*(Compute raster and camera sample positions) ≡*

364, 367

```
Point3f pFilm = Point3f(sample.pFilm.x, sample.pFilm.y, 0);
Point3f pCamera = RasterToCamera(pFilm);
```

The implementation of `GenerateRayDifferential()` performs the same computation to generate the main camera ray. The differential ray origins are found using the offsets computed in the `OrthographicCamera` constructor, and then the full ray differential is transformed to world space.

*(OrthographicCamera Definitions) +≡*

```
Float OrthographicCamera::GenerateRayDifferential(
    const CameraSample &sample, RayDifferential *ray) const {
    (Compute main orthographic viewing ray)
    (Compute ray differentials for OrthographicCamera 364)
    ray->time = Lerp(sample.time, shutterOpen, shutterClose);
    ray->hasDifferentials = true;
    ray->medium = medium;
    *ray = CameraToWorld(*ray);
    return 1;
}
```

*(Compute ray differentials for OrthographicCamera) ≡*

364

```
if (lensRadius > 0) {
    (Compute OrthographicCamera ray differentials accounting for lens)
} else {
    ray->rxOrigin = ray->o + dxCamera;
    ray->ryOrigin = ray->o + dyCamera;
    ray->rxDirection = ray->ryDirection = ray->d;
}
```

Camera::CameraToWorld 356  
 Camera::medium 356  
 Camera::shutterClose 356  
 Camera::shutterOpen 356  
 CameraSample 357  
 CameraSample::pFilm 357  
 CameraSample::time 357  
 Float 1062  
 Lerp() 1079  
 OrthographicCamera 361  
 OrthographicCamera::dxCamera 363  
 OrthographicCamera::dyCamera 363  
 Point3f 68  
 ProjectiveCamera::lensRadius 374  
 ProjectiveCamera::  
 RasterToCamera 360  
 Ray 73  
 Ray::d 73  
 Ray::medium 74  
 Ray::o 73  
 Ray::time 73  
 RayDifferential 75  
 RayDifferential::  
 hasDifferentials 75  
 RayDifferential::rxDirection 75  
 RayDifferential::rxOrigin 75  
 RayDifferential::ryDirection 75  
 RayDifferential::ryOrigin 75  
 Vector3f 60

### 6.2.2 PERSPECTIVE CAMERA

The perspective projection is similar to the orthographic projection in that it projects a volume of space onto a 2D film plane. However, it includes the effect of foreshortening: objects that are far away are projected to be smaller than objects of the same size that are closer. Unlike the orthographic projection, the perspective projection doesn't preserve distances or angles, and parallel lines no longer remain parallel. The perspective projection is a reasonably close match to how an eye or camera lens generates images of the 3D world. The perspective camera is implemented in the files cameras/perspective.h and cameras/perspective.cpp.

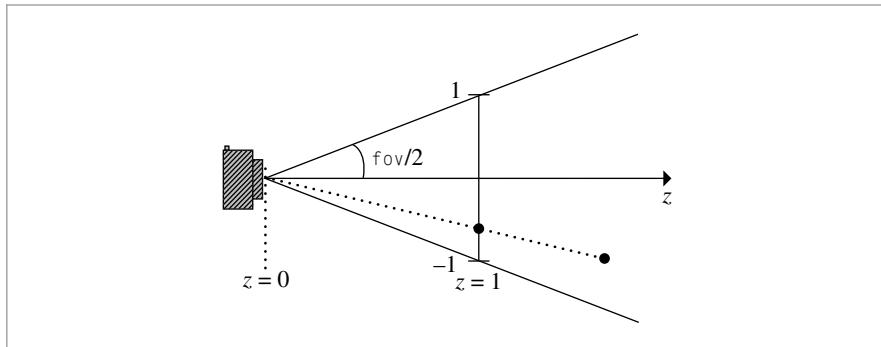
```
(PerspectiveCamera Declarations) ≡
class PerspectiveCamera : public ProjectiveCamera {
public:
    (PerspectiveCamera Public Methods 367)
private:
    (PerspectiveCamera Private Data 367)
};

(PerspectiveCamera Method Definitions) ≡
PerspectiveCamera::PerspectiveCamera(
    const AnimatedTransform &CameraToWorld,
    const Bounds2f &screenWindow, Float shutterOpen,
    Float shutterClose, Float lensRadius, Float focalDistance,
    Float fov, Film *film, const Medium *medium)
: ProjectiveCamera(CameraToWorld, Perspective(fov, 1e-2f, 1000.f),
    screenWindow, shutterOpen, shutterClose,
    lensRadius, focalDistance, film, medium) {
    (Compute differential changes in origin for perspective camera rays 367)
    (Compute image plane bounds at z = 1 for PerspectiveCamera 951)
}
```

The perspective projection describes perspective viewing of the scene. Points in the scene are projected onto a viewing plane perpendicular to the  $z$  axis. The `Perspective()` function computes this transformation; it takes a field-of-view angle in `fov` and the distances to a near  $z$  plane and a far  $z$  plane. After the perspective projection, points at the near  $z$  plane are mapped to have  $z = 0$ , and points at the far plane have  $z = 1$  (Figure 6.5). For rendering systems based on rasterization, it's important to set the positions of these planes carefully; they determine the  $z$  range of the scene that is rendered, but setting them with too many orders of magnitude variation between their values can lead to numerical precision errors. For a ray tracers like pbrt, they can be set arbitrarily as they are here.

AnimatedTransform 103  
 Bounds2f 76  
 Film 484  
 Float 1062  
 Medium 684  
 Perspective() 365  
 PerspectiveCamera 365  
 ProjectiveCamera 358  
 Transform 83

```
(Transform Method Definitions) +≡
Transform Perspective(Float fov, Float n, Float f) {
    (Perform projective divide for perspective projection 366)
    (Scale canonical perspective view to specified field of view 367)
}
```



**Figure 6.5:** The perspective transformation matrix projects points in camera space onto the film plane. The  $x'$  and  $y'$  coordinates of the projected points are equal to the unprojected  $x$  and  $y$  coordinates divided by the  $z$  coordinate. The projected  $z'$  coordinate is computed so that points on the near plane map to  $z' = 0$  and points on the far plane map to  $z' = 1$ .

The transformation is most easily understood in two steps:

1. Points  $p$  in camera space are projected onto the viewing plane. A bit of algebra shows that the projected  $x'$  and  $y'$  coordinates on the viewing plane can be computed by dividing  $x$  and  $y$  by the point's  $z$  coordinate value. The projected  $z$  depth is remapped so that  $z$  values at the near plane are 0 and  $z$  values at the far plane are 1. The computation we'd like to do is

$$\begin{aligned}x' &= x/z \\y' &= y/z \\z' &= \frac{f(z - n)}{z(f - n)}.\end{aligned}$$

All of this computation can be encoded in a  $4 \times 4$  matrix using homogeneous coordinates:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & -\frac{fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

*(Perform projective divide for perspective projection)  $\equiv$*   
 Matrix4x4 persp(1, 0, 0, 0,  
 0, 1, 0, 0,  
 0, 0, f / (f - n), -fn / (f - n),  
 0, 0, 1, 0);

365

2. The angular field of view (fov) specified by the user is accounted for by scaling the  $(x, y)$  values on the projection plane so that points inside the field of view project to coordinates between  $[-1, 1]$  on the view plane. For square images, both  $x$  and  $y$  lie between  $[-1, 1]$  in screen space. Otherwise, the direction in which the image is narrower maps to  $[-1, 1]$ , and the wider direction maps to a proportionally larger

Matrix4x4 1081

range of screen space values. Recall that the tangent is equal to the ratio of the opposite side of a right triangle to the adjacent side. Here the adjacent side has length 1, so the opposite side has the length  $\tan(\text{fov}/2)$ . Scaling by the reciprocal of this length maps the field of view to range from  $[-1, 1]$ .

```

<Scale canonical perspective view to specified field of view> ≡
  Float invTanAng = 1 / std::tan(Radians(fov) / 2);
  return Scale(invTanAng, invTanAng, 1) * Transform(persp);

```

365

Similar to the `OrthographicCamera`, information about how the camera rays generated by the `PerspectiveCamera` change as we shift pixels on the film plane can be precomputed in the constructor. Here, we compute the change in position on the near perspective plane in camera space with respect to shifts in pixel location.

```
(Compute differential changes in origin for perspective camera rays) ==
    dxCamera = (RasterToCamera(Point3f(1, 0, 0)) -
                RasterToCamera(Point3f(0, 0, 0)));
    dyCamera = (RasterToCamera(Point3f(0, 1, 0)) -
                RasterToCamera(Point3f(0, 0, 0)));
```

365

*<PerspectiveCamera Private Data>* ≡  
Vector3f dxCamera, dyCamera;

365

Camera::CameraToWorld 356  
Camera::medium 356  
Camera::shutterClose 356  
Camera::shutterOpen 356  
CameraSample 357  
CameraSample::time 357  
Float 1062  
Lerp() 1079  
OrthographicCamera 361  
PerspectiveCamera 365  
PerspectiveCamera::dxCamera  
367  
PerspectiveCamera::dyCamera  
367  
Point3f 68  
ProjectiveCamera::  
RasterToCamera  
360  
Radians() 1063  
Ray 73  
Ray::medium 74  
Ray::time 73  
RayDifferential 75  
Scale() 87  
Transform 83  
Vector3f 60

With the perspective projection, all rays originate from the origin,  $(0, 0, 0)$ , in camera space. A ray's direction is given by the vector from the origin to the point on the near plane,  $p_{\text{Camera}}$ , that corresponds to the provided `CameraSample`'s `pFilm` location. In other words, the ray's vector direction is component-wise equal to this point's position, so rather than doing a useless subtraction to compute the direction, we just initialize the direction directly from the point  $p_{\text{Camera}}$ .

*⟨PerspectiveCamera Method Definitions⟩* + ≡

```

    Float PerspectiveCamera::GenerateRay(const CameraSample &sample,
                                          Ray *ray) const {
    (Compute raster and camera sample positions 364)
    *ray = Ray(Point3f(0, 0, 0), Normalize(Vector3f(pCamera)));
    (Modify ray for depth of field 374)
    ray->time = Lerp(sample.time, shutterOpen, shutterClose);
    ray->medium = medium;
    *ray = CameraToWorld(*ray);
    return 1;
}

```

The `GenerateRayDifferential()` method follows the implementation of `GenerateRay()`, except for an additional fragment that computes the differential rays.

*\PerspectiveCamera Public Methods*  $\equiv$

865

```
(Compute offset rays for PerspectiveCamera ray differentials) ≡
    if (lensRadius > 0) {
        (Compute PerspectiveCamera ray differentials accounting for lens)
    } else {
        ray->rxOrigin = ray->ryOrigin = ray->o;
        ray->rxDirection = Normalize(Vector3f(pCamera) + dxCamera);
        ray->ryDirection = Normalize(Vector3f(pCamera) + dyCamera);
    }
```

### 6.2.3 THE THIN LENS MODEL AND DEPTH OF FIELD

An ideal pinhole camera that only allows rays passing through a single point to reach the film isn't physically realizable; while it's possible to make cameras with extremely small apertures that approach this behavior, small apertures allow relatively little light to reach the film sensor. With a small aperture, long exposure times are required to capture enough photons to accurately capture the image, which in turn can lead to blur from objects in the scene moving while the camera shutter is open.

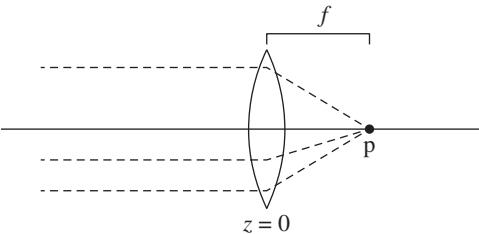
Real cameras have lens systems that focus light through a finite-sized aperture onto the film plane. Camera designers (and photographers using cameras with adjustable apertures) face a trade-off: the larger the aperture, the more light reaches the film and the shorter the exposures that are needed. However, lenses can only focus on a single plane (the *focal plane*), and the farther objects in the scene are from this plane, the blurrier they are. The larger the aperture, the more pronounced this effect is: objects at depths different from the one the lens system has in focus become increasingly blurry.

The camera model in Section 6.4 implements a fairly accurate simulation of lens systems in realistic cameras. For the simple camera models introduced so far, we can apply a classic approximation from optics, the *thin lens approximation*, to model the effect of finite apertures with traditional computer graphics projection models. The thin lens approximation models an optical system as a single lens with spherical profiles, where the thickness of the lens is small relative to the radius of curvature of the lens. (The more general thick lens approximation, which doesn't assume that the lens's thickness is negligible, is introduced in Section 6.4.3.)

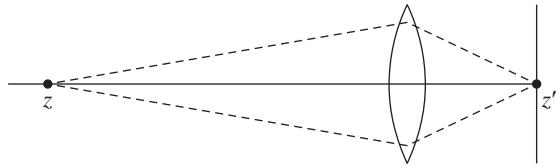
Under the thin lens approximation, parallel incident rays passing through the lens focus at a point called behind the lens called the *focal point*. The distance the focal point is behind the lens,  $f$ , is the lens's *focal length*. If the film plane is placed at a distance equal to the focal length behind the lens, then objects infinitely far away will be in focus, as they image to a single point on the film.

Figure 6.6 illustrates the basic setting. Here we've followed the typical lens coordinate system convention of placing the lens perpendicular to the  $z$  axis, with the lens at  $z = 0$  and the scene along  $-z$ . (Note that this is a different coordinate system from the one we used for camera space, where the viewing direction is  $+z$ .) Distances on the scene side of the lens are denoted with unprimed variables  $z$ , and distances on the film side of the lens (positive  $z$ ) are primed,  $z'$ .

PerspectiveCamera 365  
 PerspectiveCamera::dxCamera 367  
 PerspectiveCamera::dyCamera 367  
 RayDifferential::rxDirection 75  
 RayDifferential::rxOrigin 75  
 RayDifferential::ryDirection 75  
 RayDifferential::ryOrigin 75  
 Vector3f 60



**Figure 6.6:** A thin lens, located along the  $z$  axis at  $z = 0$ . Parallel incident rays (dashed lines) passing through a thin lens all pass through a point  $p$ , the focal point. The distance between the lens and the focal point,  $f$ , is the lens's focal length.



**Figure 6.7:** To focus a thin lens at a depth  $z$  in the scene, Equation (6.2) can be used to compute the distance  $z'$  on the film side of the lens that points at  $z$  focus to. Focusing is performed by adjusting the distance between the lens and the film plane.

For points in the scene at a depth  $z$  from a thin lens with focal length  $f$ , the *Gaussian lens equation* relates the distances from the object to the lens and from lens to the image of the point:

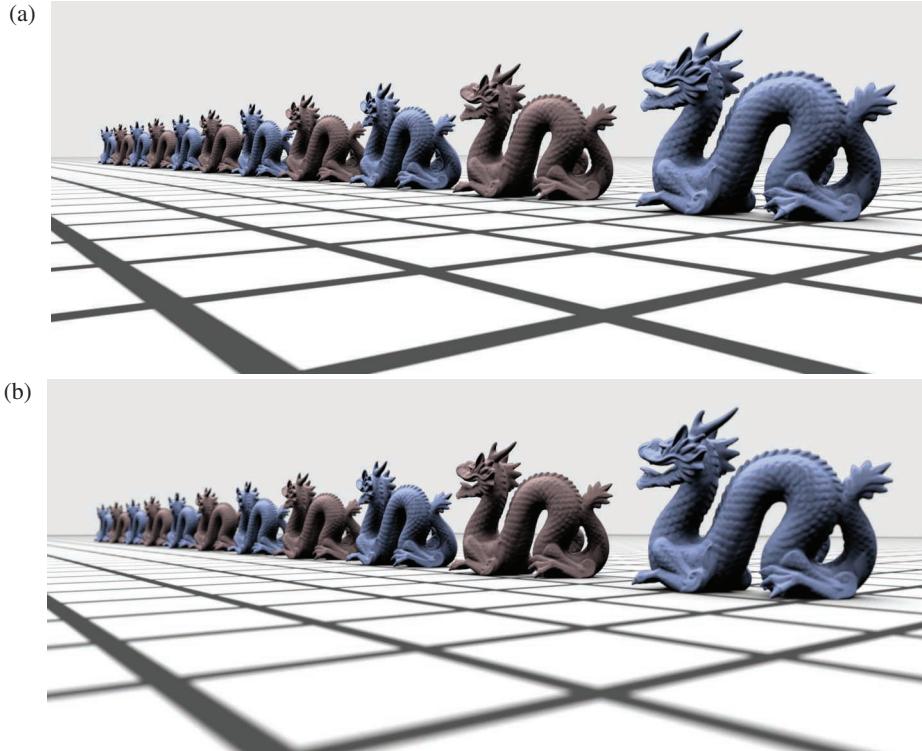
$$\frac{1}{z'} - \frac{1}{z} = \frac{1}{f}. \quad (6.1)$$

Note that for  $z = -\infty$ , we have  $z' = f$ , as expected.

We can use the Gaussian lens equation to solve for the distance between the lens and the film that sets the plane of focus at some  $z$ , the *focal distance* (Figure 6.7):

$$z' = \frac{fz}{f + z}. \quad (6.2)$$

A point that doesn't lie on the plane of focus is imaged to a disk on the film plane, rather than to a single point. This boundary of this disk is called the *circle of confusion*. The size of the circle of confusion is affected by the diameter of the aperture that light rays pass through, the focal distance, and the distance between the object and the lens. Figures 6.8 and 6.9 show this effect, depth of field, in a scene with a series of copies of the dragon model. Figure 6.8(a) is rendered with an infinitesimal aperture and thus without any



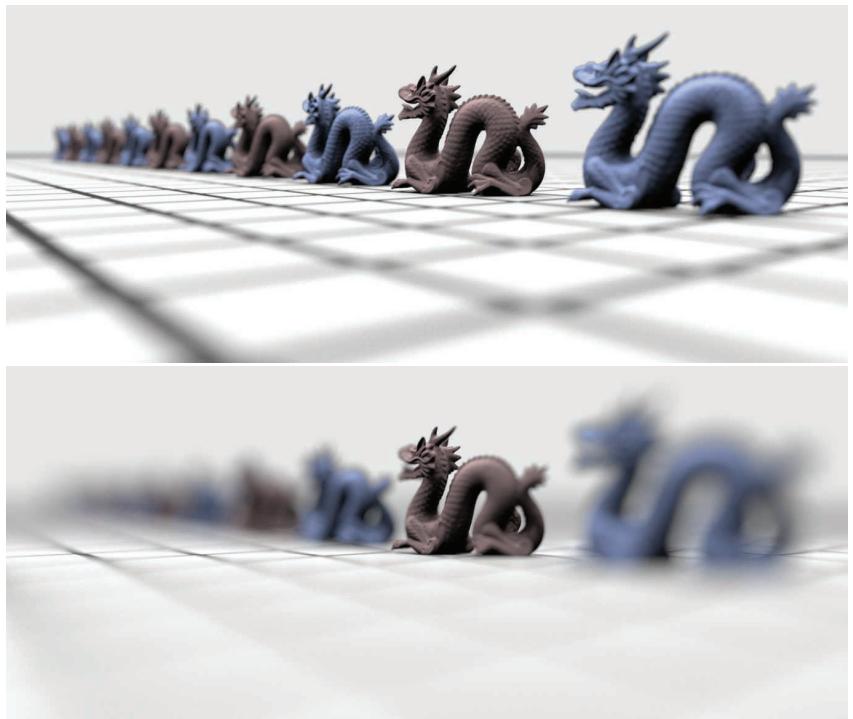
**Figure 6.8:** (a) Scene rendered with no depth of field and (b) depth of field due to a relatively small lens aperture, which gives only a small amount of blurriness in the out-of-focus regions.

depth of field effects. Figures 6.8(b) and 6.9 show the increase in blurriness as the size of the lens aperture is increased. Note that the second dragon from the right remains in focus throughout all of the images, as the plane of focus has been placed at its depth. Figure 6.10 shows depth of field used to render the landscape scene. Note how the effect draws the viewer's eye to the in-focus grass in the center of the image.

In practice, objects do not have to be exactly on the plane of focus to appear in sharp focus; as long as the circle of confusion is roughly smaller than a pixel on the film sensor, objects appear to be in focus. The range of distances from the lens at which objects appear in focus is called the lens's *depth of field*.

The Gaussian lens equation also lets us compute the size of the circle of confusion; given a lens with focal length  $f$  that is focused at a distance  $z_f$ , the film plane is at  $z'_f$ . Given another point at depth  $z$ , the Gaussian lens equation gives the distance  $z'$  that the lens focuses the point to. This point is either in front of or behind the film plane; Figure 6.11(a) shows the case where it is behind.

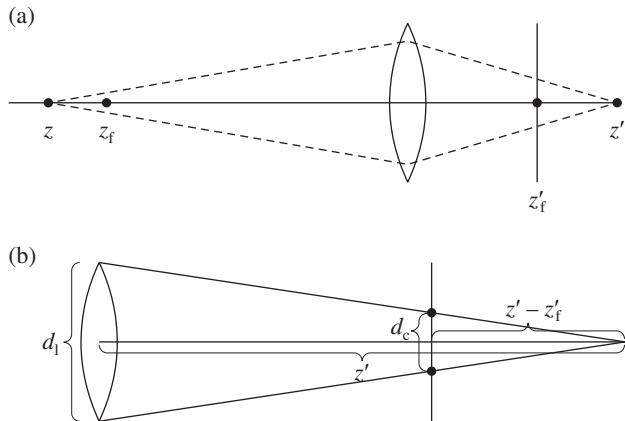
The diameter of the circle of confusion is given by the intersection of the cone between  $z'$  and the lens with the film plane. If we know the diameter of the lens  $d_l$ , then we can use



**Figure 6.9:** As the size of the lens aperture increases, the size of the circle of confusion in the out-of-focus areas increases, giving a greater amount of blur on the film plane.



**Figure 6.10:** Depth of field gives a greater sense of depth and scale to this part of the landscape scene. (*Scene courtesy of Laubwerk*)



**Figure 6.11:** (a) If a thin lens with focal length  $f$  is focused at some depth  $z_f$ , then the distance from the lens to the film plane is  $z'_f$ , given by the Gaussian lens equation. A point in the scene at depth  $z \neq z_f$  will be imaged as a circle on the film plane; here  $z$  focuses at  $z'$ , which is behind the film plane. (b) To compute the diameter of the circle of confusion, we can apply similar triangles: the ratio of  $d_l$ , the diameter of the lens, to  $z'$  must be the same as the ratio of  $d_c$  the diameter of the circle of confusion, to  $z' - z'_f$ .

similar triangles to solve for the diameter of the circle of confusion  $d_c$  (Figure 6.11(b)):

$$\frac{d_l}{z'} = \frac{d_c}{|z' - z'_f|}.$$

Solving for  $d_c$ , we have

$$d_c = \left| \frac{d_l (z' - z'_f)}{z'} \right|.$$

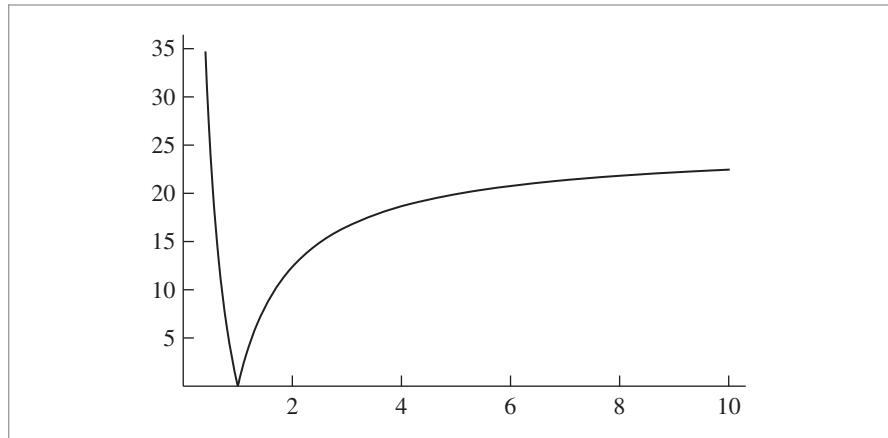
Applying the Gaussian lens equation to express the result in terms of scene depths, we can find that

$$d_c = \left| \frac{d_l f (z - z_f)}{z(f + z_f)} \right|.$$

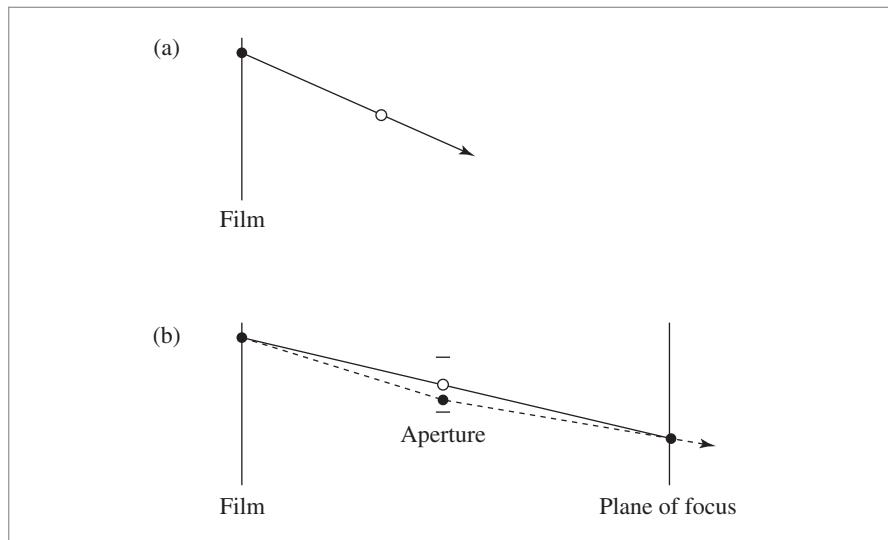
Note that the diameter of the circle of confusion is proportional to the diameter of the lens. The lens diameter is often expressed as the lens's *f-number*  $n$ , which expresses diameter as a fraction of focal length,  $d_l = f/n$ .

Figure 6.12 shows a graph of this function for a 50-mm focal length lens with a 25-mm aperture, focused at  $z_f = 1$  m. Note that the blur is asymmetric with depth around the focal plane and grows much more quickly for objects in front of the plane of focus than for objects behind it.

Modeling a thin lens in a ray tracer is remarkably straightforward: all that is necessary is to choose a point on the lens and find the appropriate ray that starts on the lens at that point such that objects in the plane of focus are in focus on the film (Figure 6.13).



**Figure 6.12:** The diameter of the circle of confusion as a function of depth for a 50-mm focal length lens with 25-mm aperture, focused at 1 meter.



**Figure 6.13:** (a) For a pinhole camera model, a single camera ray is associated with each point on the film plane (filled circle), given by the ray that passes through the single point of the pinhole lens (empty circle). (b) For a camera model with a finite aperture, we sample a point (filled circle) on the disk-shaped lens for each ray. We then compute the ray that passes through the center of the lens (corresponding to the pinhole model) and the point where it intersects the plane of focus (solid line). We know that all objects in the plane of focus must be in focus, regardless of the lens sample position. Therefore, the ray corresponding to the lens position sample (dashed line) is given by the ray starting on the lens sample point and passing through the computed intersection point on the plane of focus.



**Figure 6.14:** Landscape scene with depth of field and only four samples per pixel: the depth of field is undersampled and the image is grainy. (*Scene courtesy of Laubwerk.*)

Therefore, projective cameras take two extra parameters for depth of field: one sets the size of the lens aperture, and the other sets the focal distance.

```
(ProjectiveCamera Protected Data) +≡ 358
    Float lensRadius, focalDistance;
```

```
(Initialize depth of field parameters) ≡ 360
    lensRadius = lensr;
    focalDistance = focald;
```

It is generally necessary to trace many rays for each image pixel in order to adequately sample the lens for smooth depth of field. Figure 6.14 shows the landscape scene from Figure 6.10 with only four samples per pixel (Figure 6.10 had 128 samples per pixel).

```
(Modify ray for depth of field) ≡ 364, 367
    if (lensRadius > 0) {
        (Sample point on lens 374)
        (Compute point on plane of focus 375)
        (Update ray for effect of lens 375)
    }
```

CameraSample 357  
CameraSample::pLens 357  
ConcentricSampleDisk() 778  
Float 1062  
Point2f 68  
ProjectiveCamera::  
 focalDistance  
 374  
ProjectiveCamera::lensRadius  
 374

The ConcentricSampleDisk() function, defined in Chapter 13, takes a  $(u, v)$  sample position in  $[0, 1]^2$  and maps it to a 2D unit disk centered at the origin  $(0, 0)$ . To turn this into a point on the lens, these coordinates are scaled by the lens radius. The CameraSample class provides the  $(u, v)$  lens-sampling parameters in the pLens member variable.

```
(Sample point on lens) ≡ 374
    Point2f pLens = lensRadius * ConcentricSampleDisk(sample.pLens);
```

The ray's origin is this point on the lens. Now it is necessary to determine the proper direction for the new ray. We know that *all* rays from the given image sample through the lens must converge at the same point on the plane of focus. Furthermore, we know that rays pass through the center of the lens without a change in direction, so finding the appropriate point of convergence is a matter of intersecting the unperturbed ray from the pinhole model with the plane of focus and then setting the new ray's direction to be the vector from the point on the lens to the intersection point.

For this simple model, the plane of focus is perpendicular to the  $z$  axis and the ray starts at the origin, so intersecting the ray through the lens center with the plane of focus is straightforward. The  $t$  value of the intersection is given by

$$t = \frac{\text{focalDistance}}{\mathbf{d}_z}.$$

*(Compute point on plane of focus)* ≡ 374

```
Float ft = focalDistance / ray->d.z;
Point3f pFocus = (*ray)(ft);
```

Now the ray can be initialized. The origin is set to the sampled point on the lens, and the direction is set so that the ray passes through the point on the plane of focus, *pFocus*.

*(Update ray for effect of lens)* ≡ 374

```
ray->o = Point3f(pLens.x, pLens.y, 0);
ray->d = Normalize(pFocus - ray->o);
```

To compute ray differentials with the thin lens, the approach used in the fragment *(Update ray for effect of lens)* is applied to rays offset one pixel in the  $x$  and  $y$  directions on the film plane. The fragments that implement this, *(Compute OrthographicCamera ray differentials accounting for lens)* and *(Compute PerspectiveCamera ray differentials accounting for lens)*, aren't included here.

## 6.3 ENVIRONMENT CAMERA

One advantage of ray tracing compared to scan line or rasterization-based rendering methods is that it's easy to employ unusual image projections. We have great freedom in how the image sample positions are mapped into ray directions, since the rendering algorithm doesn't depend on properties such as straight lines in the scene always projecting to straight lines in the image.

In this section, we will describe a camera model that traces rays in all directions around a point in the scene, giving a 2D view of everything that is visible from that point. Consider a sphere around the camera position in the scene; choosing points on that sphere gives directions to trace rays in. If we parameterize the sphere with spherical coordinates, each point on the sphere is associated with a  $(\theta, \phi)$  pair, where  $\theta \in [0, \pi]$  and  $\phi \in [0, 2\pi]$ . (See Section 5.5.2 for more details on spherical coordinates.) This type of image is particularly useful because it represents all of the incident light at a point on the scene. (One important use of this image representation is environment lighting—a rendering technique that uses image-based representations of light in a scene.) Figure 6.15 shows

Float 1062  
 OrthographicCamera 361  
 PerspectiveCamera 365  
 Point3f 68  
 ProjectiveCamera::  
     focalDistance  
     374  
 Ray::d 73  
 Ray::o 73  
 Vector3f::Normalize() 66



**Figure 6.15:** The San Miguel model rendered with the `EnvironmentCamera`, which traces rays in all directions from the camera position. The resulting image gives a representation of all light arriving at that point in the scene and can be used for the image-based lighting techniques described in Chapters 12 and 14.

this camera in action with the San Miguel model.  $\theta$  values range from 0 at the top of the image to  $\pi$  at the bottom of the image, and  $\phi$  values range from 0 to  $2\pi$ , moving from left to right across the image.<sup>1</sup>

```
(EnvironmentCamera Declarations) ≡
class EnvironmentCamera : public Camera {
public:
(EnvironmentCamera Public Methods 376)
};
```

The `EnvironmentCamera` derives directly from the `Camera` class, not the `ProjectiveCamera` class. This is because the environmental projection is nonlinear and cannot be captured by a single  $4 \times 4$  matrix. This camera is defined in the files `cameras/environment.h` and `cameras/environment.cpp`.

```
(EnvironmentCamera Public Methods) ≡ 376
EnvironmentCamera(const AnimatedTransform &CameraToWorld,
                  Float shutterOpen, Float shutterClose, Film *film,
                  const Medium *medium)
: Camera(CameraToWorld, shutterOpen, shutterClose, film, medium) {
```

`AnimatedTransform` 103  
`Camera` 356  
`EnvironmentCamera` 376  
`Film` 484  
`Float` 1062  
`Medium` 684  
`ProjectiveCamera` 358

<sup>1</sup> Readers familiar with cartography will recognize this as an equirectangular projection.

```
<EnvironmentCamera Method Definitions> ≡
    Float EnvironmentCamera::GenerateRay(const CameraSample &sample,
        Ray *ray) const {
    (Compute environment camera ray direction 377)
        *ray = Ray(Point3f(0, 0, 0), dir, Infinity,
            Lerp(sample.time, shutterOpen, shutterClose));
        ray->medium = medium;
        *ray = CameraToWorld(*ray);
        return 1;
}
```

To compute the  $(\theta, \phi)$  coordinates for this ray, NDC coordinates are computed from the raster image sample position and then scaled to cover the  $(\theta, \phi)$  range. Next, the spherical coordinate formula is used to compute the ray direction, and finally the direction is converted to world space. (Note that because the  $y$  direction is “up” in camera space, here the  $y$  and  $z$  coordinates in the spherical coordinate formula are exchanged in comparison to usage elsewhere in the system.)

*<Compute environment camera ray direction> ≡*

377

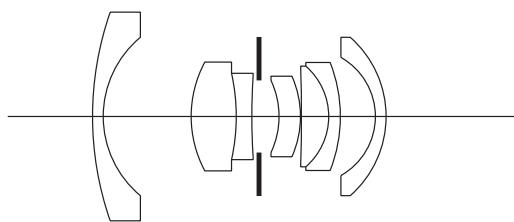
```
    Float theta = Pi * sample.pFilm.y / film->fullResolution.y;
    Float phi = 2 * Pi * sample.pFilm.x / film->fullResolution.x;
    Vector3f dir(std::sin(theta) * std::cos(phi), std::cos(theta),
        std::sin(theta) * std::sin(phi));
```

## \* 6.4 REALISTIC CAMERAS

Camera::CameraToWorld 356  
 Camera::medium 356  
 Camera::shutterClose 356  
 Camera::shutterOpen 356  
 CameraSample 357  
 CameraSample::pFilm 357  
 EnvironmentCamera 376  
 Film::fullResolution 485  
 Float 1062  
 Infinity 210  
 Lerp() 1079  
 Pi 1063  
 Point3f 68  
 Ray 73  
 Ray::medium 74  
 RealisticCamera 378  
 SamplerIntegrator 25  
 Vector3f 60

The thin lens model makes it possible to render images with blur due to depth of field, but it is a fairly rough approximation of actual camera lens systems, which are comprised of a series of multiple *lens elements*, each of which modifies the distribution of radiance passing through it. (Figure 6.16 shows a cross section of a 22-mm focal length wide-angle lens with eight elements.) Even basic cell phone cameras tend to have on the order of five individual lens elements, while DSLR lenses may have ten or more. In general, more complex lens systems with larger numbers of lens elements can create higher quality images than simpler lens systems.

This section discusses the implementation of `RealisticCamera`, which simulates the focusing of light through lens systems like the one in Figure 6.16 to render images like Figure 6.17. Its implementation is based on ray tracing, where the camera follows ray paths through the lens elements, accounting for refraction at the interfaces between media (air, different types of glass) with different indices of refraction, until the ray path either exits the optical system or until it is absorbed by the aperture stop or lens housing. Rays leaving the front lens element represent samples of the camera’s response profile and can be used with integrators that estimate the incident radiance along arbitrary rays, such as the `SamplerIntegrator`. The `RealisticCamera` implementation is in the files `cameras/realistic.h` and `cameras/realistic.cpp`.



**Figure 6.16:** Cross section of a wide-angle lens system (scenes/lenses/wide.22.dat in the pbrt distribution). The lens coordinate system has the film plane perpendicular to the  $z$  axis and located at  $z = 0$ . The lenses are to the left, along negative  $z$ , and then the scene is to the left of the lenses. The aperture stop, indicated by the thick black lines in the middle of the lens system, blocks rays that hit it. In many lens systems, the size of the aperture stop can be adjusted to trade off between shorter exposure times (with larger apertures) and more depth of field (with smaller apertures).



**Figure 6.17:** Image rendered using a fish-eye lens with a very wide field of view. Note the darkening at the edges, which is due to accurate simulation of the radiometry of image formation (Section 6.4.7) and the distortion of straight lines to curves, which is characteristic of many wide-angle lenses but isn't accounted for when using projection matrices to represent the lens projection model.

```

⟨RealisticCamera Declarations⟩ ≡
class RealisticCamera : public Camera {
public:
    ⟨RealisticCamera Public Methods⟩
private:
    ⟨RealisticCamera Private Declarations 381⟩
    ⟨RealisticCamera Private Data 379⟩
    ⟨RealisticCamera Private Methods 381⟩
};
```

Camera 356

In addition to the usual transformation to place the camera in the scene, the `Film`, and the shutter open and close times, the `RealisticCamera` constructor takes a filename for a lens system description file, the distance to the desired plane of focus, and a diameter for the aperture stop. The effect of the `simpleWeighting` parameter is described later, in Section 13.6.6, after preliminaries related to Monte Carlo integration in Chapter 13 and the radiometry of image formation in Section 6.4.7.

*(RealisticCamera Method Definitions) ≡*

```
RealisticCamera::RealisticCamera(const AnimatedTransform &CameraToWorld,
    Float shutterOpen, Float shutterClose, Float apertureDiameter,
    Float focusDistance, bool simpleWeighting, const char *lensFile,
    Film *film, const Medium *medium)
: Camera(CameraToWorld, shutterOpen, shutterClose, film, medium),
  simpleWeighting(simpleWeighting) {
    <Load element data from lens description file>
    <Compute lens–film distance for given focus distance 389>
    <Compute exit pupil bounds at sampled points on the film 390>
}
```

*(RealisticCamera Private Data) ≡*

378

```
const bool simpleWeighting;
```

After loading the lens description file from disk, the constructor adjusts the spacing between the lenses and the film so that the plane of focus is at the desired depth, `focusDistance`, and then precomputes some information about which areas of the lens element closest to the film carry light from the scene to the film, as seen from various points on the film plane. After background material has been introduced, the fragments *(Compute lens–film distance for given focus distance)* and *(Compute exit pupil bounds at sampled points on the film)* will be defined in Sections 6.4.4 and 6.4.5, respectively.

### 6.4.1 LENS SYSTEM REPRESENTATION

A lens system is made from a series of lens elements, where each element is generally some form of glass. A lens system designer’s challenge is to design a series of elements that form high-quality images on a film or sensor subject to limitations of space (e.g., the thickness of mobile phone cameras is very limited in order to keep phones thin), cost, and ease of manufacture.

It’s easiest to manufacture lenses with cross sections that are spherical, and lens systems are generally symmetric around the *optical axis*, which is conventionally denoted by  $z$ . We will assume both of these properties in the remainder of this section. As in Section 6.2.3, lens systems are defined using a coordinate system where the film is aligned with the  $z = 0$  plane and lenses are to the left of the film, along the  $-z$  axis.

Lens systems are commonly represented in terms of the series of interfaces between the individual lens elements (or air) rather than having an explicit representation of each element. Table 6.1 shows the quantities that define each interface. The last entry in the table defines the rightmost interface, which is shown in Figure 6.18: it’s a section of a sphere with radius equal to the curvature radius. The thickness of an element is the

`AnimatedTransform` 103

`Camera` 356

`Film` 484

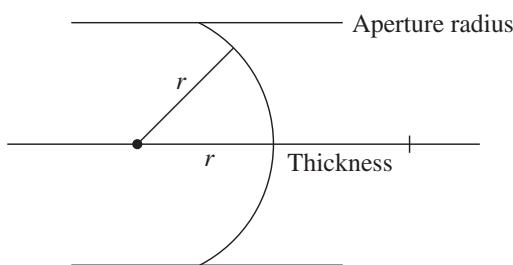
`Float` 1062

`Medium` 684

`RealisticCamera` 378

**Table 6.1: Tabular description of the lens system in Figure 6.16.** Each line describes the interface between two lens elements, the interface between an element and air, or the aperture stop. The first line describes the leftmost interface. The element with radius 0 corresponds to the aperture stop. Distances are measured in mm.

Curvature Radius	Thickness	Index of Refraction	Aperture Diameter
35.98738	1.21638	1.54	23.716
11.69718	9.9957	1	17.996
13.08714	5.12622	1.772	12.364
-22.63294	1.76924	1.617	9.812
71.05802	0.8184	1	9.152
0	2.27766	0	8.756
-9.58584	2.43254	1.617	8.184
-11.28864	0.11506	1	9.152
-166.7765	3.09606	1.713	10.648
-7.5911	1.32682	1.805	11.44
-16.7662	3.98068	1	12.276
-7.70286	1.21638	1.617	13.42
-11.97328	(depends on focus)	1	17.996



**Figure 6.18:** A lens interface (solid curved line) intersecting the optical axis at a position  $z$ . The interface geometry is described by the interface's aperture radius, which describes its extent above and below the optical axis, and the element's curvature radius  $r$ . If the element has a spherical cross section, then its profile is given by a sphere with center a distance  $r$  away on the optical axis, where the sphere also passes through  $z$ . If  $r$  is negative, the element interface is concave as seen from the scene (as is shown here); otherwise it is convex. The thickness of the lens gives the distance to the next interface to the right, or the distance to the film plane for the rearmost interface.

distance along  $z$  to the next element to the right (or to the film plane), and the index of refraction is for the medium to the right of the interface. The element's extent above and below the  $z$  axis is set by the aperture diameter.

The `LensElementInterface` structure represents a single lens element interface.

*(RealisticCamera Private Declarations)*  $\equiv$

```
struct LensElementInterface {
    float curvatureRadius;
    float thickness;
    float eta;
    float apertureRadius;
};
```

378

The fragment *(Load element data from lens description file)*, not included here, reads the lens elements and initializes the `RealisticCamera::elementInterfaces` array. See comments in the source code for details of the file format, which parallels the structure of Table 6.1, and see the directory `scenes/lenses` in the `pbrt` distribution for a number of example lens descriptions.

Two adjustments are made to the values read from the file: first, lens systems are traditionally described in units of millimeters, but `pbrt` assumes a scene measured in meters. Therefore, the fields other than the index of refraction are scaled by 1/1000. Second, the element's diameter is divided by two; the radius is a more convenient quantity to have at hand in the code to follow.

*(RealisticCamera Private Data)*  $+ \equiv$

```
std::vector<LensElementInterface> elementInterfaces;
```

378

Once the element interface descriptions have been loaded, it's useful to have a few values related to the lens system easily at hand. `LensRearZ()` and `LensFrontZ()` return the  $z$  depths of the rear and front elements of the lens system, respectively. Note that the returned  $z$  depths are in camera space, not lens space, and thus have positive values.

*(RealisticCamera Private Methods)*  $\equiv$

```
float LensRearZ() const {
    return elementInterfaces.back().thickness;
}
```

378

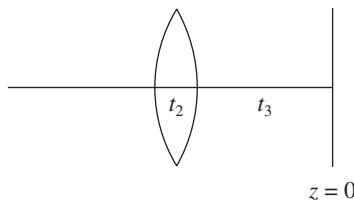
Finding the front element's  $z$  position requires summing all of the element thicknesses (see Figure 6.19). This value isn't needed in any code that is in a performance-sensitive part of the system, so recomputing it when needed is fine. If performance of this method was a concern, it would be better to cache this value in the `RealisticCamera`.

*(RealisticCamera Private Methods)*  $+ \equiv$

```
float LensFrontZ() const {
    float zSum = 0;
    for (const LensElementInterface &element : elementInterfaces)
        zSum += element.thickness;
    return zSum;
}
```

378

```
float 1062
LensElementInterface 381
LensElementInterface::
    thickness
    381
RealisticCamera::
    elementInterfaces
    381
```



**Figure 6.19: The Relationship between Element Thickness and Position on the Optical Axis.**

The film plane is at  $z = 0$ , and the rear element's thickness,  $t_3$ , gives the distance to its interface from the film; the rear interface intersects the axis here at  $z = -t_3$ . The next element has a thickness  $t_2$  and is positioned at  $z = -t_3 - t_2$ , and so forth. The front element intersects the  $z$  axis at  $\sum_i -t_i$ .

`RearElementRadius()` returns the aperture radius of the rear element in meters.

*(RealisticCamera Private Methods)*  $\equiv$

378

```
Float RearElementRadius() const {
    return elementInterfaces.back().apertureRadius;
}
```

#### 6.4.2 TRACING RAYS THROUGH LENSES

Given a ray starting from the film side of the lens system, `TraceLensesFromFilm()` computes intersections with each element in turn, terminating the ray and returning `false` if its path is blocked along the way through the lens system. Otherwise it returns `true` and initializes `*rOut` with the exiting ray in camera space. During traversal, `elementZ` tracks the  $z$  intercept of the current lens element. Because the ray is starting from the film, the lenses are traversed in reverse order compared to how they are stored in `elementInterfaces`.

*(RealisticCamera Method Definitions)*  $\equiv$

```
bool RealisticCamera::TraceLensesFromFilm(const Ray &rCamera,
    Ray *rOut) const {
    Float elementZ = 0;
    (Transform rCamera from camera to lens system space 383)
    for (int i = elementInterfaces.size() - 1; i >= 0; --i) {
        const LensElementInterface &element = elementInterfaces[i];
        (Update ray from film accounting for interaction with element 383)
    }
    (Transform rLens from lens system space back to camera space 385)
    return true;
}
```

Float 1062

LensElementInterface 381

LensElementInterface::  
    apertureRadius  
    381

Ray 73

RealisticCamera::  
    elementInterfaces  
    381

Transform 83

Because the camera points down the  $+z$  axis in pbrt's camera space but lenses are along  $-z$ , the  $z$  components of the origin and direction of the ray need to be negated. While this is a simple enough transformation that it could be applied directly, we prefer an explicit `Transform` to make the intent clear.

```
(Transform rCamera from camera to lens system space) ≡ 382
    static const Transform CameraToLens = Scale(1, 1, -1);
    Ray rLens = CameraToLens(rCamera);
```

Recall from Figure 6.19 how the  $z$  intercept of elements is computed: because we are visiting the elements from back-to-front, the element's thickness must be subtracted from `elementZ` to compute its  $z$  intercept before the element interaction is accounted for.

```
(Update ray from film accounting for interaction with element) ≡ 382
    elementZ -= element.thickness;
(Compute intersection of ray with lens element 383)
(Test intersection point against element aperture 384)
(Update ray path for element interface interaction 385)
```

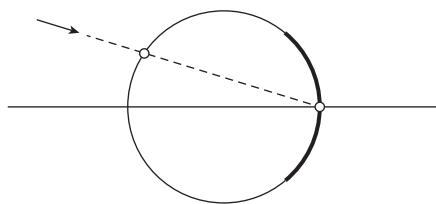
Given the element's  $z$  axis intercept, the next step is to compute the parametric  $t$  value along the ray where it intersects the element interface (or the plane of the aperture stop). For the aperture stop, a ray–plane test (following Section 3.1.2) is used. For spherical interfaces, `IntersectSphericalElement()` performs this test and also returns the surface normal if an intersection is found; the normal will be needed for computing the refracted ray direction.

```
(Compute intersection of ray with lens element) ≡ 383
    Float t;
    Normal3f n;
    bool isStop = (element.curvatureRadius == 0);
    if (isStop)
        t = (elementZ - rLens.o.z) / rLens.d.z;
    else {
        Float radius = element.curvatureRadius;
        Float zCenter = elementZ + element.curvatureRadius;
        if (!IntersectSphericalElement(radius, zCenter, rLens, &t, &n))
            return false;
    }

    Float 1062
    LensElementInterface::
    curvatureRadius
    381
    LensElementInterface::
    thickness
    381
    Normal3f 71
    Ray 73
    Ray::d 73
    Ray::o 73
    RealisticCamera::
    IntersectSphericalElement
    383
    Scale() 87
    Sphere::Intersect() 134
    Transform 83
```

The `IntersectSphericalElement()` method is generally similar to `Sphere::Intersect()`, though it's specialized for the fact that the element's center is along the  $z$  axis (and thus, the center's  $x$  and  $y$  components are zero). The fragments *(Compute  $t_0$  and  $t_1$  for ray–element intersection)* and *(Compute surface normal of element at ray intersection point)* aren't included in the text here due to their similarity with the `Sphere::Intersect()` implementation.

```
(RealisticCamera Method Definitions) +≡
    bool RealisticCamera::IntersectSphericalElement(Float radius,
                                                    Float zCenter, const Ray &ray, Float *t, Normal3f *n) {
        (Compute  $t_0$  and  $t_1$  for ray–element intersection)
        (Select intersection  $t$  based on ray direction and element curvature 384)
        (Compute surface normal of element at ray intersection point)
        return true;
    }
```



**Figure 6.20:** When computing the intersection of a ray with a spherical lens element, the first intersection of the ray with the full sphere isn't necessarily the desired one. Here, the second intersection is the one on the actual element interface (thick line) and the first should be ignored.

There is, however, a subtlety in choosing which intersection point to return: the closest intersection with  $t > 0$  isn't necessarily on the element interface; see Figure 6.20.<sup>2</sup> For example, for a ray approaching from the scene and intersecting a concave lens (with negative curvature radius), the farther of the two intersections should be returned regardless of whether the closer one has  $t > 0$ . Fortunately, simple logic based on the ray direction and the curvature radius indicates which  $t$  value to use.

*(Select intersection t based on ray direction and element curvature)* ≡

383

```
bool useCloserT = (ray.d.z > 0) ^ (radius < 0);
*t = useCloserT ? std::min(t0, t1) : std::max(t0, t1);
if (*t < 0)
    return false;
```

Each lens element extends for some radius around the optical axis; if the intersection point with the element is outside this radius, then the ray will actually intersect the lens housing and terminate. In a similar fashion, if a ray intersects the aperture stop, it also terminates. Therefore, here we test the intersection point against the appropriate limit for the current element, either terminating the ray or updating its origin to the current intersection point if it survives.

*(Test intersection point against element aperture)* ≡

383

```
Point3f pHit = rLens(t);
Float r2 = pHit.x * pHit.x + pHit.y * pHit.y;
if (r2 > element.apertureRadius * element.apertureRadius)
    return false;
rLens.o = pHit;
```

If the current element is the aperture, the ray's path isn't affected by traveling through the element's interface. For glass (or, forbid, plastic) lens elements, the ray's direction changes at the interface as it goes from a medium with one index of refraction to one with another. (The ray may be passing from air to glass, from glass to air, or from glass with one index of refraction to a different type of glass with a different index of refraction.)

Float 1062

LensElementInterface::  
apertureRadius  
381

Point3f 68

Ray::o 73

<sup>2</sup> As usual, “subtlety” means “the authors spent a number of hours debugging this.”

Section 8.2 discusses how a change in index of refraction at the boundary between two media changes the direction of a ray and the amount of radiance carried by the ray. (In this case, we can ignore the change of radiance, as it cancels out if the ray is in the same medium going into the lens system as it is when it exits—here, both are air.) The `Refract()` function is defined in Section 8.2.3; note that it expects that the incident direction will point away from the surface, so the ray direction is negated before being passed to it. This function returns `false` in the presence of total internal reflection, in which case the ray path terminates. Otherwise, the refracted direction is returned in `w`.

In general, some light passing through an interface like this is transmitted and some is reflected. Here we ignore reflection and assume perfect transmission. Though an approximation, it is a reasonable one: lenses are generally manufactured with coatings designed to reduce the reflection to around 0.25% of the radiance carried by the ray. (However, modeling this small amount of reflection can be important for capturing effects like lens flare.)

```
(Update ray path for element interface interaction) ≡ 383
    if (!isStop) {
        Vector3f w;
        Float etaI = element.eta;
        Float etaT = (i > 0 && elementInterfaces[i - 1].eta != 0) ?
            elementInterfaces[i - 1].eta : 1;
        if (!Refract(Normalize(-rLens.d), n, etaI / etaT, &w))
            return false;
        rLens.d = w;
    }
```

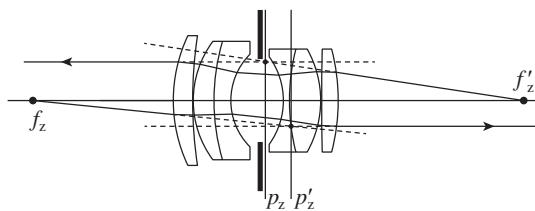
If the ray has successfully made it out of the front lens element, it just needs to be transformed from lens space to camera space.

```
(Transform rLens from lens system space back to camera space) ≡ 382
    if (rOut != nullptr) {
        static const Transform LensToCamera = Scale(1, 1, -1);
        *rOut = LensToCamera(rLens);
    }
```

Float 1062  
 LensElementInterface::eta 381  
 Ray 73  
 Ray::d 73  
 RealisticCamera::elementInterfaces 381  
 Refract() 531  
 Scale() 87  
 Transform 83  
 Vector3::Normalize() 66  
 Vector3f 60

The `TraceLensesFromScene()` method is quite similar to `TraceLensesFromFilm()` and isn't included here. The main differences are that it traverses the elements from front-to-back rather than back-to-front. Note that it assumes that the ray passed to it is already in camera space; the caller is responsible for performing the transformation if the ray is starting from world space. The returned ray is in camera space, leaving the rear lens element toward the film.

```
(RealisticCamera Private Methods) +≡ 378
    bool TraceLensesFromScene(const Ray &rCamera, Ray *rOut) const;
```



**Figure 6.21: Computing the Cardinal Points of a Lens System.** The lens system described in the file `tenses/dgauss.dat` with an incident ray from the scene parallel to the optical axis (above the axis), and a ray from the film parallel to the optical axis (below). The intersections with the optical axis of the rays leaving the lens system due to these incident rays give the two focal points,  $f'_z$  (on the film side) and  $f_z$  (on the scene side). The principal planes  $z = p_z$  and  $z = p'_z$  are given by the intersection of the extension of each pair of incident and exiting rays with the original rays and are shown here as planes perpendicular to the axis.

### 6.4.3 THE THICK LENS APPROXIMATION

The thin lens approximation used in Section 6.2.3 was based on the simplifying assumption that the lens system had 0 thickness along the optical axis. The thick lens approximation of a lens system is slightly more accurate in that it accounts for the lens system's  $z$  extent. After introducing the basic concepts of the thick lenses here, we'll use the thick lens approximation to determine how far to place the lens system from the film in order to focus at the desired focal depth in Section 6.4.4.

The thick lens approximation represents a lens system by two pairs of distances along the optical axis—the *focal points* and the depths of the *principal planes*; these are two of the *cardinal points* of a lens system. If rays parallel to the optical axis are traced through an ideal lens system, all of the rays will intersect the optical axis at the same point—this is the focal point. (In practice, real lens systems aren't perfectly ideal and incident rays at different heights will intersect the optical axis along a small range of  $z$  values—this is the *spherical aberration*.) Given a specific lens system, we can trace rays parallel to the optical axis through it from each side and compute their intersections with the  $z$  axis to find the focal points. (See Figure 6.21.)

Each principal plane is found by extending the incident ray parallel to the optical axis and the ray leaving the lens until they intersect; the  $z$  depth of the intersection gives the depth of the corresponding principal plane. Figure 6.21 shows a lens system with its focal points  $f_z$  and  $f'_z$  and principal planes at  $z$  values  $p_z$  and  $p'_z$ . (As in Section 6.2.3, primed variables represent points on the film side of the lens system, and unprimed variables represent points in the scene being imaged.)

Given the ray leaving the lens, finding the focal point requires first computing the  $t_f$  value where the ray's  $x$  and  $y$  components are zero. If the entering ray was offset from the optical axis only along  $x$ , then we'd like to find  $t_f$  such that  $\mathbf{o}_x + t_f \mathbf{d}_x = \mathbf{0}$ . Thus,

$$t_f = -\mathbf{o}_x / \mathbf{d}_x.$$

In a similar manner, to find the  $t_p$  for the principal plane where the ray leaving the lens has the same  $x$  height as the original ray, we have  $\mathbf{o}_x + t_p \mathbf{d}_x = \mathbf{x}$ , and so

$$t_p = (x - o_x) / d_x.$$

Once these two  $t$  values have been computed, the ray equation can be used to find the  $z$  coordinates of the corresponding points.

The `ComputeCardinalPoints()` method computes the  $z$  depths of the focal point and the principal plane for the given rays. Note that it assumes that the rays are in camera space but returns  $z$  values along the optical axis in lens space.

*(RealisticCamera Method Definitions)* +≡

```
void RealisticCamera::ComputeCardinalPoints(const Ray &rIn,
                                             const Ray &rOut, Float *pz, Float *fz) {
    Float tf = -rOut.o.x / rOut.d.x;
    *fz = -rOut(tf).z;
    Float tp = (rIn.o.x - rOut.o.x) / rOut.d.x;
    *pz = -rOut(tp).z;
}
```

The `ComputeThickLensApproximation()` method computes both pairs of cardinal points for the lens system.

*(RealisticCamera Method Definitions)* +≡

```
void RealisticCamera::ComputeThickLensApproximation(Float pz[2],
                                                    Float fz[2]) const {
    (Find height x from optical axis for parallel rays 387)
    (Compute cardinal points for film side of lens system 387)
    (Compute cardinal points for scene side of lens system 388)
}
```

First, we must choose a height along the  $x$  axis for the rays to be traced. It should be far enough from  $x = 0$  so that there is sufficient numeric precision to accurately compute where rays leaving the lens system intersect the  $z$  axis, but not so high up the  $x$  axis that it hits the aperture stop on the ray through the lens system. Here, we use a small fraction of the film's diagonal extent; this works well unless the aperture stop is extremely small.

*(Find height x from optical axis for parallel rays)* ≡

387

```
Float x = .001 * film->diagonal;
```

To construct the ray from the scene entering the lens system `rScene`, we offset a bit from the front of the lens. (Recall that the ray passed to `TraceLensesFromScene()` should be in camera space.)

*(Compute cardinal points for film side of lens system)* ≡

387

```
Ray rScene(Point3f(x, 0, LensFrontZ() + 1), Vector3f(0, 0, -1));
Ray rFilm;
TraceLensesFromScene(rScene, &rFilm);
ComputeCardinalPoints(rScene, rFilm, &pz[0], &fz[0]);
```

An equivalent process starting from the film side of the lens system gives us the other two cardinal points.

```
Camera::film 356
Film::diagonal 485
Float 1062
Point3f 68
Ray 73
RealisticCamera::
    ComputeCardinalPoints()
    387
RealisticCamera::LensFrontZ()
    381
RealisticCamera::
    TraceLensesFromScene()
    385
Vector3f 60
```

```
(Compute cardinal points for scene side of lens system) ≡ 387
rFilm = Ray(Point3f(x, 0, LensRearZ() - 1), Vector3f(0, 0, 1));
TraceLensesFromFilm(rFilm, &rScene);
ComputeCardinalPoints(rFilm, rScene, &pz[1], &fz[1]);
```

#### 6.4.4 FOCUSING

Lens systems can be focused at a given depth in the scene by moving the system in relation to the film so that a point at the desired focus depth images to a point on the film plane. The Gaussian lens equation, (6.3), gives us a relation that we can solve to focus a thick lens.

For thick lenses, the Gaussian lens equation relates distances from a point in the scene at  $z$  and the point it focuses to  $z'$  by

$$\frac{1}{z' - p'_z} - \frac{1}{z - p_z} = \frac{1}{f}. \quad (6.3)$$

For thin lenses,  $p_z = p'_z = 0$ , and Equation (6.1) follows.

If we know the positions  $p_z$  and  $p'_z$  of the principal planes and the focal length of the lens  $f$  and would like to focus at some depth  $z$  along the optical axis, then we need to determine how far to translate the system  $\delta$  so that

$$\frac{1}{z' - p'_z + \delta} - \frac{1}{z - p_z + \delta} = \frac{1}{f}.$$

The focal point on the film side should be at the film, so  $z' = 0$ , and  $z = z_f$ , the given focus depth. The only unknown is  $\delta$ , and some algebraic manipulation gives us

$$\delta = \frac{1}{2} \left( p_z - z_f + p'_z - \sqrt{(p_z - z_f - z')(z - z_f - 4f - p'_z)} \right). \quad (6.4)$$

(There are actually two solutions, but this one, which is the closer of the two, gives a small adjustment to the lens position and is thus the appropriate one.)

`FocusThickLens()` focuses the lens system using this approximation. After computing  $\delta$ , it returns the offset along the  $z$  axis from the film where the lens system should be placed.

```
(RealisticCamera Method Definitions) +≡
Float RealisticCamera::FocusThickLens(Float focusDistance) {
    Float pz[2], fz[2];
    ComputeThickLensApproximation(pz, fz);
    (Compute translation of lens, delta, to focus at focusDistance 389)
    return elementInterfaces.back().thickness + delta;
}
```

Equation (6.4) gives the offset  $\delta$ . The focal length of the lens  $f$  is the distance between the cardinal points  $f'_z$  and  $p'_z$ . Note also that the negation of the focus distance is used for  $z$ , since the optical axis points along negative  $z$ .

```
Float 1062
LensElementInterface::
    thickness
    381
Point3f 68
Ray 73
RealisticCamera::
    ComputeCardinalPoints()
    387
RealisticCamera::
    ComputeThickLens
    Approximation()
    387
RealisticCamera::
    elementInterfaces
    381
RealisticCamera::LensRearZ()
    381
RealisticCamera::
    TraceLensesFromFilm()
    382
Vector3f 60
```

```
<Compute translation of lens, delta, to focus at focusDistance> ≡ 388
    Float f = fz[0] - pz[0];
    Float z = -focusDistance;
    Float delta = 0.5f * (pz[1] - z + pz[0] -
        std::sqrt((pz[1] - z - pz[0]) * (pz[1] - z - 4 * f - pz[0])));
```

We can now finally implement the fragment in the `RealisticCamera` constructor that focuses the lens system. (Recall that the thickness of the rearmost element interface is the distance from the interface to the film.)

```
<Compute lens-film distance for given focus distance> ≡ 379
    elementInterfaces.back().thickness = FocusThickLens(focusDistance);
```

#### 6.4.5 THE EXIT PUPIL

From a given point on the film plane, not all rays toward the rear lens element will successfully exit the lens system; some will be blocked by the aperture stop or will intersect the lens system enclosure. In turn, not all points on the rear lens element transmit radiance to the point on the film. The set of points on the rear element that do carry light through the lens system is called the *exit pupil*; its size and position vary across viewpoints on the film plane. (Analogously, the entrance pupil is the area over the front lens element where rays from a given point in the scene will reach the film.)

Figure 6.22 shows the exit pupil as seen from two points on the film plane with a wide angle lens. The exit pupil gets smaller for points toward the edges of the film. An implication of this shrinkage is vignetting.

When tracing rays starting from the film, we'd like to avoid tracing too many rays that don't make it through the lens system; therefore, it's worth limiting sampling to the exit pupil itself and a small area around it rather than, for example, wastefully sampling the entire area of the rear lens element.

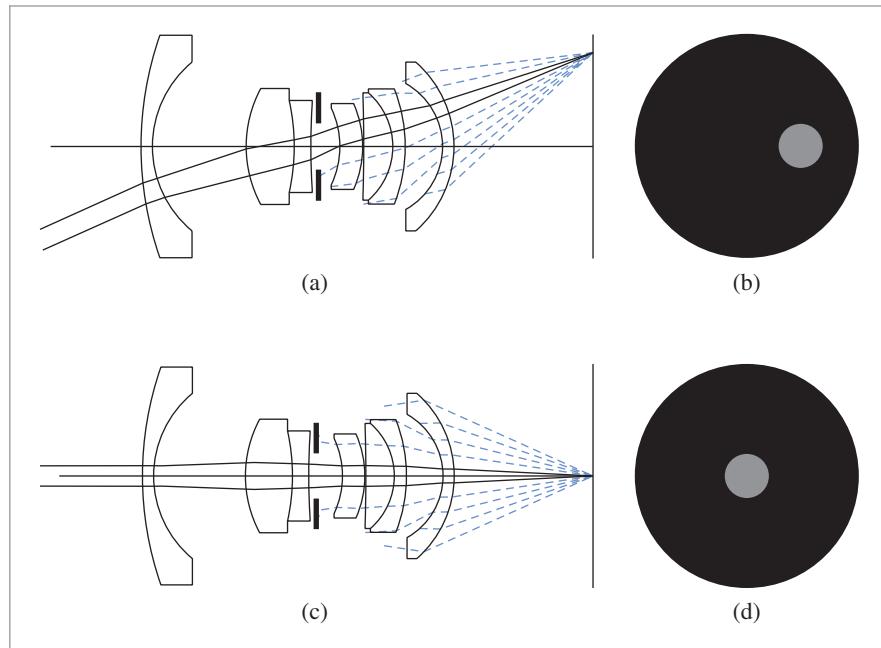
Computing the exit pupil at each point on the film plane before tracing a ray would be prohibitively expensive; instead the `RealisticCamera` implementation precomputes exit pupil bounds along segments of a line on the film plane. Since we assumed that the lens system is radially symmetric around the optical axis, exit pupil bounds will also be radially symmetric, and bounds for arbitrary points on the film plane can be found by rotating these segment bounds appropriately (Figure 6.23). These bounds are then used to efficiently find exit pupil bounds for specific film sample positions.

```
Float 1062
LensElementInterface::thickness 381
RealisticCamera::elementInterfaces 381
RealisticCamera::FocusThickLens() 388
```

One important subtlety to be aware of is that because the lens system is focused by translating it along the optical axis, the shape and position of the exit pupil change when the focus of the lens system is adjusted. Therefore, it's critical that these bounds be computed after focusing.<sup>3</sup>

---

<sup>3</sup> See footnote 2.



**Figure 6.22: The Exit Pupil for a 22-mm-Wide Angle Lens with a 5.5-mm Aperture (f/4).** (a) Rays from a point at the edge of the film plane entering the rear lens element at various points. Dashed lines indicate rays that are blocked and don't exit the lens system. (b) Image of the exit pupil as seen from the vantage point in (a). The rear lens element is black, while the exit pupil is shown in gray. (c) At the center of the film, a different region of the exit pupil allows rays out into the scene. (d) Image of the exit pupil as seen from the center of the film.

*(Compute exit pupil bounds at sampled points on the film)* ≡

379

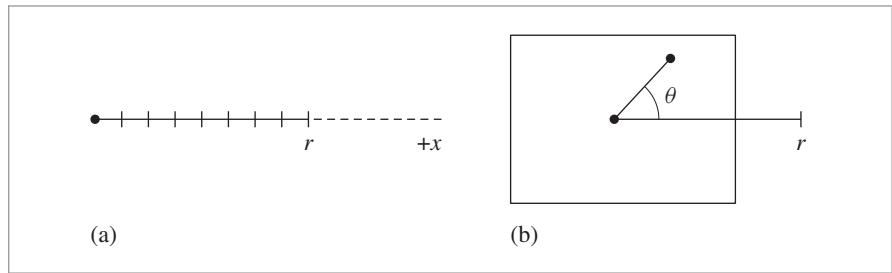
```
int nSamples = 64;
exitPupilBounds.resize(nSamples);
ParallelFor(
    [&](int i) {
        Float r0 = (Float)i / nSamples * film->diagonal / 2;
        Float r1 = (Float)(i + 1) / nSamples * film->diagonal / 2;
        exitPupilBounds[i] = BoundExitPupil(r0, r1);
    }, nSamples);
```

*(RealisticCamera Private Data)* +≡
   
std::vector<Bounds2f> exitPupilBounds;

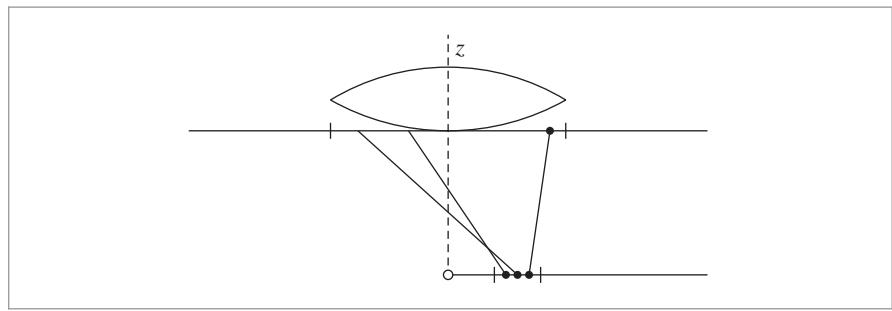
378

The `BoundExitPupil()` method computes a 2D bounding box of the exit pupil as seen from a point along a segment on the film plane. The bounding box is computed by attempting to trace rays through the lens system at a set of points on a plane tangent to the rear lens element. The bounding box of the rays that make it through the lens system gives an approximate bound on the exit pupil—see Figure 6.24.

Bounds2f 76  
 Camera::film 356  
 Film::diagonal 485  
 Float 1062  
 ParallelFor() 1088  
 RealisticCamera::  
 BoundExitPupil()  
 391  
 RealisticCamera::  
 exitPupilBounds  
 390



**Figure 6.23: Precomputing Exit Pupil Bounds.** (a) The `RealisticCamera` computes bounds of the exit pupil at a series of segments along the  $x$  axis of the film plane, up to the distance  $r$  from the center of the film to a corner. (b) Due to the assumption of radial symmetry, we can find exit pupil bounds for an arbitrary point on the film (solid dot) by computing the angle  $\theta$  between the point and the  $x$  axis. If a point is sampled in the original exit pupil bounds and is then rotated by  $-\theta$ , we have a point in the exit pupil bounds at the original point.



**Figure 6.24: 2D Illustration of How Exit Pupil Bounds Are Computed.** `BoundExitPupil()` takes an interval along the  $x$  axis on the film. It samples a series of points along the interval (bottom of the figure). For each point, it also samples a point on the bounding box of the rear lens element's extent on the plane tangent to its rear. It computes the bounding box on the tangent plane of all of the rays that make it through the lens system from points along the interval.

*(RealisticCamera Method Definitions)*  $+ \equiv$

```
Bounds2f RealisticCamera::BoundExitPupil(Float pFilmX0,
                                         Float pFilmX1) const {
    Bounds2f pupilBounds;
    <Sample a collection of points on the rear lens to find exit pupil 392>
    <Return entire element bounds if no rays made it through the lens system 393>
    <Expand bounds to account for sample spacing 393>
    return pupilBounds;
}
```

Bounds2f 76

Float 1062

The implementation samples the exit pupil fairly densely—at a total of  $1024^2$  points for each segment. We've found this sampling rate to provide good exit pupil bounds in practice.

```
(Sample a collection of points on the rear lens to find exit pupil) ≡ 391
const int nSamples = 1024 * 1024;
int nExitingRays = 0;
(Compute bounding box of projection of rear element on sampling plane 392)
for (int i = 0; i < nSamples; ++i) {
    (Find location of sample points on x segment and rear lens element 392)
    (Expand pupil bounds if ray makes it through the lens system 392)
}
```

The bounding box of the rear element in the plane perpendicular to it is not enough to be a conservative bound of the projection of the exit pupil on that plane; because the element is generally curved, rays that pass through the plane outside of that bound may themselves intersect the valid extent of the rear lens element. Rather than compute a precise bound, we'll increase the bounds substantially. The result is that many of the samples taken to compute the exit pupil bound will be wasted; in practice, this is a minor price to pay, as these samples are generally quickly terminated during the lens ray-tracing phase.

```
(Compute bounding box of projection of rear element on sampling plane) ≡ 392
Float rearRadius = RearElementRadius();
Bounds2f projRearBounds(Point2f(-1.5f * rearRadius, -1.5f * rearRadius),
                        Point2f( 1.5f * rearRadius, 1.5f * rearRadius));
```

The  $x$  sample point on the film is found by linearly interpolating between the  $x$  interval endpoints. The `RadicalInverse()` function that is used to compute the interpolation offsets for the sample point inside the exit pupil bounding box will be defined later, in Section 7.4.1. There, we will see that the sampling strategy implemented here corresponds to using Hammersley points in 3D; the resulting point set minimizes gaps in the coverage of the overall 3D domain, which in turn ensures an accurate exit pupil bound estimate.

```
(Find location of sample points on x segment and rear lens element) ≡ 392
Point3f pFilm(Lerp((i + 0.5f) / nSamples, pFilmX0, pFilmX1), 0, 0);
Float u[2] = { RadicalInverse(0, i), RadicalInverse(1, i) };
Point3f pRear(Lerp(u[0], projRearBounds.pMin.x, projRearBounds.pMax.x),
              Lerp(u[1], projRearBounds.pMin.y, projRearBounds.pMax.y),
              LensRearZ());
```

Bounds2::Inside() 79

Bounds2::Union() 78

Bounds2f 76

Float 1062

Lerp() 1079

Point2f 68

Point3f 68

RadicalInverse() 444

Ray 73

RealisticCamera::LensRearZ()
 381

RealisticCamera::
 RearElementRadius()
 382

RealisticCamera::
 TraceLensesFromFilm()
 382

Now we can construct a ray from `pFilm` to `pRear` and determine if it is within the exit pupil by seeing if it makes it out of the front of the lens system. If so, the exit pupil bounds are expanded to include this point. If the sampled point is already inside the exit pupil's bounding box as computed so far, then we can skip the lens ray tracing step to save a bit of unnecessary work.

```
(Expand pupil bounds if ray makes it through the lens system) ≡ 392
if (Inside(Point2f(pRear.x, pRear.y), pupilBounds) ||
    TraceLensesFromFilm(Ray(pFilm, pRear - pFilm), nullptr)) {
    pupilBounds = Union(pupilBounds, Point2f(pRear.x, pRear.y));
    ++nExitingRays;
}
```

It may be that none of the sample rays makes it through the lens system; this case can legitimately happen with some very wide-angle lenses where the exit pupil vanishes at the edges of the film extent, for example. In this case, the bound doesn't matter and `BoundExitPupil()` returns the bound that encompasses the entire rear lens element.

```
(Return entire element bounds if no rays made it through the lens system) ≡ 391
if (nExitingRays == 0)
    return projRearBounds;
```

While one sample may have made it through the lens system and one of its neighboring samples didn't, it may well be that another sample very close to the neighbor actually would have made it out. Therefore, the final bound is expanded by roughly the spacing between samples in each direction in order to account for this uncertainty.

```
(Expand bounds to account for sample spacing) ≡ 391
pupilBounds = Expand(pupilBounds,
                      2 * projRearBounds.Diagonal().Length() /
                      std::sqrt(nSamples));
```

Given the precomputed bounds stored in `RealisticCamera::exitPupilBounds`, the `SampleExitPupil()` method can fairly efficiently find the bounds on the exit pupil for a given point on the film plane. It then samples a point inside this bounding box for the ray from the film to pass through. In order to accurately model the radiometry of image formation, the following code will need to know the area of this bounding box, so it is returned via `sampleBoundsArea`.

```
(RealisticCamera Method Definitions) +≡
Point3f RealisticCamera::SampleExitPupil(const Point2f &pFilm,
                                         const Point2f &lensSample, Float *sampleBoundsArea) const {
    (Find exit pupil bound for sample distance from film center 393)
    (Generate sample point inside exit pupil bound 393)
    (Return sample point rotated by angle of pFilm with +x axis 394)
}
```

```
(Find exit pupil bound for sample distance from film center) ≡ 393
Float rFilm = std::sqrt(pFilm.x * pFilm.x + pFilm.y * pFilm.y);
int rIndex = rFilm / (film->diagonal / 2) * exitPupilBounds.size();
rIndex = std::min((int)exitPupilBounds.size() - 1, rIndex);
Bounds2f pupilBounds = exitPupilBounds[rIndex];
if (sampleBoundsArea) *sampleBoundsArea = pupilBounds.Area();
```

Given the pupil's bounding box, a point inside it is sampled via linear interpolation with the provided `lensSample` value, which is in  $[0, 1]^2$ .

```
(Generate sample point inside exit pupil bound) ≡ 393
Point2f pLens = pupilBounds.Lerp(lensSample);
```

Because the exit pupil bound was computed from a point on the film along the  $+x$  axis but the point `pFilm` is an arbitrary point on the film, the sample point in the exit pupil bound must be rotated by the same angle as `pFilm` makes with the  $+x$  axis.

*(Return sample point rotated by angle of pFilm with +x axis) ≡ 393*

```
Float sinTheta = (rFilm != 0) ? pFilm.y / rFilm : 0;
Float cosTheta = (rFilm != 0) ? pFilm.x / rFilm : 1;
return Point3f(cosTheta * pLens.x - sinTheta * pLens.y,
               sinTheta * pLens.x + cosTheta * pLens.y,
               LensRearZ());
```

## 6.4.6 GENERATING RAYS

Now that we have the machinery to trace rays through lens systems and to sample points in the exit pupil bound from points on the film plane, transforming a CameraSample into a ray leaving the camera is fairly straightforward: we need to compute the sample's position on the film plane and generate a ray from this point to the rear lens element, which is then traced through the lens system.

*(RealisticCamera Method Definitions) +≡*

```
Float RealisticCamera::GenerateRay(const CameraSample &sample,
                                    Ray *ray) const {
    (Find point on film, pFilm, corresponding to sample.pFilm 394)
    (Trace ray from pFilm through lens system 394)
    (Finish initialization of RealisticCamera ray 395)
    (Return weighting for RealisticCamera ray 783)
}
```

The CameraSample::pFilm value is with respect to the overall resolution of the image in pixels. Here, we're operating with a physical model of a sensor, so we start by converting back to a sample in  $[0, 1]^2$ . Next, the corresponding point on the film is found by linearly interpolating with this sample value over its area.

*(Find point on film, pFilm, corresponding to sample.pFilm) ≡ 394*

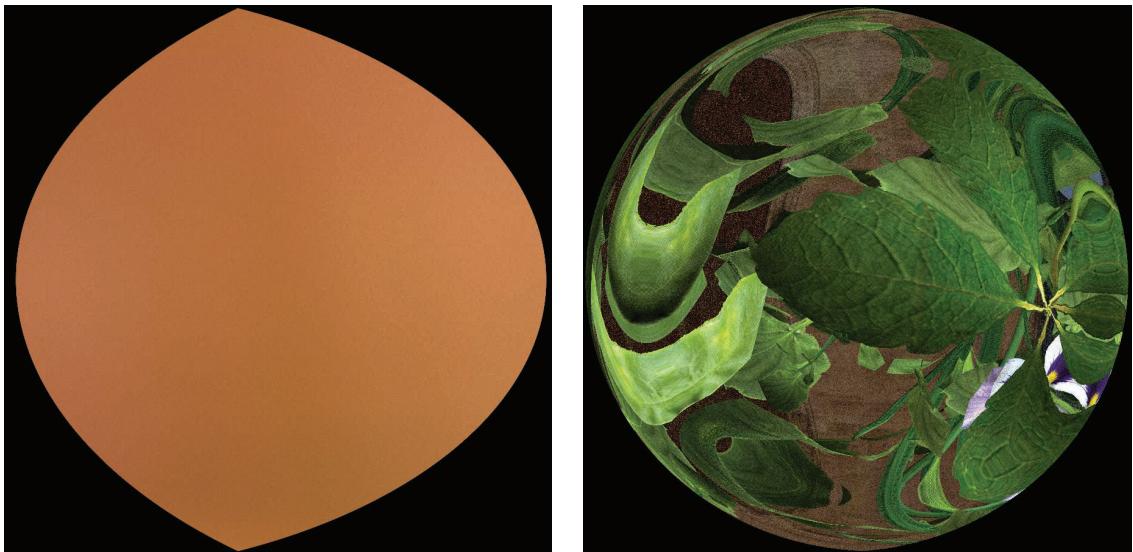
```
Point2f s(sample.pFilm.x / film->fullResolution.x,
           sample.pFilm.y / film->fullResolution.y);
Point2f pFilm2 = film->GetPhysicalExtent().Lerp(s);
Point3f pFilm(-pFilm2.x, pFilm2.y, 0);
```

SampleExitPupil() then gives us a point on the plane tangent to the rear lens element, which in turn lets us determine the ray's direction. In turn, we can trace this ray through the lens system. If the ray is blocked by the aperture stop or otherwise doesn't make it through the lens system, GenerateRay() returns a 0 weight. (Callers should be sure to check for this case.)

*(Trace ray from pFilm through lens system) ≡ 394*

```
Float exitPupilBoundsArea;
Point3f pRear = SampleExitPupil(Point2f(pFilm.x, pFilm.y), sample.pLens,
                                 &exitPupilBoundsArea);
Ray rFilm(pFilm, pRear - pFilm, Infinity,
          Lerp(sample.time, shutterOpen, shutterClose));
if (!TraceLensesFromFilm(rFilm, ray))
    return 0;
```

```
Bounds2::Lerp() 80
Camera::film 356
Camera::shutterClose 356
Camera::shutterOpen 356
CameraSample 357
CameraSample::pFilm 357
CameraSample::time 357
Film::GetPhysicalExtent()
    488
Float 1062
Infinity 210
Lerp() 1079
Point2f 68
Point3f 68
Ray 73
RealisticCamera::LensRearZ()
    381
RealisticCamera::
    SampleExitPupil()
    393
RealisticCamera::
    TraceLensesFromFilm()
    382
```



**Figure 6.25: The Exit Pupil, as Seen from Two Points on the Film Plane in Figure 6.17.** (a) The exit pupil as seen from a point where the scene is in sharp focus; the incident radiance is effectively constant over its area. (b) As seen from a pixel in an out-of-focus area, the exit pupil is a small image of part of the scene, with potentially rapidly varying radiance.

If the ray does successfully exit the lens system, then the usual details have to be handled to finish its initialization.

```
<Finish initialization of RealisticCamera ray> ≡
    *ray = CameraToWorld(*ray);
    ray->d = Normalize(ray->d);
    ray->medium = medium;
```

394

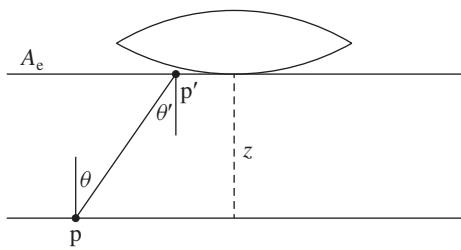
The fragment *<Return weighting for RealisticCamera ray>* will be defined later, in Section 13.6.6, after some necessary background from Monte Carlo integration has been introduced.

#### 6.4.7 THE CAMERA MEASUREMENT EQUATION

Given this more accurate simulation of the process of real image formation, it's also worthwhile to more carefully define the radiometry of the measurement made by a film or a camera sensor. Rays from the exit pupil to the film carry radiance from the scene; as considered from a point on the film plane, there is thus a set of directions from which radiance is incident. The distribution of radiance leaving the exit pupil is affected by the amount of defocus blur seen by the point on the film—Figure 6.25 shows two renderings of the exit pupil's radiance as seen from two points on the film.

Camera::CameraToWorld 356  
 Camera::medium 356  
 Ray::d 73  
 Ray::medium 74  
 Vector3::Normalize() 66

Given the incident radiance function, we can define the irradiance at a point on the film plane. If we start with the definition of irradiance in terms of radiance, Equation (5.4), we can then convert from an integral over solid angle to an integral over area (in this case, an area of the plane tangent to the rear lens element that bounds the exit pupil,  $A_e$ ) using



**Figure 6.26:** Geometric setting for the irradiance measurement equation, (6.5). Radiance can be measured as it passes through points  $p'$  on the plane tangent to the rear lens element to a point on the film plane  $p$ .  $z$  is the axial distance from the film plane to the rear element tangent plane, `RealisticCamera::LensRearZ()`, and  $\theta$  is the angle between the vector from  $p'$  to  $p$  and the optical axis.

Equation (5.6). This gives us the irradiance for a point  $p$  on the film plane:

$$E(p) = \int_{A_e} L_i(p, p') \frac{|\cos \theta \cos \theta'|}{||p' - p||^2} dA_e.$$

Figure 6.26 shows the geometry of the situation. Because the film plane is perpendicular to the exit pupil plane,  $\theta = \theta'$ . We can further take advantage of the fact that the distance between  $p$  and  $p'$  is equal to the axial distance from the film plane to the exit pupil (which we'll denote here by  $z$ ) times  $\cos \theta$ . Putting this all together, we have

$$E(p) = \frac{1}{z^2} \int_{A_e} L_i(p, p') |\cos^4 \theta| dA_e. \quad (6.5)$$

For cameras where the extent of the film is relatively large with respect to the distance  $z$ , the  $\cos^4 \theta$  term can meaningfully reduce the incident irradiance—this factor also contributes to vignetting. Most modern digital cameras correct for this effect with preset correction factors that increase pixel values toward the edges of the sensor.

Integrating irradiance at a point on the film over the time that the shutter is open gives *fluence*, which is the radiometric unit for energy per area,  $\text{J/m}^2$ .

$$H(p) = \frac{1}{z^2} \int_{t_0}^{t_1} \int_{A_e} L_i(p, p', t') |\cos^4 \theta| dA_e dt'. \quad (6.6)$$

Measuring fluence at a point captures the effect that the amount of energy received on the film plane is partially related to the length of time the camera shutter is open.

Photographic film (or CCD or CMOS sensors in digital cameras) actually measure radiant energy over a small area.<sup>4</sup> Taking Equation (6.6) and also integrating over sensor pixel area,  $A_p$ , we have

<sup>4</sup> A typical size for pixels in digital cameras in 2015-era mobile phones is 1.5 microns per side.

$$J = \frac{1}{z^2} \int_{A_p} \int_{t_0}^{t_1} \int_{A_e} L_i(p, p', t') |\cos^4 \theta| dA_e dt' dA_p, \quad [6.7]$$

the Joules arriving at a pixel.

In Section 13.2, we'll see how Monte Carlo can be applied to estimate the values of these various integrals. Then in Section 13.6.6 we will define the fragment `<Return weighting for RealisticCamera ray>` in `RealisticCamera::GenerateRay()`; various approaches to computing the weight allow us to compute each of these quantities. Section 16.1.1 defines the *importance function* of a camera model, which characterizes its sensitivity to incident illumination arriving along different rays.

## FURTHER READING

In his seminal Sketchpad system, Sutherland (1963) was the first to use projection matrices for computer graphics. Akenine-Möller, Haines, and Hoffman (2008) have provided a particularly well-written derivation of the orthographic and perspective projection matrices. Other good references for projections are Rogers and Adams's *Mathematical Elements for Computer Graphics* (1990), and Eberly's book (2001) on game engine design.

An unusual projection method was used by Greene and Heckbert (1986) for generating images for Omnimax® theaters. The `EnvironmentCamera` in this chapter is similar to the camera model described by Musgrave (1992).

Potmesil and Chakravarty (1981, 1982, 1983) did early work on depth of field and motion blur in computer graphics. Cook and collaborators developed a more accurate model for these effects based on the thin lens model; this is the approach used for the depth of field calculations in Section 6.2.3 (Cook, Porter, and Carpenter 1984; Cook 1986). See Adams and Levoy (2007) for a broad analysis of the types of radiance measurements that can be taken with cameras that have non-pinhole apertures.

Kolb, Mitchell, and Hanrahan (1995) showed how to simulate complex camera lens systems with ray tracing in order to model the imaging effects of real cameras; the `RealisticCamera` in Section 6.4 is based on their approach. Steinert et al. (2011) improve a number of details of this simulation, incorporating wavelength-dependent effects and accounting for both diffraction and glare. Our approach for approximating the exit pupil in Section 6.4.5 is similar to theirs. See the books by Hecht (2002) and Smith (2007) for excellent introductions to optics and lens systems.

Hullin et al. (2012) use polynomials to model the effect of lenses on rays passing through them; they are able to construct polynomials that approximate entire lens systems from polynomial approximations of individual lenses. This approach saves the computational expense of tracing rays through lenses, though for complex scenes, this cost is generally negligible in relation to the rest of rendering computations. Hanika and Dachsbacher (2014) improved the accuracy of this approach and showed how to combine the method with bidirectional path tracing.

Chen et al. (2009) describe the implementation a fairly complete simulation of a digital camera, including the analog-to-digital conversion and noise in the measured pixel values inherent in this process.

## EXERCISES

- ② 6.1 Some types of cameras expose the film by sliding a rectangular slit across the film. This leads to interesting effects when objects are moving in a different direction from the exposure slit (Glassner 1999; Stephenson 2006). Furthermore, most digital cameras read out pixel values from scanlines in succession over a period of a few milliseconds; this leads to *rolling shutter* artifacts, which have similar visual characteristics. Modify the way that time samples are generated in one or more of the camera implementations in this chapter to model such effects. Render images with moving objects that clearly show the effect of accounting for this issue.
- ② 6.2 Write an application that loads images rendered by the `EnvironmentCamera`, and uses texture mapping to apply them to a sphere centered at the eyepoint such that they can be viewed interactively. The user should be able to freely change the viewing direction. If the correct texture-mapping function is used for generating texture coordinates on the sphere, the image generated by the application will appear as if the viewer was at the camera's location in the scene when it was rendered, thus giving the user the ability to interactively look around the scene.
- ② 6.3 The aperture stop in the `RealisticCamera` is modeled as a perfect circle; for cameras with adjustable apertures, the aperture is generally formed by movable blades with straight edges and is thus an  $n$ -gon. Modify the `RealisticCamera` to model a more realistic aperture shape and render images showing the differences from your model. You may find it useful to render a scene with small, bright, out-of-focus objects (e.g., specular highlights), to show off the differences.
- ② 6.4 The standard model for depth of field in computer graphics models the circle of confusion as imaging a point in the scene to a disk with uniform intensity, although many real lenses produce circles of confusion with nonlinear variation such as a Gaussian distribution. This effect is known as "Bokeh" (Buhler and Wexler 2002). For example, catadioptric (mirror) lenses produce doughnut-shaped highlights when small points of light are viewed out of focus. Modify the implementation of depth of field in the `RealisticCamera` to produce images with this effect (e.g., by biasing the distribution of lens sample positions). Render images showing the difference between this and the standard model.
- ② 6.5 *Focal stack rendering:* a focal stack is a series of images of a fixed scene where the camera is focused at a different distance for each image. Hasinoff and Kutulakos (2011) and Jacobs et al. (2012) introduce a number of applications of focal stacks, including freeform depth of field, where the user can specify arbitrary depths that are in focus, achieving effects not possible with traditional optics. Render focal stacks with `pbrt` and write an interactive tool to control focus effects with them.
- ③ 6.6 *Light field camera:* Ng et al. (2005) discuss the physical design and applications of a camera that captures small images of the exit pupil across the film,

`EnvironmentCamera` 376

`RealisticCamera` 378

rather than averaging the radiance over the entire exit pupil at each pixel, as conventional cameras do. Such a camera captures a representation of the light field—the spatially and directionally varying distribution of radiance arriving at the camera sensor. By capturing the light field, a number of interesting operations are possible, including refocusing photographs after they have been taken. Read Ng et al.’s paper and implement a Camera in pbrt that captures the light field of a scene. Write a tool to allow users to interactively refocus these light fields.

- ③ 6.7 The RealisticCamera implementation places the film at the center of and perpendicular to the optical axis. While this is the most common configuration of actual cameras, interesting effects can be achieved by adjusting the film’s placement with respect to the lens system.

For example, the plane of focus in the current implementation is always perpendicular to the optical axis; if the film plane (or the lens system) is tilted so that the film isn’t perpendicular to the optical axis, then the plane of focus is no longer perpendicular to the optical axis. (This can be useful for landscape photography, for example, where aligning the plane of focus with the ground plane allows greater depth of field even with larger apertures.) Alternatively, the film plane can be shifted so that it’s not centered on the optical axis; this shift can be used to keep the plane of focus aligned with a very tall object, for example.

Modify RealisticCamera to allow one or both of these adjustments and render images showing the result. Note that a number of places in the current implementation (e.g., the exit pupil computation) have assumptions that will be violated by these changes that you will need to address.

# CHAPTER SEVEN

## 07 SAMPLING AND RECONSTRUCTION

Although the final output of a renderer like `pbrt` is a 2D grid of colored pixels, incident radiance is actually a continuous function defined over the film plane. The manner in which the discrete pixel values are computed from this continuous function can noticeably affect the quality of the final image generated by the renderer; if this process is not performed carefully, artifacts will be present. Conversely, if it is performed well, a relatively small amount of additional computation to this end can substantially improve the quality of the rendered images.

This chapter starts by introducing *sampling theory*—the theory of taking discrete sample values from functions defined over continuous domains and then using those samples to reconstruct new functions that are similar to the original. Building on principles of sampling theory as well as ideas from low-discrepancy point sets, which are a particular type of well-distributed sample points, the `Samplers` defined in this chapter generate  $n$ -dimensional sample vectors in various ways.<sup>1</sup> Five `Sampler` implementations are described in this chapter, spanning a variety of approaches to the sampling problem.

This chapter concludes with the `Filter` class and the `Film` class. The `Filter` is used to determine how multiple samples near each pixel are blended together to compute the final pixel value, and the `Film` class accumulates image sample contributions into pixels of images.

---

<sup>1</sup> Recall that in the previous chapter `Cameras` used `CameraSamples` to place points on the film plane, on the lens, and in time—the `CameraSample` values are set by taking the first few dimensions of these sample vectors.

## 7.1 SAMPLING THEORY

A digital image is represented as a set of pixel values, typically aligned on a rectangular grid. When a digital image is displayed on a physical device, these values are used to determine the spectral power emitted by pixels on the display. When thinking about digital images, it is important to differentiate between image pixels, which represent the value of a function at a particular sample location, and display pixels, which are physical objects that emit light with some distribution. (For example, in an LCD display, the color and brightness may change substantially when the display is viewed at oblique angles.) Displays use the image pixel values to construct a new image function over the display surface. This function is defined at all points on the display, not just the infinitesimal points of the digital image’s pixels. This process of taking a collection of sample values and converting them back to a continuous function is called *reconstruction*.

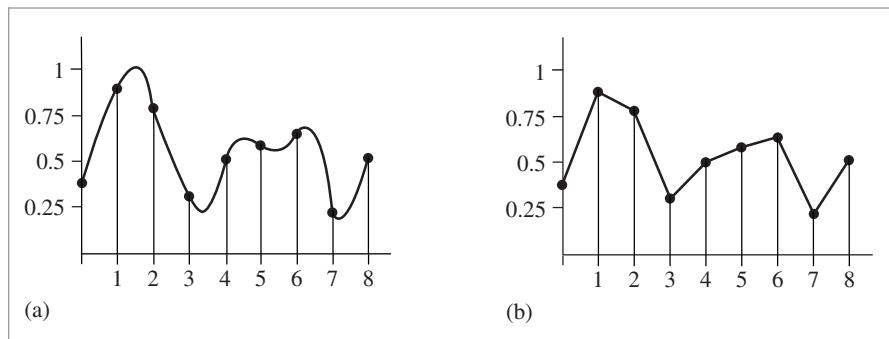
In order to compute the discrete pixel values in the digital image, it is necessary to sample the original continuously defined image function. In pbrt, like most other ray-tracing renderers, the only way to get information about the image function is to sample it by tracing rays. For example, there is no general method that can compute bounds on the variation of the image function between two points on the film plane. While an image could be generated by just sampling the function precisely at the pixel positions, a better result can be obtained by taking more samples at different positions and incorporating this additional information about the image function into the final pixel values. Indeed, for the best quality result, the pixel values should be computed such that the reconstructed image on the display device is as close as possible to the original image of the scene on the virtual camera’s film plane. Note that this is a subtly different goal from expecting the display’s pixels to take on the image function’s actual value at their positions. Handling this difference is the main goal of the algorithms implemented in this chapter.<sup>2</sup>

Because the sampling and reconstruction process involves approximation, it introduces error known as *aliasing*, which can manifest itself in many ways, including jagged edges or flickering in animations. These errors occur because the sampling process is not able to capture all of the information from the continuously defined image function.

As an example of these ideas, consider a 1D function (which we will interchangeably refer to as a signal), given by  $f(x)$ , where we can evaluate  $f(x')$  at any desired location  $x'$  in the function’s domain. Each such  $x'$  is called a *sample position*, and the value of  $f(x')$  is the *sample value*. Figure 7.1 shows a set of samples of a smooth 1D function, along with a reconstructed signal  $\tilde{f}$  that approximates the original function  $f$ . In this example,  $\tilde{f}$  is a piecewise linear function that approximates  $f$  by linearly interpolating neighboring sample values (readers already familiar with sampling theory will recognize this as reconstruction with a hat function). Because the only information available about

---

<sup>2</sup> In this book, we will ignore issues related to the characteristics of physical display pixels and will work under the assumption that the display performs the ideal reconstruction process described later in this section. This assumption is patently at odds with how actual displays work, but it avoids unnecessary complication of the analysis here. Chapter 3 of Glassner (1995) has a good treatment of nonidealized display devices and their impact on the image sampling and reconstruction process.



**Figure 7.1:** (a) By taking a set of *point samples* of  $f(x)$  (indicated by black dots), we determine the value of the function at those positions. (b) The sample values can be used to *reconstruct* a function  $\tilde{f}(x)$  that is an approximation to  $f(x)$ . The sampling theorem, introduced in Section 7.1.3, makes a precise statement about the conditions on  $f(x)$ , the number of samples taken, and the reconstruction technique used under which  $\tilde{f}(x)$  is exactly the same as  $f(x)$ . The fact that the original function can sometimes be reconstructed exactly from point samples alone is remarkable.

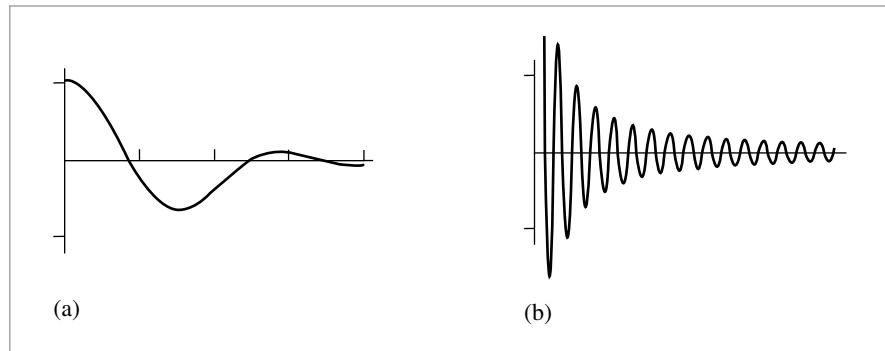
$f$  comes from the sample values at the positions  $x'$ ,  $\tilde{f}$  is unlikely to match  $f$  perfectly since there is no information about  $f$ 's behavior between the samples.

*Fourier analysis* can be used to evaluate the quality of the match between the reconstructed function and the original. This section will introduce the main ideas of Fourier analysis with enough detail to work through some parts of the sampling and reconstruction processes but will omit proofs of many properties and skip details that aren't directly relevant to the sampling algorithms used in pbrt. The “Further Reading” section of this chapter has pointers to more detailed information about these topics.

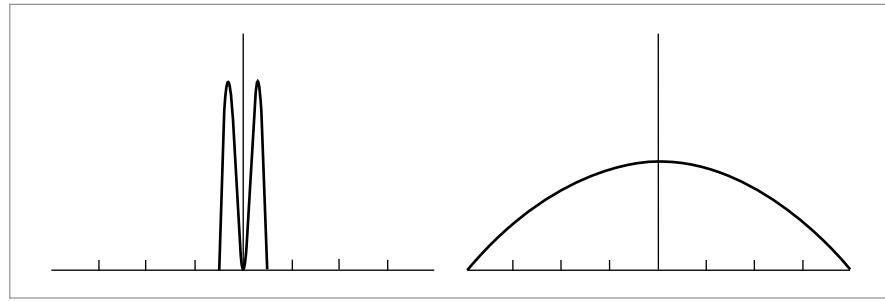
### 7.1.1 THE FREQUENCY DOMAIN AND THE FOURIER TRANSFORM

One of the foundations of Fourier analysis is the Fourier transform, which represents a function in the *frequency domain*. (We will say that functions are normally expressed in the *spatial domain*.) Consider the two functions graphed in Figure 7.2. The function in Figure 7.2(a) varies relatively slowly as a function of  $x$ , while the function in Figure 7.2(b) varies much more rapidly. The more slowly varying function is said to have lower frequency content. Figure 7.3 shows the frequency space representations of these two functions; the lower frequency function's representation goes to 0 more quickly than does the higher frequency function.

Most functions can be decomposed into a weighted sum of shifted sinusoids. This remarkable fact was first described by Joseph Fourier, and the Fourier transform converts a function into this representation. This frequency space representation of a function gives insight into some of its characteristics—the distribution of frequencies in the sine functions corresponds to the distribution of frequencies in the original function. Using this form, it is possible to use Fourier analysis to gain insight into the error that is introduced by the sampling and reconstruction process and how to reduce the perceptual impact of this error.



**Figure 7.2:** (a) Low-frequency function and (b) high-frequency function. Roughly speaking, the higher frequency a function is, the more quickly it varies over a given region.



**Figure 7.3: Frequency Space Representations of the Functions in Figure 7.2.** The graphs show the contribution of each frequency  $\omega$  to each of the functions in the spatial domain.

The Fourier transform of a 1D function  $f(x)$  is<sup>3</sup>

$$F(\omega) = \int_{-\infty}^{\infty} f(x) e^{-i2\pi\omega x} dx. \quad (7.1)$$

(Recall that  $e^{ix} = \cos x + i \sin x$ , where  $i = \sqrt{-1}$ .) For simplicity, here we will consider only *even* functions where  $f(-x) = f(x)$ , in which case the Fourier transform of  $f$  has no imaginary terms. The new function  $F$  is a function of *frequency*,  $\omega$ .<sup>4</sup> We will denote the Fourier transform operator by  $\mathcal{F}$ , such that  $\mathcal{F}\{f(x)\} = F(\omega)$ .  $\mathcal{F}$  is clearly a linear operator—that is,  $\mathcal{F}\{af(x)\} = a\mathcal{F}\{f(x)\}$  for any scalar  $a$ , and  $\mathcal{F}\{f(x) + g(x)\} = \mathcal{F}\{f(x)\} + \mathcal{F}\{g(x)\}$ .

<sup>3</sup> The reader should be warned that the constants in front of these integrals are not always the same in different fields. For example, some authors (including many in the physics community) prefer to multiply both integrals by  $1/\sqrt{2\pi}$ .

<sup>4</sup> In this chapter, we will use the  $\omega$  symbol to denote frequency. Throughout the rest of the book,  $\omega$  denotes normalized direction vectors. This overloading of notation should never be confusing, given the contexts where these symbols are used. Similarly, when we refer to a function's "spectrum," we are referring to its distribution of frequencies in its frequency space representation, rather than anything related to color.

**Table 7.1: Fourier Pairs.** Functions in the spatial domain and their frequency space representations. Because of the symmetry properties of the Fourier transform, if the left column is instead considered to be frequency space, then the right column is the spatial equivalent of those functions as well.

Spatial Domain	Frequency Space Representation
Box: $f(x) = 1$ if $ x  < 1/2$ , 0 otherwise	Sinc: $f(\omega) = \text{sinc}(\omega) = \sin(\pi\omega)/(\pi\omega)$
Gaussian: $f(x) = e^{-\pi x^2}$	Gaussian: $f(\omega) = e^{-\pi\omega^2}$
Constant: $f(x) = 1$	Delta: $f(\omega) = \delta(\omega)$
Sinusoid: $f(x) = \cos x$	Translated delta: $f(\omega) = \pi(\delta(1 - 2\pi\omega) + \delta(1 + 2\pi\omega))$
Shah: $f(x) = \mathbb{U}_T(x) = T \sum_i \delta(x - iT)$	Shah: $f(\omega) = \mathbb{U}_{1/T}(\omega) = (1/T) \sum_i \delta(\omega - i/T)$

Equation (7.1) is called the *Fourier analysis* equation, or sometimes just the *Fourier transform*. We can also transform from the frequency domain back to the spatial domain using the *Fourier synthesis* equation, or the *inverse Fourier transform*:

$$f(x) = \int_{-\infty}^{\infty} F(\omega) e^{i2\pi\omega x} d\omega. \quad [7.2]$$

Table 7.1 shows a number of important functions and their frequency space representations. A number of these functions are based on the Dirac delta distribution, a special function that is defined such that  $\int \delta(x) dx = 1$ , and for all  $x \neq 0$ ,  $\delta(x) = 0$ . An important consequence of these properties is that

$$\int f(x) \delta(x) dx = f(0).$$

The delta distribution cannot be expressed as a standard mathematical function, but instead is generally thought of as the limit of a unit area box function centered at the origin with width approaching 0.

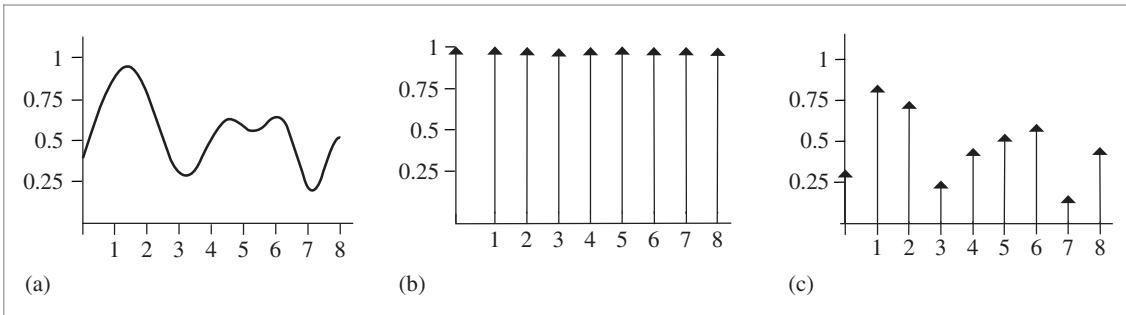
### 7.1.2 IDEAL SAMPLING AND RECONSTRUCTION

Using frequency space analysis, we can now formally investigate the properties of sampling. Recall that the sampling process requires us to choose a set of equally spaced sample positions and compute the function's value at those positions. Formally, this corresponds to multiplying the function by a “shah,” or “impulse train,” function, an infinite sum of equally spaced delta functions. The shah  $\mathbb{U}_T(x)$  is defined as

$$\mathbb{U}_T(x) = T \sum_{i=-\infty}^{\infty} \delta(x - iT),$$

where  $T$  defines the period, or *sampling rate*. This formal definition of sampling is illustrated in Figure 7.4. The multiplication yields an infinite sequence of values of the function at equally spaced points:

$$\mathbb{U}_T(x)f(x) = T \sum_i \delta(x - iT)f(iT).$$



**Figure 7.4: Formalizing the Sampling Process.** (a) The function  $f(x)$  is multiplied by (b) the shah function  $\Pi_T(x)$ , giving (c) an infinite sequence of scaled delta functions that represent its value at each sample point.

These sample values can be used to define a reconstructed function  $\tilde{f}$  by choosing a reconstruction filter function  $r(x)$  and computing the *convolution*

$$(\Pi_T(x)f(x)) \otimes r(x),$$

where the convolution operation  $\otimes$  is defined as

$$f(x) \otimes g(x) = \int_{-\infty}^{\infty} f(x') g(x - x') dx'.$$

For reconstruction, convolution gives a weighted sum of scaled instances of the reconstruction filter centered at the sample points:

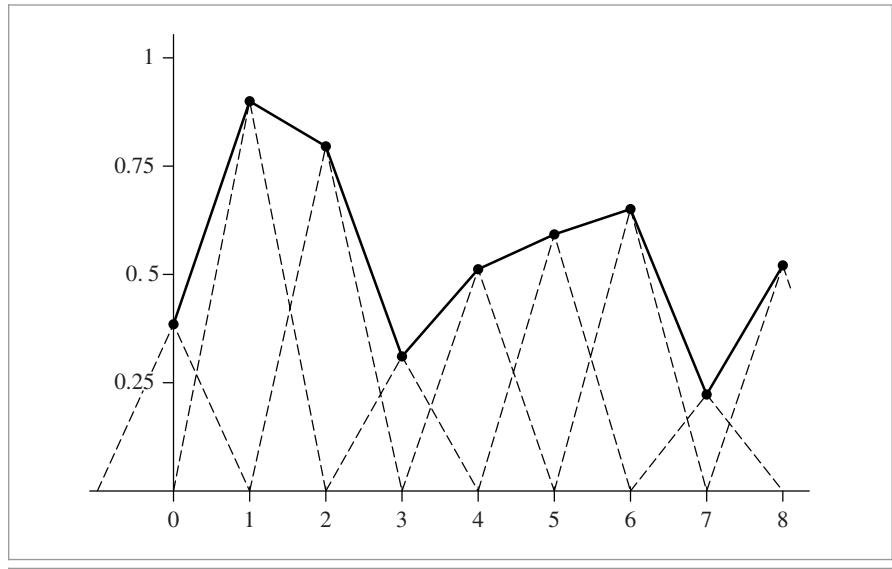
$$\tilde{f}(x) = T \sum_{i=-\infty}^{\infty} f(iT) r(x - iT).$$

For example, in Figure 7.1, the triangle reconstruction filter,  $r(x) = \max(0, 1 - |x|)$ , was used. Figure 7.5 shows the scaled triangle functions used for that example.

We have gone through a process that may seem gratuitously complex in order to end up at an intuitive result: the reconstructed function  $\tilde{f}(x)$  can be obtained by interpolating among the samples in some manner. By setting up this background carefully, however, Fourier analysis can now be applied to the process more easily.

We can gain a deeper understanding of the sampling process by analyzing the sampled function in the frequency domain. In particular, we will be able to determine the conditions under which the original function can be exactly recovered from its values at the sample locations—a very powerful result. For the discussion here, we will assume for now that the function  $f(x)$  is *band limited*—there exists some frequency  $\omega_0$  such that  $f(x)$  contains no frequencies greater than  $\omega_0$ . By definition, band-limited functions have frequency space representations with compact support, such that  $F(\omega) = 0$  for all  $|\omega| > \omega_0$ . Both of the spectra in Figure 7.3 are band limited.

An important idea used in Fourier analysis is the fact that the Fourier transform of the product of two functions  $\mathcal{F}\{f(x)g(x)\}$  can be shown to be the convolution of their individual Fourier transforms  $F(\omega)$  and  $G(\omega)$ :



**Figure 7.5:** The sum of instances of the triangle reconstruction filter, shown with dashed lines, gives the reconstructed approximation to the original function, shown with a solid line.

$$\mathcal{F}\{f(x)g(x)\} = F(\omega) \otimes G(\omega).$$

It is similarly the case that convolution in the spatial domain is equivalent to multiplication in the frequency domain:

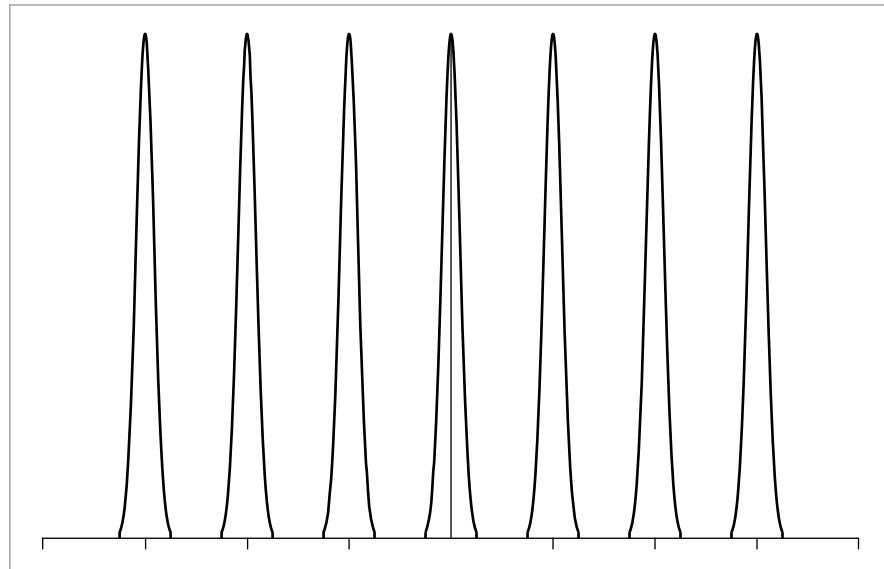
$$\mathcal{F}\{f(x) \otimes g(x)\} = F(\omega)G(\omega). \quad [7.3]$$

These properties are derived in the standard references on Fourier analysis. Using these ideas, the original sampling step in the spatial domain, where the product of the shah function and the original function  $f(x)$  is found, can be equivalently described by the convolution of  $F(\omega)$  with another shah function in frequency space.

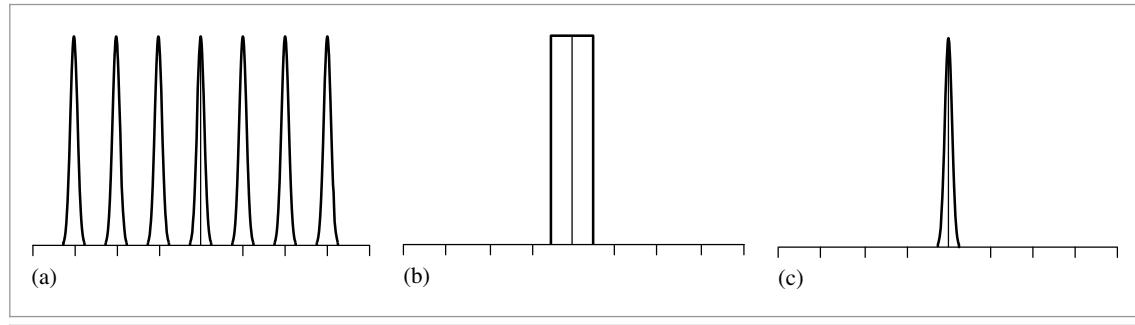
We also know the spectrum of the shah function  $\mathbb{W}_T(x)$  from Table 7.1; the Fourier transform of a shah function with period  $T$  is another shah function with period  $1/T$ . This reciprocal relationship between periods is important to keep in mind: it means that if the samples are farther apart in the spatial domain, they are closer together in the frequency domain.

Thus, the frequency domain representation of the sampled signal is given by the convolution of  $F(\omega)$  and this new shah function. Convolving a function with a delta function just yields a copy of the function, so convolving with a shah function yields an infinite sequence of copies of the original function, with spacing equal to the period of the shah (Figure 7.6). This is the frequency space representation of the series of samples.

Now that we have this infinite set of copies of the function's spectrum, how do we reconstruct the original function? Looking at Figure 7.6, the answer is obvious: just discard all of the spectrum copies except the one centered at the origin, giving the original



**Figure 7.6: The Convolution of  $F(\omega)$  and the Shah Function.** The result is infinitely many copies of  $F$ .



**Figure 7.7:** Multiplying (a) a series of copies of  $F(\omega)$  by (b) the appropriate box function yields (c) the original spectrum.

$F(\omega)$ . In order to throw away all but the center copy of the spectrum, we multiply by a box function of the appropriate width (Figure 7.7). The box function  $\Pi_T(x)$  of width  $T$  is defined as

$$\Pi_T(x) = \begin{cases} 1/(2T) & |x| < T \\ 0 & \text{otherwise.} \end{cases}$$

This multiplication step corresponds to convolution with the reconstruction filter in the spatial domain. This is the ideal sampling and reconstruction process. To summarize:

$$\tilde{F} = (F(\omega) \otimes \mathbb{U}_{1/T}(\omega)) \Pi_T(\omega).$$

This is a remarkable result: we have been able to determine the exact frequency space representation of  $f(x)$ , purely by sampling it at a set of regularly spaced points. Other than knowing that the function was band limited, no additional information about the composition of the function was used.

Applying the equivalent process in the spatial domain will likewise recover  $f(x)$  exactly. Because the inverse Fourier transform of the box function is the sinc function, ideal reconstruction in the spatial domain is

$$\tilde{f} = (f(x) \mathbb{U}_T(x)) \otimes \text{sinc}(x),$$

or

$$\tilde{f}(x) = \sum_{i=-\infty}^{\infty} \text{sinc}(x - i) f(i).$$

Unfortunately, because the sinc function has infinite extent, it is necessary to use all of the sample values  $f(i)$  to compute any particular value of  $\tilde{f}(x)$  in the spatial domain. Filters with finite spatial extent are preferable for practical implementations even though they don't reconstruct the original function perfectly.

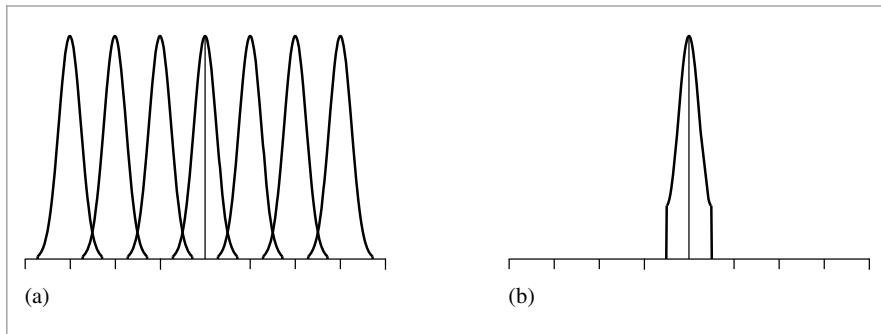
A commonly used alternative in graphics is to use the box function for reconstruction, effectively averaging all of the sample values within some region around  $x$ . This is a very poor choice, as can be seen by considering the box filter's behavior in the frequency domain: This technique attempts to isolate the central copy of the function's spectrum by multiplying by a sinc, which not only does a bad job of selecting the central copy of the function's spectrum but includes high-frequency contributions from the infinite series of other copies of it as well.

### 7.1.3 ALIASING

Beyond the issue of the sinc function's infinite extent, one of the most serious practical problems with the ideal sampling and reconstruction approach is the assumption that the signal is band limited. For signals that are not band limited, or signals that aren't sampled at a sufficiently high sampling rate for their frequency content, the process described earlier will reconstruct a function that is different from the original signal.

The key to successful reconstruction is the ability to exactly recover the original spectrum  $F(\omega)$  by multiplying the sampled spectrum with a box function of the appropriate width. Notice that in Figure 7.6, the copies of the signal's spectrum are separated by empty space, so perfect reconstruction is possible. Consider what happens, however, if the original function was sampled with a lower sampling rate. Recall that the Fourier transform of a shah function  $\mathbb{U}_T$  with period  $T$  is a new shah function with period  $1/T$ . This means that if the spacing between samples increases in the spatial domain, the sample spacing decreases in the frequency domain, pushing the copies of the spectrum  $F(\omega)$  closer together. If the copies get too close together, they start to overlap.

Because the copies are added together, the resulting spectrum no longer looks like many copies of the original (Figure 7.8). When this new spectrum is multiplied by a box function, the result is a spectrum that is similar but not equal to the original  $F(\omega)$ : high-frequency details in the original signal leak into lower frequency regions of the spectrum



**Figure 7.8:** (a) When the sampling rate is too low, the copies of the function’s spectrum overlap, resulting in (b) aliasing when reconstruction is performed.

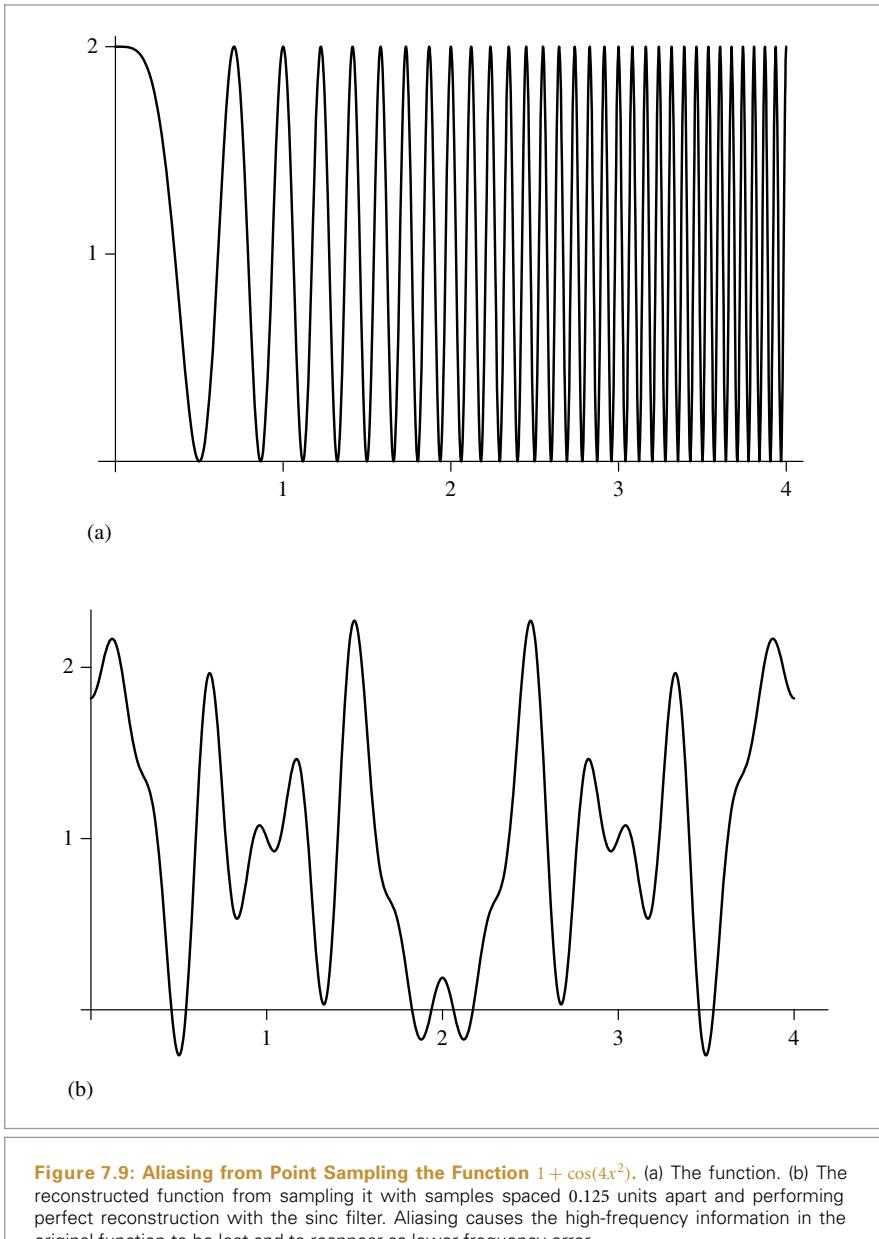
of the reconstructed signal. These new low-frequency artifacts are called *aliases* (because high frequencies are “masquerading” as low frequencies), and the resulting signal is said to be *aliased*. Figure 7.9 shows the effects of aliasing from undersampling and then reconstructing the 1D function  $f(x) = 1 + \cos(4x^2)$ .

A possible solution to the problem of overlapping spectra is to simply increase the sampling rate until the copies of the spectrum are sufficiently far apart to not overlap, thereby eliminating aliasing completely. In fact, the *sampling theorem* tells us exactly what rate is required. This theorem says that as long as the frequency of uniform sample points  $\omega_s$  is greater than twice the maximum frequency present in the signal  $\omega_0$ , it is possible to reconstruct the original signal perfectly from the samples. This minimum sampling frequency is called the *Nyquist frequency*.

For signals that are not band limited ( $\omega_0 = \infty$ ), it is impossible to sample at a high enough rate to perform perfect reconstruction. Non-band-limited signals have spectra with infinite support, so no matter how far apart the copies of their spectra are (i.e., how high a sampling rate we use), there will always be overlap. Unfortunately, few of the interesting functions in computer graphics are band limited. In particular, any function containing a discontinuity cannot be band limited, and therefore we cannot perfectly sample and reconstruct it. This makes sense because the function’s discontinuity will always fall between two samples and the samples provide no information about the location of the discontinuity. Thus, it is necessary to apply different methods besides just increasing the sampling rate in order to counteract the error that aliasing can introduce to the renderer’s results.

#### 7.1.4 ANTIALIASING TECHNIQUES

If one is not careful about sampling and reconstruction, myriad artifacts can appear in the final image. It is sometimes useful to distinguish between artifacts due to sampling and those due to reconstruction; when we wish to be precise we will call sampling artifacts *prealiasing* and reconstruction artifacts *postaliasing*. Any attempt to fix these errors is broadly classified as *antialiasing*. This section reviews a number of antialiasing techniques beyond just increasing the sampling rate everywhere.



**Figure 7.9: Aliasing from Point Sampling the Function  $1 + \cos(4x^2)$ .** (a) The function. (b) The reconstructed function from sampling it with samples spaced 0.125 units apart and performing perfect reconstruction with the sinc filter. Aliasing causes the high-frequency information in the original function to be lost and to reappear as lower frequency error.

### Nonuniform Sampling

Although the image functions that we will be sampling are known to have infinite-frequency components and thus can't be perfectly reconstructed from point samples, it is possible to reduce the visual impact of aliasing by varying the spacing between samples in a nonuniform way. If  $\xi$  denotes a random number between 0 and 1, a nonuniform set

of samples based on the impulse train is

$$\sum_{i=-\infty}^{\infty} \delta \left( x - \left( i + \frac{1}{2} - \xi \right) T \right).$$

For a fixed sampling rate that isn't sufficient to capture the function, both uniform and nonuniform sampling produce incorrect reconstructed signals. However, nonuniform sampling tends to turn the regular aliasing artifacts into noise, which is less distracting to the human visual system. In frequency space, the copies of the sampled signal end up being randomly shifted as well, so that when reconstruction is performed the result is random error rather than coherent aliasing.

### Adaptive Sampling

Another approach that has been suggested to combat aliasing is *adaptive supersampling*: if we can identify the regions of the signal with frequencies higher than the Nyquist limit, we can take additional samples in those regions without needing to incur the computational expense of increasing the sampling frequency everywhere. It can be difficult to get this approach to work well in practice, because finding all of the places where supersampling is needed is difficult. Most techniques for doing so are based on examining adjacent sample values and finding places where there is a significant change in value between the two; the assumption is that the signal has high frequencies in that region.

In general, adjacent sample values cannot tell us with certainty what is really happening between them: even if the values are the same, the functions may have huge variation between them. Alternatively, adjacent samples may have substantially different values without any aliasing actually being present. For example, the texture-filtering algorithms in Chapter 10 work hard to eliminate aliasing due to image maps and procedural textures on surfaces in the scene; we would not want an adaptive sampling routine to needlessly take extra samples in an area where texture values are changing quickly but no excessively high frequencies are actually present.

### Prefiltering

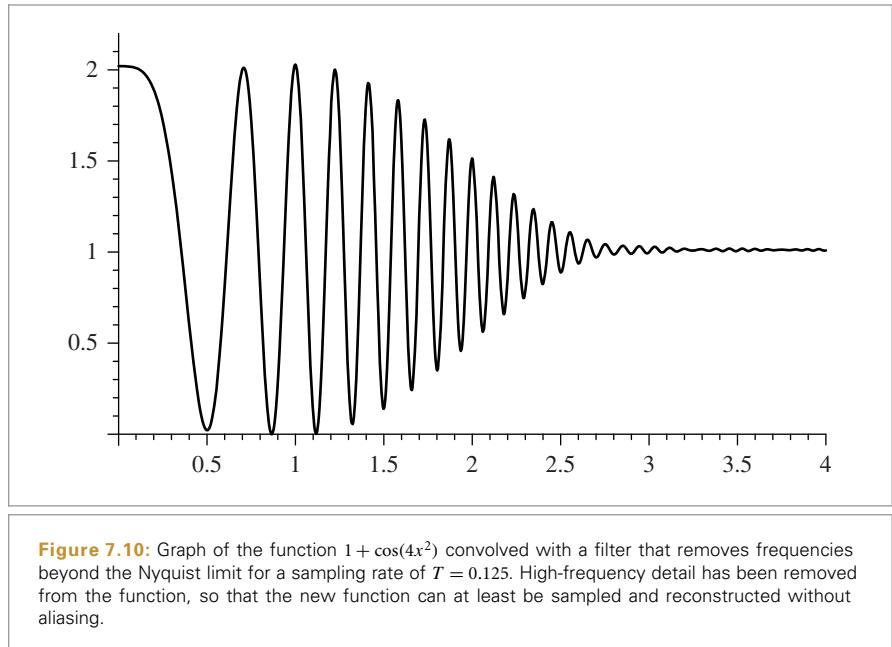
Another approach to eliminating aliasing that sampling theory offers is to filter (i.e., blur) the original function so that no high frequencies remain that can't be captured accurately at the sampling rate being used. This approach is applied in the texture functions of Chapter 10. While this technique changes the character of the function being sampled by removing information from it, blurring is generally less objectionable than aliasing.

Recall that we would like to multiply the original function's spectrum with a box filter with width chosen so that frequencies above the Nyquist limit are removed. In the spatial domain, this corresponds to convolving the original function with a sinc filter,

$$f(x) \otimes \text{sinc}(2\omega_s x).$$

In practice, we can use a filter with finite extent that works well. The frequency space representation of this filter can help clarify how well it approximates the behavior of the ideal sinc filter.

Figure 7.10 shows the function  $1 + \cos(4x^2)$  convolved with a variant of the sinc with finite extent that will be introduced in Section 7.8. Note that the high-frequency details



have been eliminated; this function can be sampled and reconstructed at the sampling rate used in Figure 7.9 without aliasing.

### 7.1.5 APPLICATION TO IMAGE SYNTHESIS

The application of these ideas to the 2D case of sampling and reconstructing images of rendered scenes is straightforward: we have an image, which we can think of as a function of 2D  $(x, y)$  image locations to radiance values  $L$ :

$$f(x, y) \rightarrow L.$$

The good news is that, with our ray tracer, we can evaluate this function at any  $(x, y)$  point that we choose. The bad news is that it's not generally possible to prefilter  $f$  to remove the high frequencies from it before sampling. Therefore, the samplers in this chapter will use both strategies of increasing the sampling rate beyond the basic pixel spacing in the final image as well as nonuniformly distributing the samples to turn aliasing into noise.

It is useful to generalize the definition of the scene function to a higher dimensional function that also depends on the time  $t$  and  $(u, v)$  lens position at which it is sampled. Because the rays from the camera are based on these five quantities, varying any of them gives a different ray and thus a potentially different value of  $f$ . For a particular image position, the radiance at that point will generally vary across both time (if there are moving objects in the scene) and position on the lens (if the camera has a finite-aperture lens).

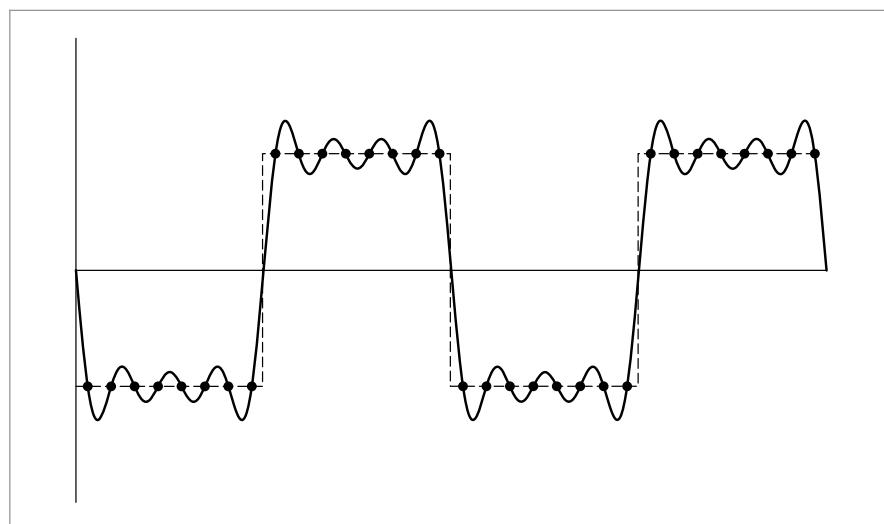
Even more generally, because many of the integrators defined in Chapters 14 through 16 use statistical techniques to estimate the radiance along a given ray, they may return a different radiance value when repeatedly given the same ray. If we further extend the scene radiance function to include sample values used by the integrator (e.g., values used to choose points on area light sources for illumination computations), we have an even higher dimensional image function

$$f(x, y, t, u, v, i_1, i_2, \dots) \rightarrow L.$$

Sampling all of these dimensions well is an important part of generating high-quality imagery efficiently. For example, if we ensure that nearby  $(x, y)$  positions on the image tend to have dissimilar  $(u, v)$  positions on the lens, the resulting rendered images will have less error because each sample is more likely to account for information about the scene that its neighboring samples do not. The Sampler classes in the next few sections will address the issue of sampling all of these dimensions effectively.

### 7.1.6 SOURCES OF ALIASING IN RENDERING

Geometry is one of the most common causes of aliasing in rendered images. When projected onto the image plane, an object's boundary introduces a step function—the image function's value instantaneously jumps from one value to another. Not only do step functions have infinite frequency content as mentioned earlier, but, even worse, the perfect reconstruction filter causes artifacts when applied to aliased samples: ringing artifacts appear in the reconstructed function, an effect known as the *Gibbs phenomenon*. Figure 7.11 shows an example of this effect for a 1D function. Choosing an effective



**Figure 7.11: Illustration of the Gibbs Phenomenon.** When a function hasn't been sampled at the Nyquist rate and the set of aliased samples is reconstructed with the sinc filter, the reconstructed function will have “ringing” artifacts, where it oscillates around the true function. Here a 1D step function (dashed line) has been sampled with a sample spacing of 0.125. When reconstructed with the sinc, the ringing appears (solid line).

reconstruction filter in the face of aliasing requires a mix of science, artistry, and personal taste, as we will see later in this chapter.

Very small objects in the scene can also cause geometric aliasing. If the geometry is small enough that it falls between samples on the image plane, it can unpredictably disappear and reappear over multiple frames of an animation.

Another source of aliasing can come from the texture and materials on an object. *Shading aliasing* can be caused by texture maps that haven't been filtered correctly (addressing this problem is the topic of much of Chapter 10) or from small highlights on shiny surfaces. If the sampling rate is not high enough to sample these features adequately, aliasing will result. Furthermore, a sharp shadow cast by an object introduces another step function in the final image. While it is possible to identify the position of step functions from geometric edges on the image plane, detecting step functions from shadow boundaries is more difficult.

The key insight about aliasing in rendered images is that we can never remove all of its sources, so we must develop techniques to mitigate its impact on the quality of the final image.

### 7.1.7 UNDERSTANDING PIXELS

There are two ideas about pixels that are important to keep in mind throughout the remainder of this chapter. First, it is crucial to remember that the pixels that constitute an image are point samples of the image function at discrete points on the image plane; there is no “area” associated with a pixel. As Alvy Ray Smith (1995) has emphatically pointed out, thinking of pixels as small squares with finite area is an incorrect mental model that leads to a series of errors. By introducing the topics of this chapter with a signal processing approach, we have tried to lay the groundwork for a more accurate mental model.

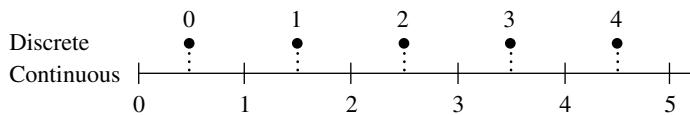
The second issue is that the pixels in the final image are naturally defined at discrete integer  $(x, y)$  coordinates on a pixel grid, but the Samplers in this chapter generate image samples at continuous floating-point  $(x, y)$  positions. The natural way to map between these two domains is to round continuous coordinates to the nearest discrete coordinate; this is appealing since it maps continuous coordinates that happen to have the same value as discrete coordinates to that discrete coordinate. However, the result is that given a set of discrete coordinates spanning a range  $[x_0, x_1]$ , the set of continuous coordinates that covers that range is  $[x_0 - 1/2, x_1 + 1/2]$ . Thus, any code that generates continuous sample positions for a given discrete pixel range is littered with  $1/2$  offsets. It is easy to forget some of these, leading to subtle errors.

If we instead truncate continuous coordinates  $c$  to discrete coordinates  $d$  by

$$d = \lfloor c \rfloor,$$

and convert from discrete to continuous by

$$c = d + 1/2,$$



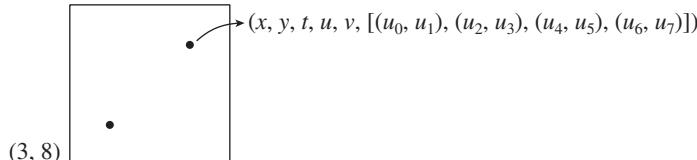
**Figure 7.12:** Pixels in an image can be addressed with either *discrete* or *continuous* coordinates. A discrete image five pixels wide covers the continuous pixel range  $[0, 5]$ . A particular discrete pixel  $d$ 's coordinate in the continuous representation is  $d + 1/2$ .

then the range of continuous coordinates for the discrete range  $[x_0, x_1]$  is naturally  $[x_0, x_1 + 1)$  and the resulting code is much simpler (Heckbert 1990a). This convention, which we will adopt in pbrt, is shown graphically in Figure 7.12.

## 7.2 SAMPLING INTERFACE

As first introduced in Section 7.1.5, the rendering approach implemented in pbrt involves choosing sample points in additional dimensions beyond 2D points on the image plane. Various algorithms will be used to generate these points, but all of their implementations inherit from an abstract `Sampler` class that defines their interface. The core sampling declarations and functions are in the files `core/sampler.h` and `core/sampler.cpp`. Each of the sample generation implementations is in its own source files in the `samplers/` directory.

The task of a `Sampler` is to generate a sequence of  $n$ -dimensional samples in  $[0, 1]^n$ , where one such sample vector is generated for each image sample and where the number of dimensions  $n$  in each sample may vary, depending on the calculations performed by the light transport algorithms. (See Figure 7.13.)



**Figure 7.13:** Samplers generate an  $n$ -dimensional sample vector for each of the image samples taken to generate the final image. Here, the pixel  $(3, 8)$  is being sampled, and there are two image samples in the pixel area. The first two dimensions of the sample vector give the  $(x, y)$  offset of the sample within the pixel, and the next three dimensions determine the time and lens position of the corresponding camera ray. Subsequent dimensions are used by the Monte Carlo light transport algorithms in Chapters 14, 15, and 16. Here, the light transport algorithm has requested a 2D array of four samples in the sample vector; these values might be used to select four points on an area light source to compute the radiance for the image sample, for example.

Because sample values must be strictly less than 1, it's useful to define a constant, `OneMinusEpsilon`, that represents the largest representable floating-point constant that is less than 1. Later, we will clamp sample vector values to be no larger than this value.

```
(Random Number Declarations) ≡
#ifndef PBRT_FLOAT_IS_DOUBLE
static const Float OneMinusEpsilon = 0x1.fffffffffffffp-1;
#else
static const Float OneMinusEpsilon = 0x1.fffffep-1;
#endif
```

The simplest possible implementation of a `Sampler` would just return uniform random values in [0, 1) each time an additional component of the sample vector was needed. Such a sampler would produce correct images but would require many more samples (and thus, many more rays traced and much more time) to create images of the same quality achievable with more sophisticated samplers. The run-time expense for using better sampling patterns is approximately the same as that for lower-quality patterns like uniform random numbers; because evaluating the radiance for each image sample is much more expensive than computing the sample's component values, doing this work pays dividends (Figure 7.14).

A few characteristics of these sample vectors are assumed in the following:

- The first five dimensions generated by `Samplers` are generally used by the `Camera`. In this case, the first two are specifically used to choose a point on the image inside the current pixel area; the third is used to compute the time at which the sample should be taken; and the fourth and fifth dimensions give a  $(u, v)$  lens position for depth of field.
- Some sampling algorithms generate better samples in some dimensions of the sample vector than in others. Elsewhere in the system, we assume that in general, the earlier dimensions have the most well-placed sample values.

Note also that the  $n$ -dimensional samples generated by the `Sampler` are generally not represented explicitly or stored in their entirety but are often generated incrementally as needed by the light transport algorithm. (However, storing the entire sample vector and making incremental changes to its components is the basis of the `MLTSampler` in Section 16.4.4, which is used by the `MLTIntegrator` in Section 16.4.5.)

### \* 7.2.1 EVALUATING SAMPLE PATTERNS: DISCREPANCY

Fourier analysis gave us one way of evaluating the quality of a 2D sampling pattern, but it took us only as far as being able to quantify the improvement from adding more evenly spaced samples in terms of the band-limited frequencies that could be represented. Given the presence of infinite frequency content from edges in images and given the need for ( $n > 2$ )-dimensional sample vectors for Monte Carlo light transport algorithms, Fourier analysis alone isn't enough for our needs.

Given a renderer and a candidate algorithm for placing samples, one way to evaluate the algorithm's effectiveness is to use that sampling pattern to render an image and to compute the error in the image compared to a reference image rendered with a large

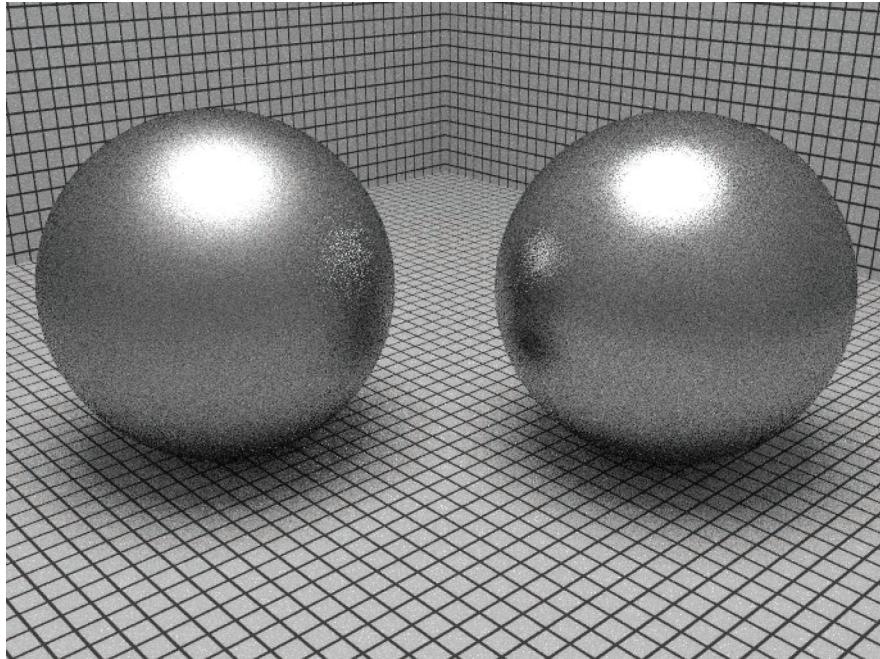
`Camera` 356

`Float` 1062

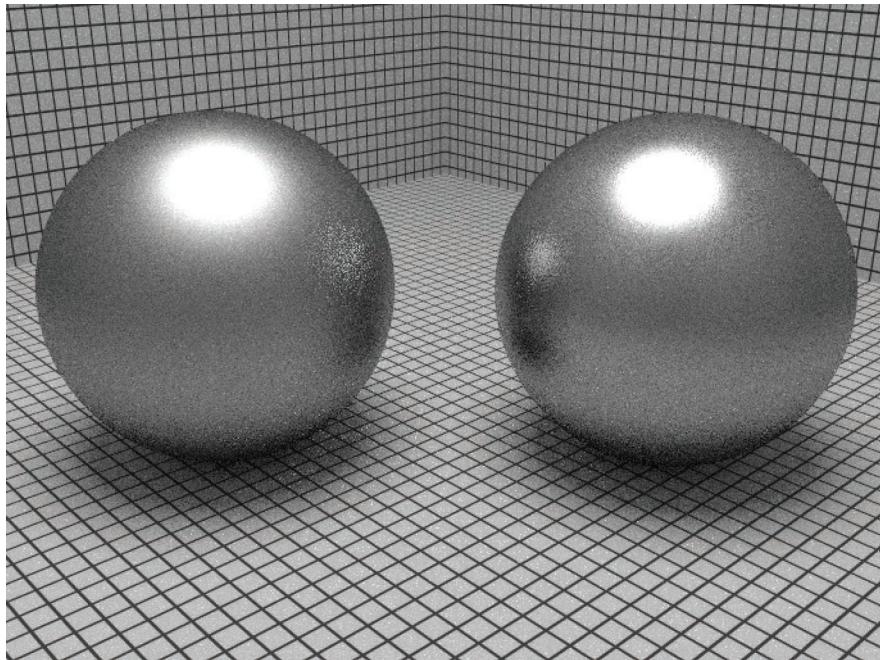
`MLTIntegrator` 1035

`MLTSampler` 1029

`OneMinusEpsilon` 417

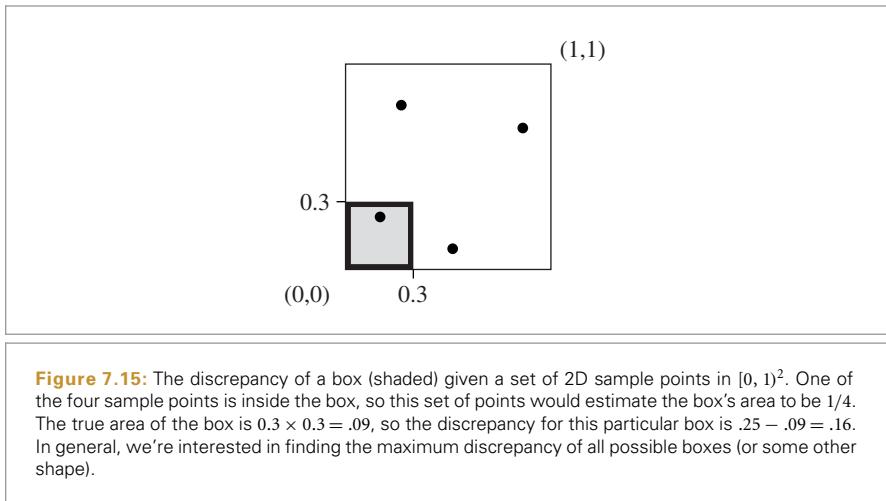


(a)



(b)

**Figure 7.14:** Scene rendered with (a) a relatively ineffective sampler and (b) a carefully designed sampler, using the same number of samples for each. The improvement in image quality, ranging from the edges of the highlights to the quality of the glossy reflections, is noticeable.



number of samples. We will use this approach to compare sampling algorithms later in this chapter, though it only tells us how well the algorithm did for one specific scene, and it doesn't give us a sense of the quality of the sample points without going through the rendering process.

Outside of Fourier analysis, mathematicians have developed a concept called *discrepancy* that can be used to evaluate the quality of a pattern of  $n$ -dimensional sample positions. Patterns that are well distributed (in a manner to be formalized shortly) have low discrepancy values, and thus the sample pattern generation problem can be considered to be one of finding a suitable *low-discrepancy* pattern of points.<sup>5</sup> A number of deterministic techniques have been developed that generate low-discrepancy point sets, even in high-dimensional spaces. (Most of the sampling algorithms used later in this chapter use these techniques.)

The basic idea of discrepancy is that the quality of a set of points in an  $n$ -dimensional space  $[0, 1]^n$  can be evaluated by looking at regions of the domain  $[0, 1]^n$ , counting the number of points inside each region, and comparing the volume of each region to the number of sample points inside. In general, a given fraction of the volume should have roughly the same fraction of the total number of sample points inside of it. While it's not possible for this always to be the case, we can still try to use patterns that minimize the maximum difference between the actual volume and the volume estimated by the points (the *discrepancy*). Figure 7.15 shows an example of the idea in two dimensions.

To compute the discrepancy of a set of points, we first pick a family of shapes  $B$  that are subsets of  $[0, 1]^n$ . For example, boxes with one corner at the origin are often used. This

<sup>5</sup> Of course, using discrepancy in this way implicitly assumes that the metric used to compute discrepancy is one that has good correlation with the quality of a pattern for image sampling, which may be a slightly different thing, particularly given the involvement of the human visual system in the process.

corresponds to

$$B = \{[0, v_1] \times [0, v_2] \times \cdots \times [0, v_n]\},$$

where  $0 \leq v_i < 1$ . Given a sequence of sample points  $P = x_1, \dots, x_N$ , the discrepancy of  $P$  with respect to  $B$  is<sup>6</sup>

$$D_N(B, P) = \sup_{b \in B} \left| \frac{\#\{x_i \in b\}}{N} - V(b) \right|, \quad (7.4)$$

where  $\#\{x_i \in b\}$  is the number of points in  $b$  and  $V(b)$  is the volume of  $b$ .

The intuition for why Equation (7.4) is a reasonable measure of quality is that the value  $\#\{x_i \in b\}/N$  is an approximation of the volume of the box  $b$  given by the particular points  $P$ . Therefore, the discrepancy is the worst error over all possible boxes from this way of approximating volume. When the set of shapes  $B$  is the set of boxes with a corner at the origin, this value is called the *star discrepancy*,  $D_N^*(P)$ . Another popular option for  $B$  is the set of all axis-aligned boxes, where the restriction that one corner be at the origin has been removed.

For a few particular point sets, the discrepancy can be computed analytically. For example, consider the set of points in one dimension

$$x_i = \frac{i}{N}.$$

We can see that the star discrepancy of  $x_i$  is

$$D_N^*(x_1, \dots, x_n) = \frac{1}{N}.$$

For example, take the interval  $b = [0, 1/N]$ . Then  $V(b) = 1/N$ , but  $\#\{x_i \in b\} = 0$ . This interval (and the intervals  $[0, 2/N]$ , etc.) is the interval where the largest differences between volume and fraction of points inside the volume are seen.

The star discrepancy of this sequence can be improved by modifying it slightly:

$$x_i = \frac{i - \frac{1}{2}}{N}. \quad (7.5)$$

Then

$$D_N^*(x_i) = \frac{1}{2N}.$$

The bounds for the star discrepancy of a sequence of points in one dimension have been shown to be

$$D_N^*(x_i) = \frac{1}{2N} + \max_{1 \leq i \leq N} \left| x_i - \frac{2i - 1}{2N} \right|.$$

Thus, the earlier sequence from Equation (7.5) has the lowest possible discrepancy for a sequence in 1D. In general, it is much easier to analyze and compute bounds for the

<sup>6</sup> The sup operator is the continuous analog of the discrete max operator. That is,  $\sup f(x)$  is a constant-valued function of  $x$  that passes through the maximum value taken on by  $f(x)$ .

discrepancy of sequences in 1D than for those in higher dimensions. For less simply constructed point sequences and for sequences in higher dimensions and for more irregular shapes than boxes, the discrepancy often must be estimated numerically by constructing a large number of shapes  $b$ , computing their discrepancy, and reporting the maximum value found.

The astute reader will notice that according to the low-discrepancy measure, this uniform sequence in 1D is optimal, but earlier in this chapter we claimed that irregular jittered patterns were perceptually superior to uniform patterns for image sampling in 2D since they replaced aliasing error with noise. In that framework, uniform samples are clearly not optimal. Fortunately, low-discrepancy patterns in higher dimensions are much less uniform than they are in one dimension and thus usually work reasonably well as sample patterns in practice. Nevertheless, their underlying uniformity means that low-discrepancy patterns can be more prone to visually objectionable aliasing than patterns with pseudo-random variation.

Discrepancy alone isn't necessarily a good metric: some low-discrepancy point sets exhibit some clumping of samples, where two or more samples may be quite close together. The Sobol' sampler in Section 7.7 particularly suffers from this issue—see Figure 7.36, which shows a plot of its first two dimensions. Intuitively, samples that are too close together aren't a good use of sampling resources: the closer one sample is to another, the less likely it is to give useful new information about the function being sampled. Therefore, computing the minimum distance between any two samples in a set of points has also proved to be a useful metric of sample pattern quality; the higher the minimum distance, the better.

There are a variety of algorithms for generating *Poisson disk* sampling patterns that score well by this metric. By construction, no two points in a Poisson disk pattern are closer than some distance  $d$ . Studies have shown that the rods and cones in the eye are distributed in a similar way, which further validates the idea that this distribution is a good one for imaging. In practice, we have found that Poisson disk patterns work very well for sampling 2D images but are less effective than the better low discrepancy patterns for the higher-dimensional sampling done in more complex rendering situations; see the “Further Reading” section for more information.

### 7.2.2 BASIC SAMPLER INTERFACE

The `Sampler` base class not only defines the interface to samplers but also provides some common functionality for use by `Sampler` implementations.

```
<Sampler Declarations> ≡
  class Sampler {
    public:
      <Sampler Interface 422>
      <Sampler Public Data 422>
    protected:
      <Sampler Protected Data 425>
    private:
      <Sampler Private Data 426>
  };
```

All Sampler implementations must supply the constructor with the number of samples that will be generated for each pixel in the final image. In rare cases, it may be useful for the system to model the film as having only a single “pixel” that covers the entire viewing region. (This overloading of the definition of pixel is something of a stretch, but we allow it to simplify certain implementation aspects.) Since this “pixel” could potentially have billions of samples, we store the sample count using a variable with 64 bits of precision.

*(Sampler Method Definitions) ≡*

```
Sampler::Sampler(int64_t samplesPerPixel)
    : samplesPerPixel(samplesPerPixel) { }
```

*(Sampler Public Data) ≡*

```
const int64_t samplesPerPixel;
```

421

When the rendering algorithm is ready to start work on a given pixel, it starts by calling `StartPixel()`, providing the coordinates of the pixel in the image. Some Sampler implementations use the knowledge of which pixel is being sampled to improve the overall distribution of the samples that they generate for the pixel, while others ignore this information.

*(Sampler Interface) ≡*

421

```
virtual void StartPixel(const Point2i &p);
```

The `Get1D()` method returns the sample value for the next dimension of the current sample vector, and `Get2D()` returns the sample values for the next two dimensions. While a 2D sample value could be constructed by using values returned by a pair of calls to `Get1D()`, some samplers can generate better point distributions if they know that two dimensions will be used together.

*(Sampler Interface) +≡*

421

```
virtual Float Get1D() = 0;
virtual Point2f Get2D() = 0;
```

In pbrt, we don’t support requests for 3D or higher dimensional sample values from samplers because these are generally not needed for the types of rendering algorithms implemented here. If necessary, multiple values from lower dimensional components can be used to construct higher dimensional sample points.

A sharp edge of these interfaces is that code that uses sample values must be carefully written so that it always requests sample dimensions in the same order. Consider the following code:

```
sampler->StartPixel(p);
do {
    Float v = a(sampler->Get1D());
    if (v > 0)
        v += b(sampler->Get1D());
    v += c(sampler->Get1D());
} while (sampler->StartNextSample());
```

Float 1062  
 Point2f 68  
 Point2i 68  
 Sampler 421  
 Sampler::samplesPerPixel 422  
 Sampler::StartNextSample()  
 424

In this case, the first dimension of the sample vector will always be passed to the function `a()`; when the code path that calls `b()` is executed, `b()` will receive the second dimension.

However, if the `if` test isn't always true or false, then `c()` will sometimes receive a sample from the second dimension of the sample vector and otherwise receive a sample from the third dimension. Thus, efforts by the sampler to provide well-distributed sample points in each dimension being evaluated have been thwarted. Code that uses Samplers should therefore be carefully written so that it consistently consumes sample vector dimensions to avoid this issue.

For convenience, the Sampler base class provides a method that initializes a CameraSample for a given pixel.

```
<Sampler Method Definitions> +≡
    CameraSample Sampler::GetCameraSample(const Point2i &pRaster) {
        CameraSample cs;
        cs.pFilm = (Point2f)pRaster + Get2D();
        cs.time = Get1D();
        cs.pLens = Get2D();
        return cs;
    }
```

Some rendering algorithms make use of arrays of sample values for some of the dimensions they sample; most sample-generation algorithms can generate higher quality arrays of samples than series of individual samples by accounting for the distribution of sample values across all elements of the array and across the samples in a pixel.

If arrays of samples are needed, they must be requested before rendering begins. The `Request[12]DArray()` methods should be called for each such dimension's array before rendering begins—for example, in methods that override the `SamplerIntegrator::Preprocess()` method. For example, in a scene with two area light sources, where the integrator traces four shadow rays to the first source and eight to the second, the integrator would ask for two 2D sample arrays for each image sample, with four and eight samples each. (A 2D array is required because two dimensions are needed to parameterize the surface of a light.) In Section 13.7, we will see how using arrays of samples corresponds to more densely sampling some of the dimensions of the light transport integral using the Monte Carlo technique of “splitting.”

`CameraSample` 357  
`CameraSample::pFilm` 357  
`CameraSample::pLens` 357  
`CameraSample::time` 357  
`Point2f` 68  
`Point2i` 68  
`Sampler` 421  
`Sampler::Get1D()` 422  
`Sampler::Get2D()` 422  
`Sampler::RoundCount()` 424  
`SamplerIntegrator::Preprocess()` 26  
`ZeroTwoSequenceSampler` 462

*<Sampler Interface>* +≡
 void Request1DArray(int n);
 void Request2DArray(int n);

421

Most Samplers can do a better job of generating some particular sizes of these arrays than others. For example, samples from the `ZeroTwoSequenceSampler` are much better distributed in quantities that are in powers of 2. The `Sampler::RoundCount()` method helps communicate this information. Code that needs arrays of samples should call this method with the desired number of samples to be taken, giving the Sampler an opportunity to adjust the number of samples to a better number. The returned value should then be used as the number of samples to actually request from the Sampler. The default implementation returns the given count unchanged.

*(Sampler Interface) +≡*

```
virtual int RoundCount(int n) const {
    return n;
}
```

421

During rendering, the `Get[12]DArray()` methods can be called to get a pointer to the start of a previously requested array of samples for the current dimension. Along the lines of `Get1D()` and `Get2D()`, these return a pointer to an array of samples whose size is given by the parameter `n` to the corresponding call to `Request[12]DArray()` during initialization. The caller must also provide the array size to the “get” method, which is used to verify that the returned buffer has the expected size.

*(Sampler Interface) +≡*

```
const Float *Get1DArray(int n);
const Point2f *Get2DArray(int n);
```

421

When the work for one sample is complete, the integrator calls `StartNextSample()`. This call notifies the Sampler that subsequent requests for sample components should return values starting at the first dimension of the next sample for the current pixel. This method returns true until the number of the originally requested samples per pixel have been generated (at which point the caller should either start work on another pixel or stop trying to use more samples.)

*(Sampler Interface) +≡*

```
virtual bool StartNextSample();
```

421

Sampler implementations store a variety of state about the current sample: which pixel is being sampled, how many dimensions of the sample have been used, and so forth. It is therefore natural for it to be unsafe for a single Sampler to be used concurrently by multiple threads. The `Clone()` method generates a new instance of an initial Sampler for use by a rendering thread; it takes a seed value for the sampler’s random number generator (if any), so that different threads see different sequences of random numbers. Reusing the same pseudo-random number sequence across multiple image tiles can lead to subtle image artifacts, such as repeating noise patterns.

Implementations of the various `Clone()` methods aren’t generally interesting, so they won’t be included in the text here.

*(Sampler Interface) +≡*

```
virtual std::unique_ptr<Sampler> Clone(int seed) = 0;
```

421

Some light transport algorithms (notably stochastic progressive photon mapping in Section 16.2) don’t use all of the samples in a pixel before going to the next pixel, but instead jump around pixels, taking one sample at a time in each one. The `SetSampleNumber()` method allows integrators to set the index of the sample in the current pixel to generate next. This method returns false once `sampleNum` is greater than or equal to the number of originally requested samples per pixel.

*(Sampler Interface) +≡*

```
virtual bool SetSampleNumber(int64_t sampleNum);
```

421

Float 1062

Point2f 68

Sampler 421

### 7.2.3 SAMPLER IMPLEMENTATION

The Sampler base class provides implementations of some of the methods in its interface. First, the `StartPixel()` method implementation records the coordinates of the current pixel being sampled and resets `currentPixelSampleIndex`, the sample number in the pixel currently being generated, to zero. Note that this is a virtual method with an implementation; subclasses that override this method are required to explicitly call `Sampler::StartPixel()`.

*(Sampler Method Definitions) +≡*

```
void Sampler::StartPixel(const Point2i &p) {
    currentPixel = p;
    currentPixelSampleIndex = 0;
    <Reset array offsets for next pixel sample 426>
}
```

The current pixel coordinates and sample number within the pixel are made available to Sampler subclasses, though they should treat these as read-only values.

*(Sampler Protected Data) ≡*

```
Point2i currentPixel;
int64_t currentPixelSampleIndex;
```

421

When the pixel sample is advanced or explicitly set, `currentPixelSampleIndex` is updated accordingly. As with `StartPixel()`, the methods `StartNextSample()` and `SetSampleNumber()` are both virtual implementations; these implementations also must be explicitly called by overridden implementations of them in Sampler subclasses.

*(Sampler Method Definitions) +≡*

```
bool Sampler::StartNextSample() {
    <Reset array offsets for next pixel sample 426>
    return ++currentPixelSampleIndex < samplesPerPixel;
}
```

*(Sampler Method Definitions) +≡*

```
bool Sampler::SetSampleNumber(int64_t sampleNum) {
    <Reset array offsets for next pixel sample 426>
    currentPixelSampleIndex = sampleNum;
    return currentPixelSampleIndex < samplesPerPixel;
}
```

Point2i 68  
 Sampler 421  
`Sampler::currentPixel` 425  
`Sampler:::currentPixelSampleIndex` 425  
`Sampler::samplesPerPixel` 422  
`Sampler::SetSampleNumber()` 424  
`Sampler::StartNextSample()` 424  
`Sampler::StartPixel()` 422

The base Sampler implementation also takes care of recording requests for arrays of sample components and allocating storage for their values. The sizes of the requested sample arrays are stored in `samples1DArraySizes` and `samples2DArraySizes`, and memory for an entire pixel's worth of array samples is allocated in `sampleArray1D` and `sampleArray2D`. The first  $n$  values in each allocation are used for the corresponding array for the first sample in the pixel, and so forth.

```
(Sampler Method Definitions) +≡
void Sampler::Request1DArray(int n) {
    samples1DArraySizes.push_back(n);
    sampleArray1D.push_back(std::vector<Float>(n * samplesPerPixel));
}
```

```
(Sampler Method Definitions) +≡
void Sampler::Request2DArray(int n) {
    samples2DArraySizes.push_back(n);
    sampleArray2D.push_back(std::vector<Point2f>(n * samplesPerPixel));
}
```

(Sampler Protected Data) +≡

```
421
std::vector<int> samples1DArraySizes, samples2DArraySizes;
std::vector<std::vector<Float>> sampleArray1D;
std::vector<std::vector<Point2f>> sampleArray2D;
```

As arrays in the current sample are accessed by the Get[12]DArray() methods, array1DOffset and array2DOffset are updated to hold the index of the next array to return for the sample vector.

(Sampler Private Data) ≡

```
421
size_t array1DOffset, array2DOffset;
```

When a new pixel is started or when the sample number in the current pixel changes, these array offsets must be reset to 0.

(Reset array offsets for next pixel sample) ≡

```
425
array1DOffset = array2DOffset = 0;
```

Returning the appropriate array pointer is a matter of first choosing the appropriate array based on how many have been consumed in the current sample vector and then returning the appropriate instance of it based on the current pixel sample index.

(Sampler Method Definitions) +≡

```
const Float *Sampler::Get1DArray(int n) {
    if (array1DOffset == sampleArray1D.size())
        return nullptr;
    return &sampleArray1D[array1DOffset++][currentPixelSampleIndex * n];
}
```

(Sampler Method Definitions) +≡

```
const Point2f *Sampler::Get2DArray(int n) {
    if (array2DOffset == sampleArray2D.size())
        return nullptr;
    return &sampleArray2D[array2DOffset++][currentPixelSampleIndex * n];
}
```

```
Float 1062
Point2f 68
Sampler::array1DOffset 426
Sampler::array2DOffset 426
Sampler::
    currentPixelSampleIndex
    425
Sampler::Get1DArray() 424
Sampler::Get2DArray() 424
Sampler::Request1DArray()
    423
Sampler::Request2DArray()
    423
Sampler::sampleArray1D 426
Sampler::sampleArray2D 426
Sampler::samples1DArraySizes
    426
Sampler::samples2DArraySizes
    426
Sampler::samplesPerPixel 422
```

### 7.2.4 PIXEL SAMPLER

While some sampling algorithms can easily incrementally generate elements of each sample vector, others more naturally generate all of the dimensions' sample values for all of the sample vectors for a pixel at the same time. The `PixelSampler` class implements some functionality that is useful for the implementation of these types of samplers.

```
<Sampler Declarations> +≡
    class PixelSampler : public Sampler {
        public:
            <PixelSampler Public Methods>
        protected:
            <PixelSampler Protected Data 427>
    };
```

The number of dimensions of the sample vectors that will be used by the rendering algorithm isn't known ahead of time. (Indeed, it's only determined implicitly by the number of `Get1D()` and `Get2D()` calls and the requested arrays.) Therefore, the `PixelSampler` constructor takes a maximum number of dimensions for which non-array sample values will be computed by the `Sampler`. If all of these dimensions of components are consumed, then the `PixelSampler` just returns uniform random values for additional dimensions.

For each precomputed dimension, the constructor allocates a vector to store sample values, with one value for each sample in the pixel. These vectors are indexed as `sample1D[dim][pixelSample]`; while interchanging the order of these indices might seem more sensible, this memory layout—where all of the sample component values for all of the samples for a given dimension are contiguous in memory—turns out to be more convenient for code that generates these values.

```
<Sampler Method Definitions> +≡
PixelSampler::PixelSampler(int64_t samplesPerPixel,
    int nSampledDimensions)
: Sampler(samplesPerPixel) {
    for (int i = 0; i < nSampledDimensions; ++i) {
        samples1D.push_back(std::vector<float>(samplesPerPixel));
        samples2D.push_back(std::vector<Point2f>(samplesPerPixel));
    }
}
```

The key responsibility of `Sampler` implementations that inherit from `PixelSampler` then is to fill in the `samples1D` and `samples2D` arrays (in addition to `sampleArray1D` and `sampleArray2D`) in their `StartPixel()` methods.

`current1DDimension` and `current2DDimension` store the offsets into the respective arrays for the current pixel sample. They must be reset to 0 at the start of each new sample.

*<PixelSampler Protected Data>* ≡

427

```
std::vector<std::vector<float>> samples1D;
std::vector<std::vector<Point2f>> samples2D;
int current1DDimension = 0, current2DDimension = 0;
```

Float 1062

PixelSampler 427

PixelSampler::samples1D 427

PixelSampler::samples2D 427

Point2f 68

Sampler 421

```
(Sampler Method Definitions) +≡
bool PixelSampler::StartNextSample() {
    current1DDimension = current2DDimension = 0;
    return Sampler::StartNextSample();
}

(Sampler Method Definitions) +≡
bool PixelSampler::SetSampleNumber(int64_t sampleNum) {
    current1DDimension = current2DDimension = 0;
    return Sampler::SetSampleNumber(sampleNum);
}
```

Given sample values in the arrays computed by the `PixelSampler` subclass, the implementation of `Get1D()` is just a matter of returning values for successive dimensions until all of the computed dimensions have been consumed, at which point uniform random values are returned.

```
(Sampler Method Definitions) +≡
Float PixelSampler::Get1D() {
    if (current1DDimension < samples1D.size())
        return samples1D[current1DDimension++][currentPixelSampleIndex];
    else
        return rng.UniformFloat();
}
```

The `PixelSampler::Get2D()` follows similarly, so it won't be included here.

The random number generator used by the `PixelSampler` is protected rather than `private`; this is a convenience for some of its subclasses that also need random numbers when they initialize `samples1D` and `samples2D`.

*(PixelSampler Protected Data)* +≡

427

RNG rng;

## 7.2.5 GLOBAL SAMPLER

Other algorithms for generating samples are very much not pixel-based but naturally generate consecutive samples that are spread across the entire image, visiting completely different pixels in succession. (Many such samplers are effectively placing each additional sample such that it fills the biggest hole in the  $n$ -dimensional sample space, which naturally leads to subsequent samples being inside different pixels.) These sampling algorithms are somewhat problematic with the `Sampler` interface as described so far: consider, for example, a sampler that generates the series of sample values shown in the middle column of Table 7.2 for the first two dimensions. These sample values are multiplied by the image resolution in each dimension to get sample positions in the image plane (here we're considering a  $2 \times 3$  image for simplicity.) Note that for the sampler here (actually the `HaltonSampler`), each pixel is visited by each sixth sample. If we are rendering an image with three samples per pixel, then to generate all of the samples for the pixel  $(0, 0)$ , we need to generate the samples with indices 0, 6, and 12, and so forth.

Float 1062  
 HaltonSampler 450  
`PixelSampler::`  
 `current1DDimension`  
`427`  
`PixelSampler::`  
 `current2DDimension`  
`427`  
`PixelSampler::rng` 428  
`PixelSampler::samples1D` 427  
 RNG 1065  
`RNG::UniformFloat()` 1066  
`Sampler::`  
 `currentPixelSampleIndex`  
`425`  
`Sampler::SetSampleNumber()`  
`424`  
`Sampler::StartNextSample()`  
`424`

**Table 7.2:** The `HaltonSampler` generates the coordinates in the middle column for the first two dimensions. Because it is a `GlobalSampler`, it must define an inverse mapping from the pixel coordinates to sample indices; here, it places samples across a  $2 \times 3$  pixel image, by scaling the first coordinate by 2 and the second coordinate by three, giving the pixel sample coordinates in the right column.

Sample index	$[0, 1]^2$ sample coordinates	Pixel sample coordinates
0	(0.000000, 0.000000)	(0.000000, 0.000000)
1	(0.500000, 0.333333)	(1.000000, 1.000000)
2	(0.250000, 0.666667)	(0.500000, 2.000000)
3	(0.750000, 0.111111)	(1.500000, 0.333333)
4	(0.125000, 0.444444)	(0.250000, 1.333333)
5	(0.625000, 0.777778)	(1.250000, 2.333333)
6	(0.375000, 0.222222)	(0.750000, 0.666667)
7	(0.875000, 0.555556)	(1.750000, 1.666667)
8	(0.062500, 0.888889)	(0.125000, 2.666667)
9	(0.562500, 0.037037)	(1.125000, 0.111111)
10	(0.312500, 0.370370)	(0.625000, 1.111111)
11	(0.812500, 0.703704)	(1.625000, 2.111111)
12	(0.187500, 0.148148)	(0.375000, 0.444444)
:		

Given the existence of such samplers, we could have defined the `Sampler` interface so that it specifies the pixel being rendered for each sample rather than the other way around (i.e., the `Sampler` being told which pixel to render).

However, there were good reasons to adopt the current design: this approach makes it easy to decompose the film into small image tiles for multi-threaded rendering, where each thread computes pixels in a local region that can be efficiently merged into the final image. Thus, we must require that such samplers generate samples out of order, so that all samples for each pixel are generated in succession.

The `GlobalSampler` helps bridge between the expectations of the `Sampler` interface and the natural operation of these types of samplers. It provides implementations of all of the pure virtual `Sampler` methods, implementing them in terms of three new pure virtual methods that its subclasses must implement instead.

```
(Sampler Declarations) +≡
class GlobalSampler : public Sampler {
public:
    (GlobalSampler Public Methods 429)
private:
    (GlobalSampler Private Data 430)
};
```

`GlobalSampler` 429  
`HaltonSampler` 450  
`Sampler` 421

*(GlobalSampler Public Methods) ≡*

```
GlobalSampler(int64_t samplesPerPixel) : Sampler(samplesPerPixel) { }
```

429

There are two methods that implementations must provide. The first one, `GetIndexForSample()`, performs the inverse mapping from the current pixel and given sample index to a global index into the overall set of sample vectors. For example, for the Sampler that generated the values in Table 7.2, if `currentPixel` was  $(0, 2)$ , then `GetIndexForSample(0)` would return 2, since the corresponding pixel sample coordinates for sample index 2,  $(0.25, 0.666667)$  correspond to the first sample that lands in that pixel's area.

*(GlobalSampler Public Methods) +≡* 429  
`virtual int64_t GetIndexForSample(int64_t sampleNum) const = 0;`

Closely related, `SampleDimension()` returns the sample value for the given dimension of the `indexth` sample vector in the sequence. Because the first two dimensions are used to offset into the current pixel, they are handled specially: the value returned by implementations of this method should be the sample offset within the current pixel, rather than the original  $[0, 1]^2$  sample value. For the example in Table 7.2, `SampleDimension(4,1)` would return 0.333333, since the second dimension of the sample with index 4 is that offset into the pixel  $(0, 1)$ .

*(GlobalSampler Public Methods) +≡* 429  
`virtual float SampleDimension(int64_t index, int dimension) const = 0;`

When it's time to start to generate samples for a pixel, it's necessary to reset the dimension of the sample and find the index of the first sample in the pixel. As with all samplers, values for sample arrays are all generated next.

*(Sampler Method Definitions) +≡*  
`void GlobalSampler::StartPixel(const Point2i &p) {`  
 `Sampler::StartPixel(p);`  
 `dimension = 0;`  
 `intervalSampleIndex = GetIndexForSample(0);`  
*(Compute arrayEndDim for dimensions used for array samples 431)*  
*(Compute 1D array samples for GlobalSampler 431)*  
*(Compute 2D array samples for GlobalSampler)*  
`}`

The `dimension` member variable tracks the next dimension that the sampler implementation will be asked to generate a sample value for; it's incremented as `Get1D()` and `Get2D()` are called. `intervalSampleIndex` records the index of the sample that corresponds to the current sample  $s_i$  in the current pixel.

*(GlobalSampler Private Data) ≡* 429  
`int dimension;`  
`int64_t intervalSampleIndex;`

It's necessary to decide which dimensions of the sample vector to use for array samples. Under the assumption that the earlier dimensions will be better quality than later dimensions, it's important to set aside the first few dimensions for the `CameraSample`, since the quality of those sample values often has a large impact on final image quality.

Therefore, the first dimensions up to `arrayStartDim` are devoted to regular 1D and 2D samples, and then the subsequent dimensions are devoted to first 1D and then

`CameraSample` 357  
`float` 1062  
`GlobalSampler::dimension` 430  
`GlobalSampler::`  
 `GetIndexForSample()`  
 `430`  
`GlobalSampler::`  
 `intervalSampleIndex`  
 `430`  
`Point2i` 68  
`Sampler::StartPixel()` 422

2D array samples. Finally, higher dimensions starting at `arrayEndDim` are used for further non-array 1D and 2D samples. It isn't possible to compute `arrayEndDim` when the `GlobalSampler` constructor runs, since array samples haven't been requested yet by the integrators. Therefore, this value is computed (repeatedly and redundantly) in the `StartPixel()` method.

```
(GlobalSampler Private Data) +≡
    static const int arrayStartDim = 5;
    int arrayEndDim;
```

429

The total number of array samples for all pixel samples is given by the product of the number of pixel samples and the requested sample array size.

```
(Compute arrayEndDim for dimensions used for array samples) ≡
    arrayEndDim = arrayStartDim +
        sampleArray1D.size() + 2 * sampleArray2D.size();
```

430

Actually generating the array samples is just a matter of computing the number of needed values in the current sample dimension.

```
(Compute 1D array samples for GlobalSampler) ≡
    for (size_t i = 0; i < samples1DArraySizes.size(); ++i) {
        int nSamples = samples1DArraySizes[i] * samplesPerPixel;
        for (int j = 0; j < nSamples; ++j) {
            int64_t index = GetIndexForSample(j);
            sampleArray1D[i][j] =
                SampleDimension(index, arrayStartDim + i);
        }
    }
```

`GlobalSampler::arrayEndDim` 431  
`GlobalSampler::arrayStartDim` 431  
`GlobalSampler::dimension` 430  
`GlobalSampler::GetIndexForSample()` 430  
`GlobalSampler::intervalSampleIndex` 430  
`GlobalSampler::SampleDimension()` 430  
`Sampler::currentPixelSampleIndex` 425  
`Sampler::sampleArray1D` 426  
`Sampler::sampleArray2D` 426  
`Sampler::samples1DArraySizes` 426  
`Sampler::samplesPerPixel` 422  
`Sampler::SetSampleNumber()` 424  
`Sampler::StartNextSample()` 424

430

The 2D sample arrays are generated analogously; the `(Compute 2D array samples for GlobalSampler)` fragment isn't included here.

When the pixel sample changes, it's necessary to reset the current sample dimension counter and to compute the sample index for the next sample inside the pixel.

```
(Sampler Method Definitions) +≡
    bool GlobalSampler::StartNextSample() {
        dimension = 0;
        intervalSampleIndex = GetIndexForSample(currentPixelSampleIndex + 1);
        return Sampler::StartNextSample();
    }

(Sampler Method Definitions) +≡
    bool GlobalSampler::SetSampleNumber(int64_t sampleNum) {
        dimension = 0;
        intervalSampleIndex = GetIndexForSample(sampleNum);
        return Sampler::SetSampleNumber(sampleNum);
    }
```

Given this machinery, getting regular 1D sample values is just a matter of skipping over the dimensions allocated to array samples and passing the current sample index and dimension to the implementation’s `SampleDimension()` method.

```
(Sampler Method Definitions) +≡
Float GlobalSampler::Get1D() {
    if (dimension >= arrayStartDim && dimension < arrayEndDim)
        dimension = arrayEndDim;
    return SampleDimension(intervalSampleIndex, dimension++);
}
```

2D samples follow analogously.

```
(Sampler Method Definitions) +≡
Point2f GlobalSampler::Get2D() {
    if (dimension + 1 >= arrayStartDim && dimension < arrayEndDim)
        dimension = arrayEndDim;
    Point2f p(SampleDimension(intervalSampleIndex, dimension),
              SampleDimension(intervalSampleIndex, dimension + 1));
    dimension += 2;
    return p;
}
```

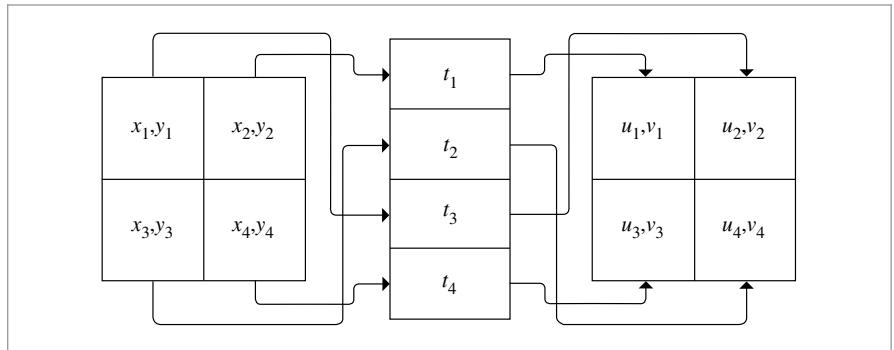
## 7.3 STRATIFIED SAMPLING

The first Sampler implementation that we will introduce subdivides pixel areas into rectangular regions and generates a single sample inside each region. These regions are commonly called *strata*, and this sampler is called the `StratifiedSampler`. The key idea behind stratification is that by subdividing the sampling domain into nonoverlapping regions and taking a single sample from each one, we are less likely to miss important features of the image entirely, since the samples are guaranteed not to all be close together. Put another way, it does us no good if many samples are taken from nearby points in the sample space, since each new sample doesn’t add much new information about the behavior of the image function. From a signal processing viewpoint, we are implicitly defining an overall sampling rate such that the smaller the strata are, the more of them we have, and thus the higher the sampling rate.

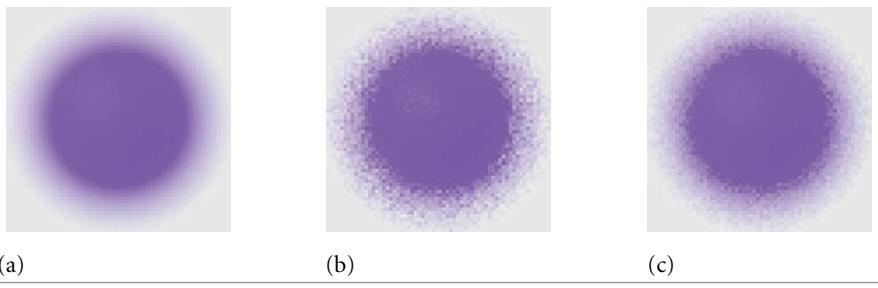
The stratified sampler places each sample at a random point inside each stratum by *jittering* the center point of the stratum by a random amount up to half the stratum’s width and height. The nonuniformity that results from this jittering helps turn aliasing into noise, as discussed in Section 7.1. The sampler also offers an unjittered mode, which gives uniform sampling in the strata; this mode is mostly useful for comparisons between different sampling techniques rather than for rendering high quality images.

Direct application of stratification to high-dimensional sampling quickly leads to an intractable number of samples. For example, if we divided the 5D image, lens, and time sample space into four strata in each dimension, the total number of samples per pixel would be  $4^5 = 1024$ . We could reduce this impact by taking fewer samples in some dimensions (or not stratifying some dimensions, effectively using a single stratum), but

```
Float 1062
GlobalSampler::arrayEndDim
431
GlobalSampler::arrayStartDim
431
GlobalSampler::dimension 430
GlobalSampler::
intervalSampleIndex
430
GlobalSampler::
SampleDimension()
430
Point2f 68
Sampler 421
StratifiedSampler 434
```



**Figure 7.16:** We can generate a good sample pattern that reaps the benefits of stratification without requiring that all of the sampling dimensions be stratified simultaneously. Here, we have split  $(x, y)$  image position, time  $t$ , and  $(u, v)$  lens position into independent strata with four regions each. Each is sampled independently, then a time sample and a lens sample are randomly associated with each image sample. We retain the benefits of stratification in each of the individual dimensions without having to exponentially increase the total number of samples.



**Figure 7.17: Effect of Sampling Patterns in Rendering a Purple Sphere with Depth of Field.**  
 (a) A high-quality reference image of a blurry sphere. (b) An image generated with random sampling in each pixel without stratification. (c) An image generated with the same number of samples, but with the `StratifiedSampler`, which stratified both the image and, more importantly for this image, the lens samples. Stratification makes a substantial improvement for this situation.

we would then lose the benefit of having well-stratified samples in those dimensions. This problem with stratification is known as the *curse of dimensionality*.

We can reap most of the benefits of stratification without paying the price in excessive total sampling by computing lower dimensional stratified patterns for subsets of the domain's dimensions and then randomly associating samples from each set of dimensions. (This process is sometimes called *padding*.) Figure 7.16 shows the basic idea: we might want to take just four samples per pixel but still have the samples be stratified over all dimensions. We independently generate four 2D stratified image samples, four 1D stratified time samples, and four 2D stratified lens samples. Then we randomly associate a time and lens sample value with each image sample. The result is that each pixel has samples that together have good coverage of the sample space. Figure 7.17 shows the improvement in image quality from using stratified lens samples versus using unstratified random samples when rendering depth of field.

Figure 7.18 shows a comparison of a few sampling patterns. The first is a completely random pattern: we generated a number of samples without using the strata at all. The result is terrible; some regions have few samples and other areas have clumps of many samples. The second is a uniform stratified pattern. In the last, the uniform pattern has been jittered, with a random offset added to each sample's location, keeping it inside its cell. This gives a better overall distribution than the purely random pattern while preserving the benefits of stratification, though there are still some clumps of samples and some regions that are undersampled. Figure 7.19 shows images rendered using the `StratifiedSampler` and shows how jittered sample positions turn aliasing artifacts into less objectionable noise.

```
(StratifiedSampler Declarations) ≡
class StratifiedSampler : public PixelSampler {
public:
    (StratifiedSampler Public Methods 434)
private:
    (StratifiedSampler Private Data 434)
};
```

```
(StratifiedSampler Public Methods) ≡ 434
StratifiedSampler(int xPixelSamples, int yPixelSamples,
                  bool jitterSamples, int nSampledDimensions)
: PixelSampler(xPixelSamples * yPixelSamples, nSampledDimensions),
  xPixelSamples(xPixelSamples), yPixelSamples(yPixelSamples),
  jitterSamples(jitterSamples) { }
```

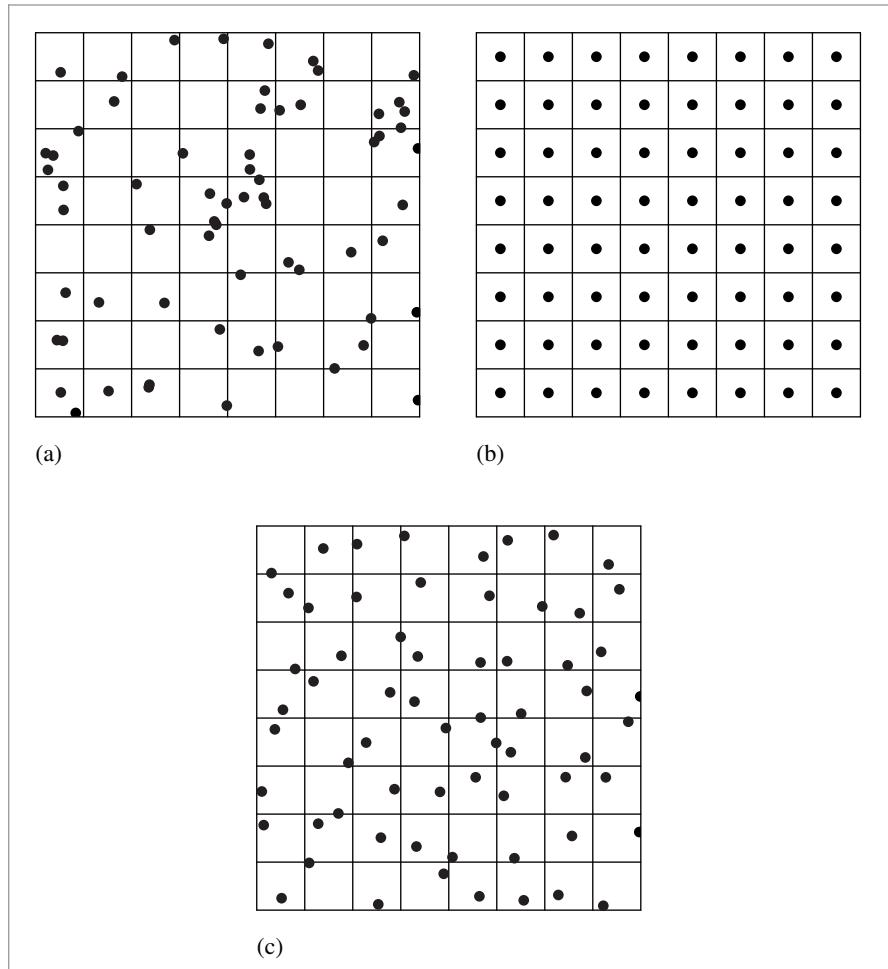
```
(StratifiedSampler Private Data) ≡ 434
const int xPixelSamples, yPixelSamples;
const bool jitterSamples;
```

As a `PixelSampler` subclass, the implementation of `StartPixel()` must both generate 1D and 2D samples for the number of dimensions `nSampledDimensions` passed to the `PixelSampler` constructor as well as samples for the requested arrays.

```
(StratifiedSampler Method Definitions) ≡
void StratifiedSampler::StartPixel(const Point2i &p) {
    (Generate single stratified samples for the pixel 437)
    (Generate arrays of stratified samples for the pixel 440)
    PixelSampler::StartPixel(p);
}
```

After the initial stratified samples are generated, they are randomly shuffled; this is the padding approach described at the start of the section. If this shuffling wasn't done, then the sample dimensions' values would be correlated in a way that would lead to errors in images—for example, both the first 2D sample used to choose the film location as well as the first 2D lens sample would always both be in the lower left stratum adjacent to the origin.

`PixelSampler` [427](#)  
`Point2i` [68](#)  
`Sampler::StartPixel()` [422](#)  
`StratifiedSampler` [434](#)  
`StratifiedSampler::`  
`jitterSamples` [434](#)  
`StratifiedSampler::`  
`xPixelSamples` [434](#)  
`StratifiedSampler::`  
`yPixelSamples` [434](#)



**Figure 7.18: Three 2D Sampling Patterns.** (a) The random pattern is an ineffective pattern, with many clumps of samples that leave large sections of the image poorly sampled. (b) A uniform stratified pattern is better distributed but can exacerbate aliasing artifacts. (c) A stratified jittered pattern turns aliasing from the uniform pattern into high-frequency noise while still maintaining the benefits of stratification.



(a)



(b)



(c)



(d)

**Figure 7.19: Comparison of Image Sampling Methods with a Checkerboard Texture.** This is a difficult image to render well, since the checkerboard's frequency with respect to the pixel spacing tends toward infinity as we approach the horizon. (a) A reference image, rendered with 256 samples per pixel, showing something close to an ideal result. (b) An image rendered with one sample per pixel, with no jittering. Note the jaggy artifacts at the edges of checks in the foreground. Notice also the artifacts in the distance where the checker function goes through many cycles between samples; as expected from the signal processing theory presented earlier, that detail reappears incorrectly as lower frequency aliasing. (c) The result of jittering the image samples, still with just one sample per pixel. The regular aliasing of the second image has been replaced by less objectionable noise artifacts. (d) The result of four jittered samples per pixel is still inferior to the reference image but is substantially better than the previous result.

```

⟨Generate single stratified samples for the pixel⟩ ≡ 434
    for (size_t i = 0; i < samples1D.size(); ++i) {
        StratifiedSample1D(&samples1D[i][0], xPixelSamples * yPixelSamples,
                           rng, jitterSamples);
        Shuffle(&samples1D[i][0], xPixelSamples * yPixelSamples, 1, rng);
    }
    for (size_t i = 0; i < samples2D.size(); ++i) {
        StratifiedSample2D(&samples2D[i][0], xPixelSamples, yPixelSamples,
                           rng, jitterSamples);
        Shuffle(&samples2D[i][0], xPixelSamples * yPixelSamples, 1, rng);
    }
}

```

The 1D and 2D stratified sampling routines are implemented as utility functions. Both loop over the given number of strata in the domain and place a sample point in each one.

```

⟨Sampling Function Definitions⟩ ≡
void StratifiedSample1D(Float *samp, int nSamples, RNG &rng,
                        bool jitter) {
    Float invNSamples = (Float)1 / nSamples;
    for (int i = 0; i < nSamples; ++i) {
        Float delta = jitter ? rng.UniformFloat() : 0.5f;
        samp[i] = std::min((i + delta) * invNSamples, OneMinusEpsilon);
    }
}

```

`StratifiedSample2D()` similarly generates samples in the range  $[0, 1]^2$ .

```

Float 1062
OneMinusEpsilon 417
PixelSampler::samples1D 427
PixelSampler::samples2D 427
Point2f 68
RNG 1065
RNG::UniformFloat() 1066
Shuffle() 438
StratifiedSample1D() 437
StratifiedSample2D() 437
StratifiedSampler::
    jitterSamples
    434
StratifiedSampler::
    xPixelSamples
    434
StratifiedSampler::
    yPixelSamples
    434

```

The `Shuffle()` function randomly permutes an array of count sample values, each of which has `nDimensions` dimensions. (In other words, blocks of values of size `nDimensions` are permuted.)

```
(Sampling Inline Functions) ≡
template <typename T>
void Shuffle(T *samp, int count, int nDimensions, RNG &rng) {
    for (int i = 0; i < count; ++i) {
        int other = i + rng.UniformUInt32(count - i);
        for (int j = 0; j < nDimensions; ++j)
            std::swap(samp[nDimensions * i + j],
                      samp[nDimensions * other + j]);
    }
}
```

Arrays of samples present us with a quandary: for example, if an integrator asks for an array of 64 2D sample values in the sample vector for each sample in a pixel, the sampler has two different goals to try to fulfill:

1. It's desirable that the samples in the array themselves be well distributed in 2D (e.g., by using an  $8 \times 8$  stratified grid). Stratification here will improve the quality of the computed results for each individual sample vector.
2. It's desirable to ensure that each the samples in the array for one image sample isn't too similar to any of the sample values for samples nearby in the image. Rather, we'd like the points to be well distributed with respect to their neighbors, so that over the region around a single pixel, there is good coverage of the entire sample space.

Rather than trying to solve both of these problems simultaneously here, the `StratifiedSampler` only addresses the first one. The other samplers later in this chapter will revisit this issue with more sophisticated techniques and solve both of them simultaneously to various degrees.

A second complication comes from the fact that the caller may have asked for an arbitrary number of samples per image sample, so stratification may not be easily applied. (For example, how do we generate a stratified 2D pattern of seven samples?) We could just generate an  $n \times 1$  or  $1 \times n$  stratified pattern, but this only gives us the benefit of stratification in one dimension and no guarantee of a good pattern in the other dimension. A `StratifiedSampler::RoundSize()` method could round requests up to the next number that's the square of integers, but instead we will use an approach called *Latin hypercube sampling* (LHS), which can generate any number of samples in any number of dimensions with a reasonably good distribution.

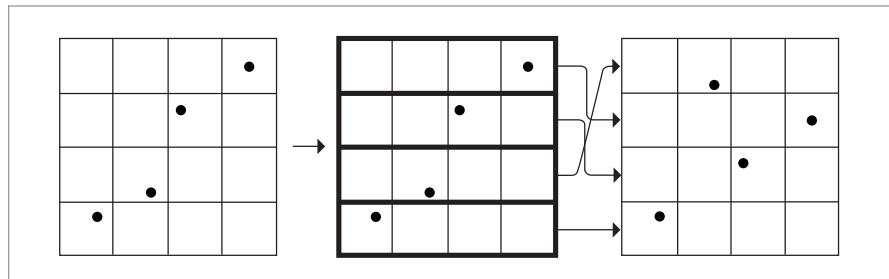
LHS uniformly divides each dimension's axis into  $n$  regions and generates a jittered sample in each of the  $n$  regions along the diagonal, as shown on the left in Figure 7.20. These samples are then randomly shuffled in each dimension, creating a pattern with good distribution. An advantage of LHS is that it minimizes clumping of the samples when they are projected onto any of the axes of the sampling dimensions. This property is in contrast to stratified sampling, where  $2n$  of the  $n \times n$  samples in a 2D pattern may project to essentially the same point on each of the axes. Figure 7.21 shows this worst-case situation for a stratified sampling pattern.

In spite of addressing the clumping problem, LHS isn't necessarily an improvement to stratified sampling; it's easy to construct cases where the sample positions are essentially

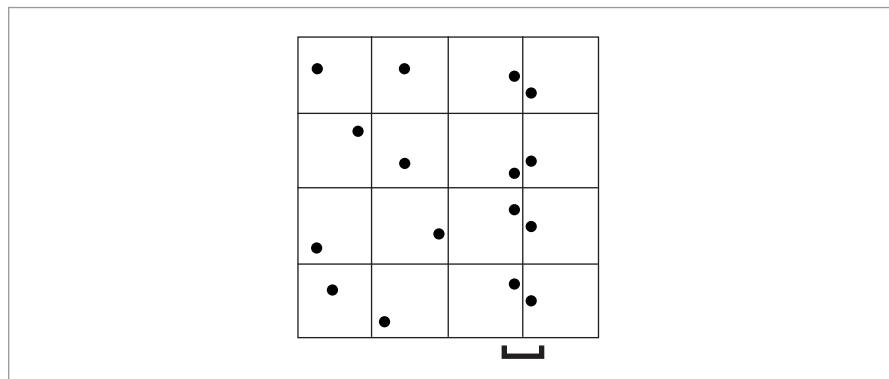
RNG 1065

RNG::UniformUInt32() 1066

StratifiedSampler 434



**Figure 7.20:** Latin hypercube sampling (sometimes called  $n$ -rooks sampling) chooses samples such that only a single sample is present in each row and each column of a grid. This can be done by generating random samples in the cells along the diagonal and then randomly permuting their coordinates. One advantage of LHS is that it can generate any number of samples with a good distribution, not just  $m \times n$  samples, as with stratified patterns.



**Figure 7.21: A Worst-Case Situation for Stratified Sampling.** In an  $n \times n$  2D pattern, up to  $2n$  of the points may project to essentially the same point on one of the axes. When “unlucky” patterns like this are generated, the quality of the results computed with them usually suffers.

colinear and large areas of the sampling domain have no samples near them (e.g., when the permutation of the original samples is the identity, leaving them all where they started). In particular, as  $n$  increases, Latin hypercube patterns are less and less effective compared to stratified patterns.<sup>7</sup>

The general-purpose `LatinHypercube()` function generates an arbitrary number of LHS samples in an arbitrary dimension. The number of elements in the `samples` array should thus be `nSamples*nDim`.

`LatinHypercube()` 440

<sup>7</sup> We will revisit this issue in the following sections, where we will discuss sample patterns that are simultaneously stratified and distributed in a Latin hypercube pattern.

```
(Sampling Function Definitions) +≡
void LatinHypercube(Float *samples, int nSamples, int nDim, RNG &rng) {
    {Generate LHS samples along diagonal 440}
    {Permute LHS samples in each dimension 440}
}
```

```
(Generate LHS samples along diagonal) ≡ 440
Float invNSamples = (Float)1 / nSamples;
for (int i = 0; i < nSamples; ++i)
    for (int j = 0; j < nDim; ++j) {
        Float sj = (i + (rng.UniformFloat())) * invNSamples;
        samples[nDim * i + j] = std::min(sj, OneMinusEpsilon);
    }
```

To do the permutation, this function loops over the samples, randomly permuting the sample points in one dimension at a time. Note that this is a different permutation than the earlier `Shuffle()` routine: that routine does one permutation, keeping all `nDim` sample points in each sample together, while here `nDim` separate permutations of a single dimension at a time are done (Figure 7.22).<sup>8</sup>

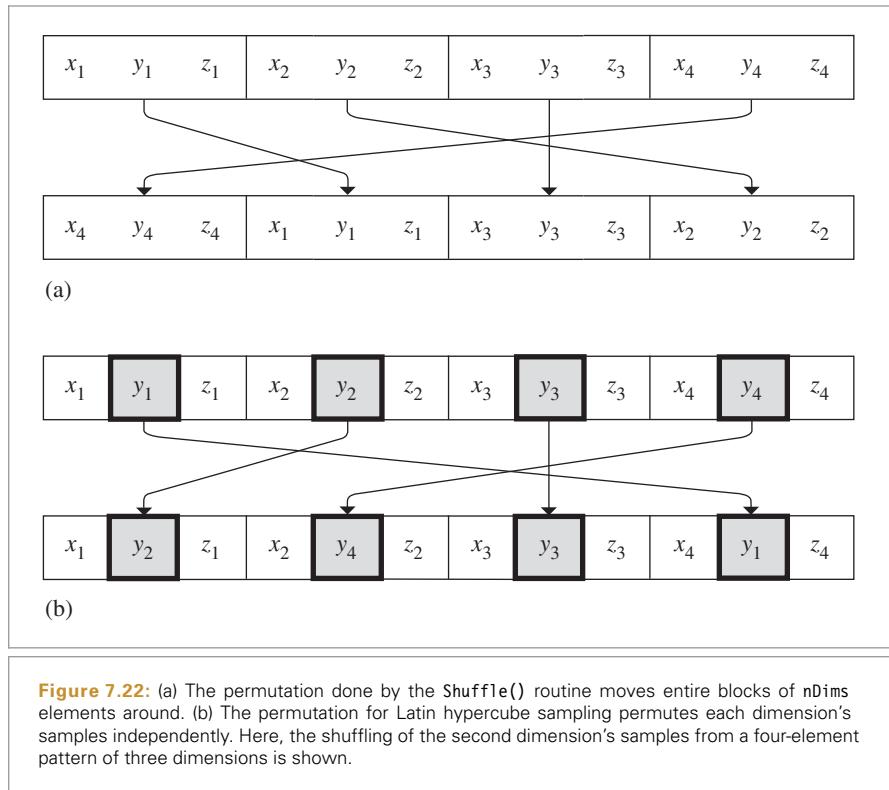
```
(Permute LHS samples in each dimension) ≡ 440
for (int i = 0; i < nDim; ++i) {
    for (int j = 0; j < nSamples; ++j) {
        int other = j + rng.UniformUInt32(nSamples - j);
        std::swap(samples[nDim * j + i], samples[nDim * other + i]);
    }
}
```

Given the `LatinHypercube()` function, we can now write the code to compute sample arrays for the current pixel. 1D samples are stratified and then randomly shuffled, while 2D samples are generated using Latin hypercube sampling.

```
(Generate arrays of stratified samples for the pixel) ≡ 434
for (size_t i = 0; i < samples1DArraySizes.size(); ++i)
    for (int64_t j = 0; j < samplesPerPixel; ++j) {
        int count = samples1DArraySizes[i];
        StratifiedSample1D(&sampleArray1D[i][j * count], count, rng,
                           jitterSamples);
        Shuffle(&sampleArray1D[i][j * count], count, 1, rng);
    }
for (size_t i = 0; i < samples2DArraySizes.size(); ++i)
    for (int64_t j = 0; j < samplesPerPixel; ++j) {
        int count = samples2DArraySizes[i];
        LatinHypercube(&sampleArray2D[i][j * count].x, count, 2, rng);
    }
```

Float 1062
LatinHypercube() 440
OneMinusEpsilon 417
RNG 1065
RNG::UniformFloat() 1066
RNG::UniformUInt32() 1066
Sampler::sampleArray1D 426
Sampler::sampleArray2D 426
Sampler::samples1DArraySizes 426
Sampler::samples2DArraySizes 426
Sampler::samplesPerPixel 422
Shuffle() 438
StratifiedSample1D() 437

<sup>8</sup> While it's not necessary to permute the first dimension of the LHS pattern, the implementation here does so anyway, since making the elements of the first dimension be randomly ordered means that LHS patterns can be used in conjunction with sampling patterns from other sources without danger of correlation between their sample points.



Starting with the scene in Figure 7.23, Figure 7.24 shows the improvement from good samples for the `DirectLightingIntegrator`. Image (a) was computed with 1 image sample per pixel, each with 16 shadow samples, and image (b) was computed with 16 image samples per pixel, each with 1 shadow sample. Because the `StratifiedSampler` could generate a good LHS pattern for the first case, the quality of the shadow is much better, even with the same total number of shadow samples taken.

## \* 7.4 THE HALTON SAMPLER

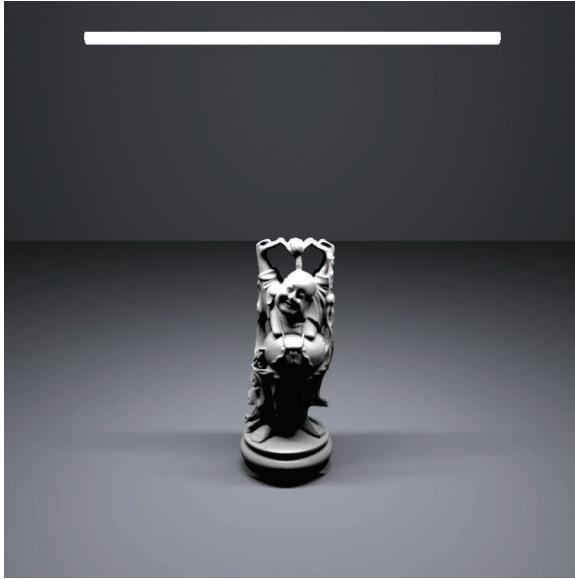
The underlying goal of the `StratifiedSampler` is to generate a well-distributed but non-uniform set of sample points, with no two sample points too close together and no excessively large regions of the sample space that have no samples. As Figure 7.18 showed, a jittered pattern does this much better than a random pattern does, although its quality can suffer when samples in adjacent strata happen to be close to the shared boundary of their two strata.

This section introduces the `HaltonSampler`, which is based on algorithms that directly generate low-discrepancy point sets. Unlike the points generated by the `StratifiedSampler`, the `HaltonSampler` not only generates points that are guaranteed to not clump too closely together, but it also generates points that are simultaneously well distributed

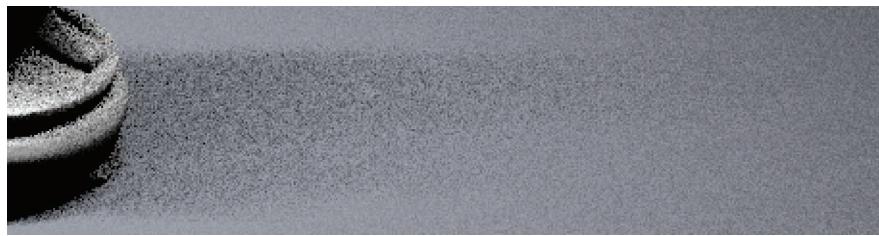
`DirectLightingIntegrator` 851

`Shuffle()` 438

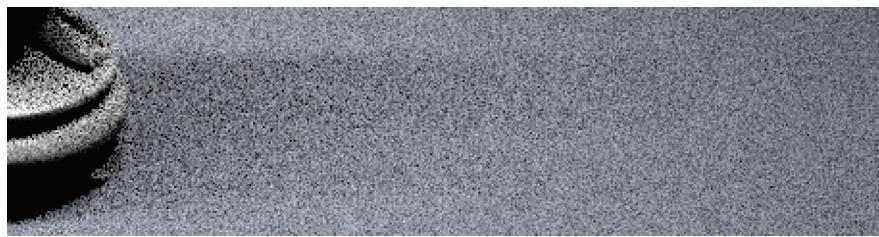
`StratifiedSampler` 434



**Figure 7.23: Area Light Sampling Example Scene.**



(a)



(b)

**Figure 7.24: Sampling an Area Light with Samples from the Stratified Sampler.** (a) shows the result of using 1 image sample per pixel and 16 shadow samples, and (b) shows the result of 16 image samples, each with just 1 shadow sample. The total number of shadow samples is the same in both cases, but because the version with 16 shadow samples per image sample is able to use an LHS pattern, all of the shadow samples in a pixel's area are well distributed, while in the second image the implementation here has no way to prevent them from being poorly distributed. The difference is striking.

over all of the dimensions of the sample vector—not just one or two dimensions at a time, as the `StratifiedSampler` did.

### 7.4.1 HAMMERSLEY AND HALTON SEQUENCES

The Halton and Hammersley sequences are two closely related low-discrepancy point sets. Both are based on a construction called the *radical inverse*, which is based on the fact that a positive integer value  $a$  can be expressed in a base  $b$  with a sequence of digits  $d_m(a) \dots d_2(a)d_1(a)$  uniquely determined by

$$a = \sum_{i=1}^m d_i(a)b^{i-1}, \quad [7.6]$$

where all digits  $d_i(a)$  are between 0 and  $b - 1$ .

The radical inverse function  $\Phi_b$  in base  $b$  converts a nonnegative integer  $a$  to a fractional value in  $[0, 1)$  by reflecting these digits about the radix point:

$$\Phi_b(a) = 0.d_1(a)d_2(a) \dots d_m(a). \quad [7.7]$$

Thus, the contribution of the digit  $d_i(a)$  to the radical inverse is  $d_i(a)/b^i$ .

One of the simplest low-discrepancy sequences is the *van der Corput sequence*, which is a 1D sequence given by the radical inverse function in base 2:

$$x_a = \Phi_2(a).$$

Table 7.3 shows the first few values of the van der Corput sequence. Notice how it recursively splits the intervals of the 1D line in half, generating a sample point at the center of each interval. The discrepancy of this sequence is

$$D_N^*(P) = O\left(\frac{\log N}{N}\right),$$

which matches the best discrepancy that has been attained for infinite sequences in

**Table 7.3:** The radical inverse  $\Phi_2(a)$  of the first few non-negative integers, computed in base 2. Notice how successive values of  $\Phi_2(a)$  are not close to any of the previous values of  $\Phi_2(a)$ . As more and more values of the sequence are generated, samples are necessarily closer to previous samples, although with a minimum distance that is guaranteed to be reasonably good.

$a$	Base 2	$\Phi_2(a)$
0	0	0
1	1	$0.1 = 1/2$
2	10	$0.01 = 1/4$
3	11	$0.11 = 3/4$
4	100	$0.001 = 1/8$
5	101	$0.101 = 5/8$
:		

$n$  dimensions,

$$D_N^*(P) = O\left(\frac{(\log N)^n}{N}\right).$$

To generate points in an  $n$ -dimensional Halton sequence, we use the radical inverse base  $b_i$ , with a different base for each dimension of the pattern. The bases used must all be relatively prime to each other, so a natural choice is to use the first  $n$  prime numbers ( $p_1, \dots, p_n$ ):

$$x_a = (\Phi_2(a), \Phi_3(a), \Phi_5(a), \dots, \Phi_{p_n}(a)).$$

One of the most useful characteristics of the Halton sequence is that it can be used even if the total number of samples needed isn't known in advance; all prefixes of the sequence are well distributed, so as additional samples are added to the sequence low discrepancy will be maintained. (However, its distribution is best when the total number of samples is the product of powers of the bases  $\Pi(p_i)^{k_i}$  for exponents  $k_i$ .)

The discrepancy of an  $n$ -dimensional Halton sequence is

$$D_N^*(x_a) = O\left(\frac{(\log N)^n}{N}\right),$$

which is asymptotically optimal.

If the number of samples  $N$  is fixed, the Hammersley point set can be used, giving slightly lower discrepancy. Hammersley point sets are defined by

$$x_a = \left(\frac{a}{N}, \Phi_{b_1}(a), \Phi_{b_2}(a), \dots, \Phi_{b_n}(a)\right),$$

where  $N$  is the total number of samples to be taken and as before all of the bases  $b_i$  are relatively prime. Figure 7.25(a) shows a plot of the first 216 points of the 2D Halton sequence. Figure 7.25(b) shows the first 256 points of the Hammersley sequence.

The function `RadicalInverse()` computes the radical inverse for a given number  $a$  using the `baseIndex` prime number as the base. The function is implemented using an enormous `switch` statement, where `baseIndex` is mapped to the appropriate prime number and then a separate `RadicalInverseSpecialized()` template function actually computes the radical inverse. (The reason for the curious `switch`-based structure will be explained in a few pages.)

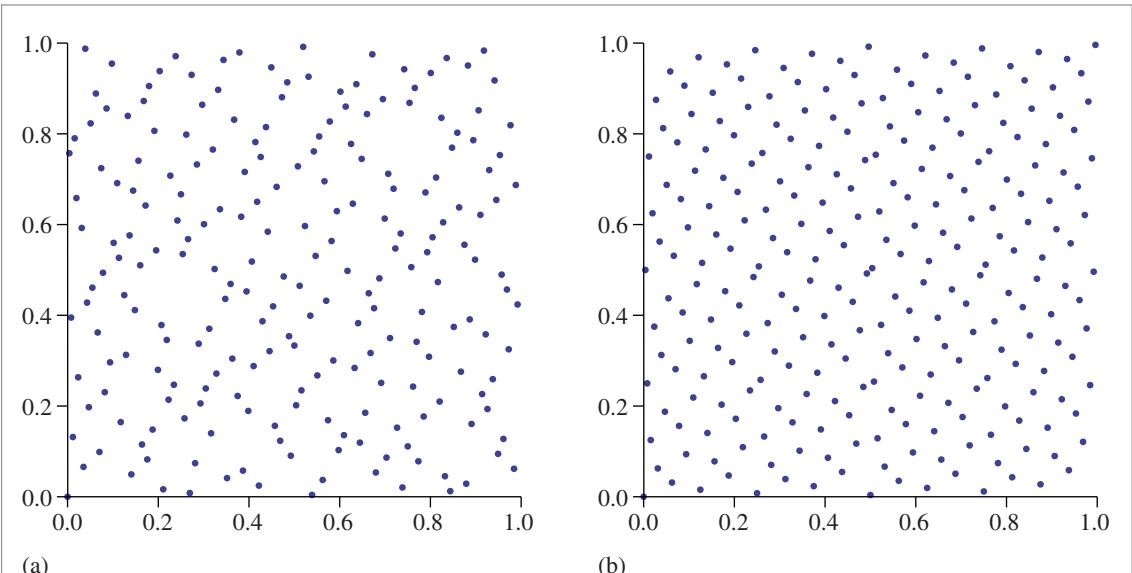
*(Low Discrepancy Function Definitions)* ≡

```
Float RadicalInverse(int baseIndex, uint64_t a) {
    switch (baseIndex) {
        case 0:
            (Compute base-2 radical inverse 446)
        case 1: return RadicalInverseSpecialized<3>(a);
        case 2: return RadicalInverseSpecialized<5>(a);
        case 3: return RadicalInverseSpecialized<7>(a);
        (Remainder of cases for RadicalInverse())
    }
}
```

Float 1062

RadicalInverse() 444

RadicalInverseSpecialized()
447



**Figure 7.25: The First Points of Two Low-Discrepancy Sequences in 2D.** (a) Halton (216 points), (b) Hammersley (256 points).

For the base-2 radical inverse, we can take advantage of the fact that numbers in digital computers are already represented in base 2 to compute the radical inverse more efficiently. For a 64-bit value  $a$ , we have from Equation (7.6)

$$a = \sum_{i=1}^{64} d_i(a) 2^{i-1}.$$

First consider the result of reversing the bits of  $a$ , still considering it as an integer value, which gives

$$\sum_{i=1}^{64} d_i(a) 2^{64-i}.$$

If we then divide this value by  $2^{64}$ , we have

$$\sum_{i=1}^{64} d_i(a) 2^{-i},$$

which is  $\Phi_2(a)$ . Thus, the base-2 radical inverse can equivalently be computed with a bit reverse and a power-of-two division.

The bits of an integer quantity can be efficiently reversed with a series of logical bit operations. The first line of the `ReverseBits32()` function, which reverses the bits of a 32-bit integer, swaps the lower 16 bits with the upper 16 bits of the value. The next line simultaneously swaps the first 8 bits of the result with the second 8 bits and the third 8 bits with the fourth. This process continues until the last line, which swaps adjacent bits.

To understand this code, it's helpful to write out the binary values of the various hexadecimal constants. For example, `0xff00ff00` is `111111100000000111111100000000` in binary; it's easy to see that a bitwise OR with this value masks off the first and third 8-bit quantities.

*(Low Discrepancy Inline Functions) ≡*

```
inline uint32_t ReverseBits32(uint32_t n) {
    n = (n << 16) | (n >> 16);
    n = ((n & 0x00ff00ff) << 8) | ((n & 0xff00ff00) >> 8);
    n = ((n & 0x0f0f0f0f) << 4) | ((n & 0xf0f0f0f0) >> 4);
    n = ((n & 0x33333333) << 2) | ((n & 0xcccccccc) >> 2);
    n = ((n & 0x55555555) << 1) | ((n & 0aaaaaaaa) >> 1);
    return n;
}
```

The bits of a 64-bit value can then be reversed by reversing the two 32-bit components individually and then interchanging them.

*(Low Discrepancy Inline Functions) +≡*

```
inline uint64_t ReverseBits64(uint64_t n) {
    uint64_t n0 = ReverseBits32((uint32_t)n);
    uint64_t n1 = ReverseBits32((uint32_t)(n >> 32));
    return (n0 << 32) | n1;
}
```

To compute the base-2 radical inverse, then, we reverse the bits and multiply by  $1/2^{64}$ , where the hexadecimal floating-point constant `0x1p-64` is used for the value  $2^{-64}$ . As explained in Section 3.9.1, implementing a power-of-two division via the corresponding power-of-two multiplication gives the same result with IEEE floating point. (And floating-point multiplication is generally more efficient than floating-point division.)

*(Compute base-2 radical inverse) ≡*

444

```
return ReverseBits64(a) * 0x1p-64;
```

For other bases, the `RadicalInverseSpecialized()` template function computes the radical inverse by computing the digits  $d_i$  starting with  $d_1$  and computing a series  $v_i$  where  $v_1 = d_1$ ,  $v_2 = bd_1 + d_2$  such that

$$v_n = b^{n-1}d_1 + b^{n-2}d_2 + \cdots + d_n.$$

(For example, in base 10, it would convert the value 1234 to 4321.) This value can be found entirely using integer arithmetic, without accumulating any round-off error.

The final value of the radical inverse is then found by converting to floating-point and multiplying by  $1/b^n$ , where  $n$  is the number of digits in the value, to get the value in Equation (7.7). The term for this multiplication is built up in `invBaseN` as the digits are processed.

`ReverseBits32()` 446

`ReverseBits64()` 446

```

<Low Discrepancy Static Functions> ≡
    template <int base>
    static Float RadicalInverseSpecialized(uint64_t a) {
        const Float invBase = (Float)1 / (Float)base;
        uint64_t reversedDigits = 0;
        Float invBaseN = 1;
        while (a) {
            uint64_t next = a / base;
            uint64_t digit = a - next * base;
            reversedDigits = reversedDigits * base + digit;
            invBaseN *= invBase;
            a = next;
        }
        return std::min(reversedDigits * invBaseN, OneMinusEpsilon);
    }
}

```

A natural question to ask would be why a template function parameterized on the base is used here (rather than, say, a regular function call that took the base as a parameter, which would avoid the generation of a separate code path for each base). The motivation is that integer division is shockingly slow on modern CPUs, and much more efficient approaches are possible for division by a compile-time constant.

For example, integer division of a 32-bit value by 3 can be computed exactly by multiplying this value by 2863311531 to get a 64-bit intermediate and then shifting the result right by 33 bits; these are both fairly efficient operations. (A similar approach can be used for dividing 64-bit values by 3, but the magic constant is much larger; see Warren (2006) for more about these techniques.) Thus, using a template function here allows the compiler to see that the division to compute the value of `next` in the `while` loop is actually a division by a constant and gives it a chance to apply this optimization. The code with this optimization runs 5.9 times faster on a 2015-era laptop than an implementation based on integer division instructions.

Another optimization is that we avoid computing a running sum over reversed digits multiplied by the reciprocal base; instead, this multiplication is postponed all the way until the end when the loop terminates. The main issue here is that floating-point and integer units on current processors operate fairly independently from each other. Referencing an integer variable within a floating computation in a tight loop would introduce pipeline bubbles related to the amount of time that is needed to convert and move the values from one unit to the other.

It will be useful to be able to compute the inverse of the radical inverse function; the `InverseRadicalInverse()` function takes the reversed integer digits in some base, corresponding to `value` in the `RadicalInverseSpecialized()` template function immediately before being multiplied by the  $1/b^n$  factor to convert to a floating-point value in  $[0, 1]$ . Note that in order to be able to compute the inverse correctly, the total number of digits in the original value must be known: for example, both 1234 and 123400 are converted to 4321 after the integer-only part of the radical inverse algorithm; trailing zeros become leading zeros, which are lost.

`Float` 1062

`OneMinusEpsilon` 417

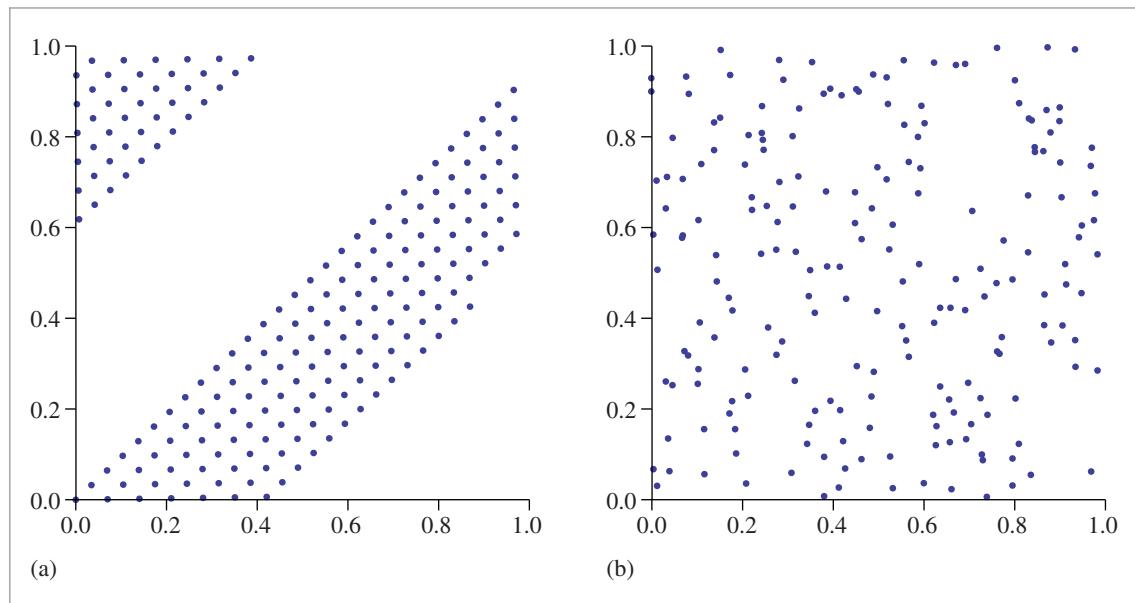
`RadicalInverseSpecialized()`  
447

```
(Low Discrepancy Inline Functions) +≡
template <int base> inline uint64_t
InverseRadicalInverse(uint64_t inverse, int nDigits) {
    uint64_t index = 0;
    for (int i = 0; i < nDigits; ++i) {
        uint64_t digit = inverse % base;
        inverse /= base;
        index = index * base + digit;
    }
    return index;
}
```

The Hammersley and Halton sequences have the shortcoming that as the base  $b$  increases, sample values can exhibit surprisingly regular patterns. This issue can be addressed with *scrambled* Halton and Hammersley sequences, where a permutation is applied to the digits when computing the radical inverse:

$$\Psi_b(a) = 0.p(d_1(a))p(d_2(a)) \dots p(d_m(a)), \quad [7.8]$$

where  $p$  is a permutation of the digits  $(0, 1, \dots, b - 1)$ . Note that the same permutation is used for each digit, and the same permutation is used for generating all of the sample points in a given base  $b$ . Figure 7.26 shows the effect of scrambling with the Halton sequence. In the following, we will use random permutations, though specific construc-



**Figure 7.26: Plot of Halton Sample Values with and without Scrambling.** (a) In higher dimensions of the sample vector, projections of sample values start to exhibit regular structure. Here, points from the dimensions  $(\Phi_{29}(a), \Phi_{31}(a))$  are shown. (b) Scrambled sequences, Equation (7.8), break up this structure by randomly permuting the digits of sample indices.

tions of permutations can give slightly better results; see the “Further Reading” section for more details.

The `ComputeRadicalInversePermutations()` function computes these random permutation tables. It initializes a single contiguous array for all of the permutations, where the first two values are a permutation of the integers zero and one for  $b = 2$ , the next three values are a permutation of 0, 1, 2 for  $b = 3$ , and so forth for successive prime bases. At entry to the `for` loop below, `p` points to the start of the permutation array to initialize for the current prime base.

```
(Low Discrepancy Function Definitions) +≡
std::vector<uint16_t> ComputeRadicalInversePermutations(RNG &rng) {
    std::vector<uint16_t> perms;
(Allocate space in perms for radical inverse permutations 449)
    uint16_t *p = &perms[0];
    for (int i = 0; i < PrimeTableSize; ++i) {
        (Generate random permutation for ith prime base 449)
        p += Primes[i];
    }
    return perms;
}
```

The total size of the permutation array is given by the sum of the prime numbers up to the end of a precomputed table of prime numbers.

```
(Allocate space in perms for radical inverse permutations) ≡ 449
int permArraySize = 0;
for (int i = 0; i < PrimeTableSize; ++i)
    permArraySize += Primes[i];
perms.resize(permArraySize);
```

```
(Low Discrepancy Declarations) ≡
static constexpr int PrimeTableSize = 1000;
extern const int Primes[PrimeTableSize];
```

```
(Low Discrepancy Data Definitions) ≡
const int Primes[PrimeTableSize] = {
    2, 3, 5, 7, 11,
(Subsequent prime numbers)
};
```

Generating each permutation is easy: we just initialize `p` to the identity permutation for the current prime length and then randomly shuffle its values.

```
(Generate random permutation for ith prime base) ≡ 449
for (int j = 0; j < Primes[i]; ++j)
    p[j] = j;
Shuffle(p, Primes[i], 1, rng);
```

The `ScrambledRadicalInverse()` function is essentially the same as `RadicalInverse()` except that it puts each digit through the permutation table for the given base. See

Primes 449

PrimeTableSize 449

RNG 1065

Shuffle() 438

Exercise 7.1 for discussion of a more efficient implementation for the base-2 case, following `RadicalInverse()`.

*(Low Discrepancy Function Definitions) +≡*

```
Float ScrambledRadicalInverse(int baseIndex, uint64_t a,
    const uint16_t *perm) {
    switch (baseIndex) {
        case 0: return ScrambledRadicalInverseSpecialized<2>(perm, a);
        case 1: return ScrambledRadicalInverseSpecialized<3>(perm, a);
        case 2: return ScrambledRadicalInverseSpecialized<5>(perm, a);
        case 3: return ScrambledRadicalInverseSpecialized<7>(perm, a);
        <Remainder of cases for ScrambledRadicalInverse()>
    }
}
```

The implementation below also accounts for a special case that can arise when `perm` maps the digit 0 to a nonzero value. In this case, the iteration stops prematurely once `a` reaches 0, incorrectly missing an infinitely long suffix of digits with value `perm[0]`. Fortunately, this is a geometric series with a simple analytic solution whose value is added in the last line.

*(Low Discrepancy Static Functions) +≡*

```
template <int base>
static Float ScrambledRadicalInverseSpecialized(const uint16_t *perm,
    uint64_t a) {
    const Float invBase = (Float)1 / (Float)base;
    uint64_t reversedDigits = 0;
    Float invBaseN = 1;
    while (a) {
        uint64_t next = a / base;
        uint64_t digit = a - next * base;
        reversedDigits = reversedDigits * base + perm[digit];
        invBaseN *= invBase;
        a = next;
    }
    return std::min(invBaseN * (reversedDigits +
        invBase * perm[0] / (1 - invBase)), OneMinusEpsilon);
}
```

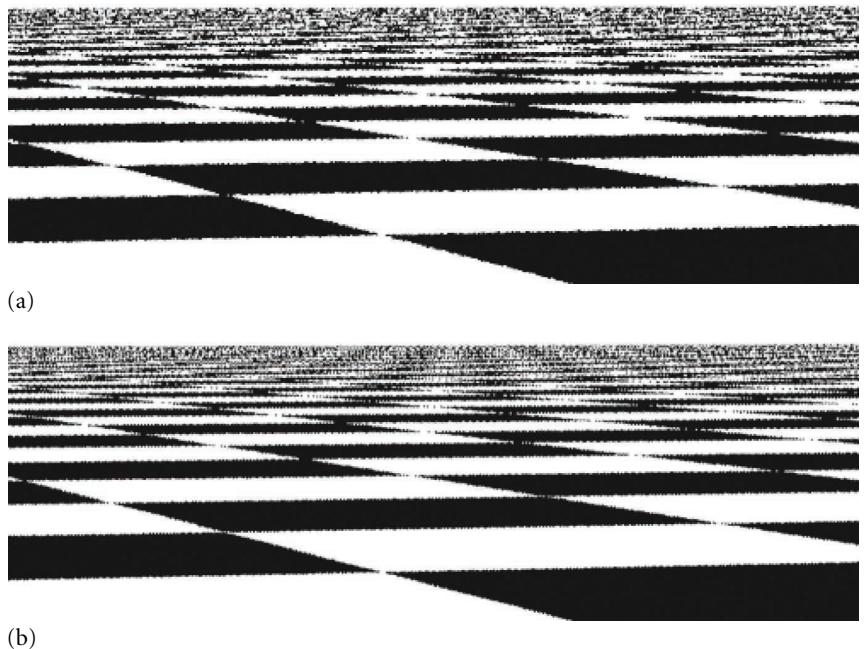
## 7.4.2 HALTON SAMPLER IMPLEMENTATION

The `HaltonSampler` generates sample vectors using the Halton sequence. Unlike the `StratifiedSampler`, it is fully deterministic; it uses no pseudo-random numbers in its operation. However, Halton samples can be lead to aliasing if the image isn't sufficiently well sampled. Figure 7.27 compares the results of sampling a checkerboard texture using a Halton-based sampler to using the stratified sampler from the previous section. Note the unpleasant pattern along edges in the foreground and toward the horizon.

*(HaltonSampler Declarations) ≡*

```
class HaltonSampler : public GlobalSampler {
public:
    <HaltonSampler Public Methods>
```

Float 1062  
 GlobalSampler 429  
 OneMinusEpsilon 417  
 RadicalInverse() 444  
 StratifiedSampler 434



**Figure 7.27: Comparison of the Stratified Sampler to a Low-Discrepancy Sampler Based on Halton Points on the Image Plane.** (a) The jittered stratified sampler with a single sample per pixel and (b) the HaltonSampler sampler with a single sample per pixel. Note that although the Halton pattern is able to reproduce the checker pattern farther toward the horizon than the stratified pattern, there is a regular structure to the error in the low-discrepancy pattern that is visually distracting; it doesn't turn aliasing into less objectionable noise as well as the jittered approach.

```

private:
    <HaltonSampler Private Data 452>
    <HaltonSampler Private Methods 452>
};

<HaltonSampler Method Definitions> ==
HaltonSampler::HaltonSampler(int samplesPerPixel,
    const Bounds2i &sampleBounds)
: GlobalSampler(samplesPerPixel) {
    <Generate random digit permutations for Halton sampler 452>
    <Find radical inverse base scales and exponents that cover sampling area 452>
    <Compute stride in samples for visiting each pixel area 453>
    <Compute multiplicative inverses for baseScales>
}

```

Bounds2i 76  
GlobalSampler 429  
HaltonSampler 450

The permutation tables for the scrambled radical inverses are shared across all Halton Sampler instances and are computed the first time the constructor runs. For pbrt's requirements, this approach is fine: the current implementation only uses different sampler instances for different tiles of the image, where we'd like to always use the same permutations anyway. For other uses, it could be worthwhile to have more control over when different permutations are used.

```
(Generate random digit permutations for Halton sampler) ≡ 451
  if (radicalInversePermutations.size() == 0) {
    RNG rng;
    radicalInversePermutations = ComputeRadicalInversePermutations(rng);
  }

(HaltonSampler Private Data) ≡ 450
  static std::vector<uint16_t> radicalInversePermutations;
```

The utility method `PermutationForDimension()` returns a pointer to the start of the permutation array for the given dimension.

```
(HaltonSampler Private Methods) ≡ 450
  const uint16_t *PermutationForDimension(int dim) const {
    if (dim >= PrimeTableSize)
      Severe("HaltonSampler can only sample %d dimensions.", PrimeTableSize);
    return &radicalInversePermutations[PrimeSums[dim]];
  }
```

To be able to quickly find the offset for a given dimension, it's helpful to have the sums of the prime numbers preceding each prime.

```
(Low Discrepancy Data Definitions) +≡
  const int PrimeSums[PrimeTableSize] = {
    0, 2, 5, 10, 17,
    (Subsequent prime sums)
  };
```

To map the first two dimensions of samples from  $[0, 1)^2$  to pixel coordinates, the `HaltonSampler` finds the smallest scale factor  $(2^j, 3^k)$  that is larger than the lower of either the image resolution or `kMaxResolution` in each dimension. (We will see shortly how this specific choice of scales makes it easy to see which pixel a sample lands in.) After scaling, any samples outside the image extent will be simply ignored.

For images with resolution greater than `kMaxResolution` in one or both dimensions, a tile of Halton points is repeated across the image. This resolution limit helps maintain sufficient floating-point precision in the computed sample values.

```
(Find radical inverse base scales and exponents that cover sampling area) ≡ 451
  Vector2i res = sampleBounds.pMax - sampleBounds.pMin;
  for (int i = 0; i < 2; ++i) {
    int base = (i == 0) ? 2 : 3;
    int scale = 1, exp = 0;
    while (scale < std::min(res[i], kMaxResolution)) {
      scale *= base;
      ++exp;
    }
    baseScales[i] = scale;
    baseExponents[i] = exp;
  }
```

```
ComputeRadicalInverse
  Permutations() 449
HaltonSampler::baseExponents 453
HaltonSampler::baseScales 453
HaltonSampler::kMaxResolution 453
HaltonSampler::
  radicalInversePermutations 452
  PrimeSums 452
  PrimeTableSize 449
  RNG 1065
  Severe() 1068
  Vector2i 60
```

For each dimension, `baseScales` holds the scale factor,  $2^j$  or  $3^k$ , and `baseExponents` holds the exponents  $j$  and  $k$ .

*(HaltonSampler Private Data)*  $\doteqdot$   
`Point2i baseScales, baseExponents;`

450

*(HaltonSampler Local Constants)*  $\equiv$   
`static constexpr int kMaxResolution = 128;`

To see why the `HaltonSampler` uses this scheme to map samples to pixel coordinates, consider the effect of scaling a value computed with the radical inverse base  $b$  by a factor  $b^n$ . If the digits of  $a$  expressed in base  $b$  are  $d_i(a)$ , then recall that the radical inverse is the value  $0.d_1(a)d_2(a)\dots$ , base  $b$ . If we multiply this value by  $b^2$ , for example, we have  $d_1(a)d_2(a).d_3(a)\dots$ ; the first two digits have moved to the left of the radix point, and the fractional component of the value starts with  $d_3(a)$ .

This operation—scaling by  $b^n$ —forms the core of being able to determine which sample indices land in which pixels. Considering the first two digits in the above example, we can see that the integer component of the scaled value ranges from 0 to  $b^2 - 1$  and that as  $a$  increases, its last two digits in base  $b$  take on any particular value once in every  $b^2$  values in this range.

Given a value  $x$ ,  $0 \leq x \leq b^2 - 1$ , we can find the first value  $a$  that gives the value  $x$  in the integer components. By definition, the digits of  $x$  in base  $b$  are  $d_2(x)d_1(x)$ . Thus, if  $d_1(a) = d_2(x)$  and  $d_2(a) = d_1(x)$ , then the scaled value of  $a$ 's radical inverse will have an integer component equal to  $x$ .

Because the bases  $b = 2$  and  $b = 3$  used in the `HaltonSampler` for pixel samples are relatively prime, it follows that if the sample values are scaled by some  $(2^j, 3^k)$ , then any particular pixel in the range  $(0, 0) \rightarrow (2^j - 1, 3^k - 1)$  will be visited once every  $2^j 3^k$  samples. This product is stored in `sampleStride`.

*(Compute stride in samples for visiting each pixel area)*  $\equiv$   
`sampleStride = baseScales[0] * baseScales[1];`

451

*(HaltonSampler Private Data)*  $\doteqdot$   
`int sampleStride;`

450

The sample index for the first Halton sample that lands in `currentPixel` is stored in `offsetForCurrentPixel`. After this offset has first been computed for the first sample in the current pixel, subsequent samples in the pixel are found at increments of `sampleStride` samples in the Halton sequence.

*(HaltonSampler Method Definitions)*  $\doteqdot$   
`int64_t HaltonSampler::GetIndexForSample(int64_t sampleNum) const {`  
 `if (currentPixel != pixelForOffset) {`  
 `(Compute Halton sample offset for currentPixel)`  
 `pixelForOffset = currentPixel;`  
 `}`  
 `return offsetForCurrentPixel + sampleNum * sampleStride;`  
`}`

```
HaltonSampler::baseScales  
453
HaltonSampler::  
    offsetForCurrentPixel  
454
HaltonSampler::pixelForOffset  
454
HaltonSampler::sampleStride  
453
Point2i 68
Sampler::currentPixel 425
```

```
(HaltonSampler Private Data) +≡
    mutable Point2i pixelForOffset = Point2i(std::numeric_limits<int>::max(),
                                              std::numeric_limits<int>::max());
    mutable int64_t offsetForCurrentPixel;
```

Computing the index of the first sample in a given pixel  $(x, y)$  where the samples have been scaled by  $(2^j, 3^k)$  involves computing the inverse radical inverse of the last  $j$  digits of  $x$  in base 2, which we'll denote by  $x_r$ , and of the last  $k$  digits of  $y$  in base 3,  $y_r$ . This gives us a system of equations

$$x_r \equiv (i \bmod 2^j)$$

$$y_r \equiv (i \bmod 3^k),$$

where the index  $i$  that satisfies these equations is the index of a sample that lies within the given pixel, after scaling. We don't include the code that solves for  $i$  (*Compute Halton sample offset for currentPixel*) here in the book; see Grünschloß et al. (2012) for details of the algorithm used to find  $i$ .

The computation of sample offsets doesn't account for random digit permutations, so those aren't included in the sample values computed here. Also, because the low `baseExponents[i]` digits of the first two dimensions are used to select which pixel is sampled, these digits must be discarded before computing the radical inverse for the first two dimensions of the sample vector, since the `SampleDimension()` method is supposed to return the fractional offset within the pixel being sampled. Higher dimensions are just sampled directly, including the random permutations.

```
(HaltonSampler Method Definitions) +≡
    Float HaltonSampler::SampleDimension(int64_t index, int dim) const {
        if (dim == 0)
            return RadicalInverse(dim, index >> baseExponents[0]);
        else if (dim == 1)
            return RadicalInverse(dim, index / baseScales[1]);
        else
            return ScrambledRadicalInverse(dim, index,
                                           PermutationForDimension(dim));
    }
```

```
Float 1062
HaltonSampler::baseExponents 453
HaltonSampler::baseScales 453
HaltonSampler::PermutationForDimension() 452
Point2i 68
RadicalInverse() 444
ScrambledRadicalInverse() 450
StratifiedSampler 434
```

## ★ 7.5 (0, 2)-SEQUENCE SAMPLER

Another approach for generating high-quality samples takes advantage of a remarkable property of certain low-discrepancy sequences that allows us to satisfy two desirable properties of samples (only one of which was satisfied with the `StratifiedSampler`): they generate sample vectors for a pixel's worth of image samples such that the sample values for each pixel sample are well distributed with respect to each other, and simultaneously such that the aggregate collection of sample values for all of the pixel samples in the pixel are collectively well distributed.

This sequence uses the first two dimensions of a low-discrepancy sequence derived by Sobol'.<sup>9</sup> This sequence is a special type of low-discrepancy sequence known as a (0, 2)-sequence. (0, 2)-sequences are stratified in a very general way. For example, the first 16 samples in a (0, 2)-sequence satisfy the stratification constraint from stratified sampling in Section 7.3, meaning there is just one sample in each of the boxes of extent  $(\frac{1}{4}, \frac{1}{4})$ . However, they also satisfy the Latin hypercube constraint, as only one of them is in each of the boxes of extent  $(\frac{1}{16}, 1)$  and  $(1, \frac{1}{16})$ . Furthermore, there is only one sample in each of the boxes of extent  $(\frac{1}{2}, \frac{1}{8})$  and  $(\frac{1}{8}, \frac{1}{2})$ . Figure 7.28 shows all of the possibilities for dividing the domain into regions where the first 16 samples of a (0, 2)-sequence satisfy the stratification properties. Each succeeding sequence of 16 samples from this pattern also satisfies these distribution properties.

In general, any sequence of length  $2^{l_1+l_2}$  (where  $l_i$  is a nonnegative integer) from a (0, 2)-sequence satisfies this general stratification constraint. The set of *elementary intervals* in two dimensions, base 2, is defined as

$$E = \left\{ \left[ \frac{a_1}{2^{l_1}}, \frac{a_1+1}{2^{l_1}} \right) \times \left[ \frac{a_2}{2^{l_2}}, \frac{a_2+1}{2^{l_2}} \right) \right\},$$

where the integer  $a_i = 0, 1, 2, 4, \dots, 2^{l_i} - 1$ . One sample from each of the first  $2^{l_1+l_2}$  values in the sequence will be in each of the elementary intervals. Furthermore, the same property is true for each subsequent set of  $2^{l_1+l_2}$  values.

To understand now how (0, 2)-sequences can be applied to generating 2D samples, consider a pixel with  $2 \times 2$  image samples, each with an array of  $4 \times 4$  2D samples. The first  $(2 \times 2) \times (4 \times 4) = 2^6$  values of a (0, 2)-sequence are well distributed with respect to each other according to the corresponding set of elementary intervals. Furthermore, the first  $4 \times 4 = 2^4$  samples are themselves well distributed according to their corresponding elementary intervals, as are the next  $2^4$  of them, and the subsequent ones, and so on. Therefore, we can use the first 16 (0, 2)-sequence samples for the samples for the  $4 \times 4$  array for the first image sample for a pixel, then the next 16 for the next image sample, and so forth. The result is an extremely well-distributed set of sample points.

### 7.5.1 SAMPLING WITH GENERATOR MATRICES

The Sobol' sequence is based on a different mechanism for generating sample points than the `HaltonSampler`, which used the radical inverse in various dimensions. Even with the integer divides in the radical inverse function converted to multiplies and shifts, the amount of computation needed to compute the billions of samples that can be needed for high-quality, high-resolution renderings can still be significant. Most of the computational expense comes from the cost of performing non-base-2 computation on computers that natively operate in base 2. (Consider the contrast between the `<Compute base-2 radical inverse>` fragment and the `RadicalInverseSpecialized()` template function.)

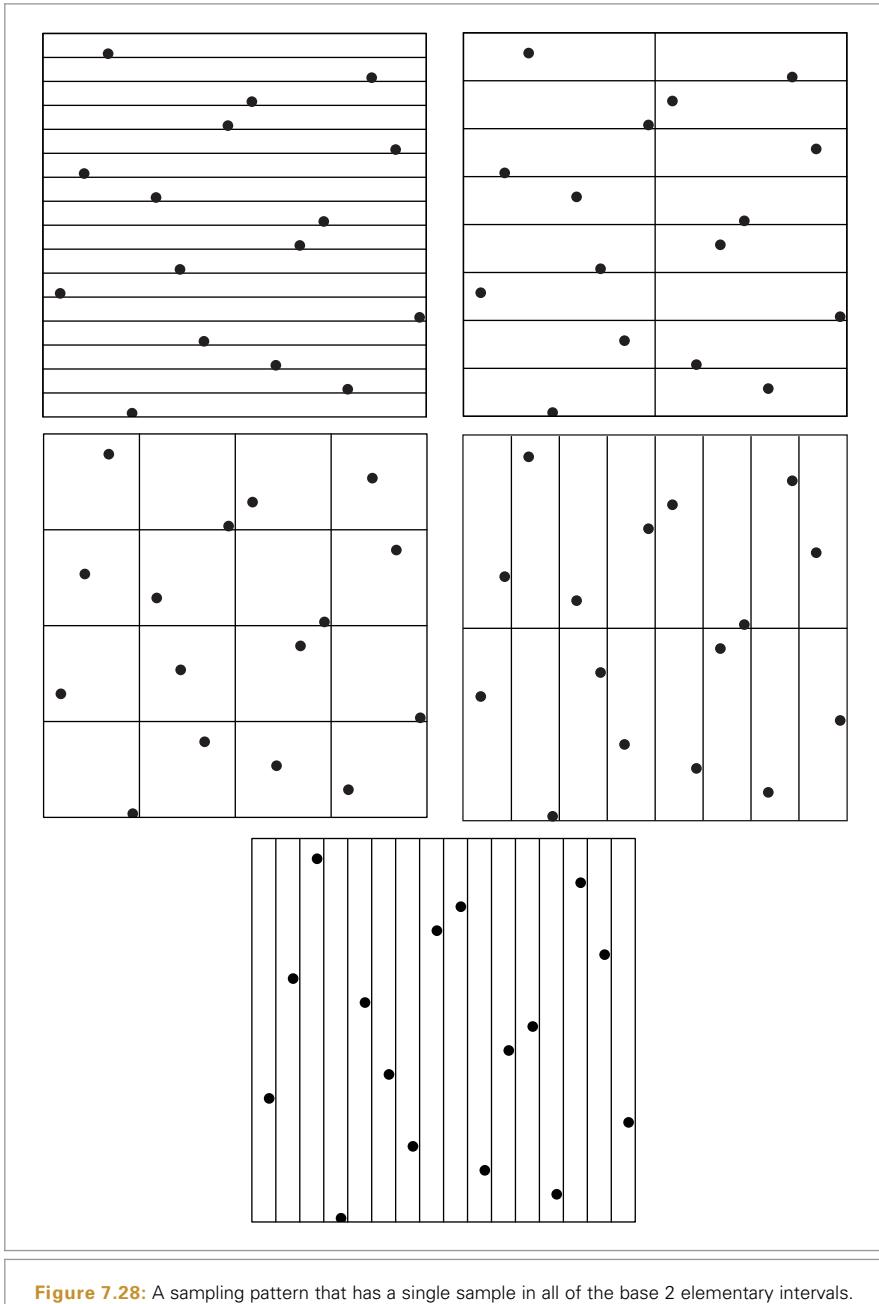
Given the high cost of non-base-2 operations, it's natural to try to develop sample generation algorithms that operate entirely in base 2. One such approach that has been effective

`RadicalInverseSpecialized()`  
447

`SobolSampler` 468

---

9 The `SobolSampler` in Section 7.7 uses all of the dimensions of the Sobol' sequence.



**Figure 7.28:** A sampling pattern that has a single sample in all of the base 2 elementary intervals. It satisfies both the  $4 \times 4$  stratification and Latin hypercube constraints as well as the other two stratification constraints shown.

has been to use *generator matrices* that allow all computation to be done in the same base. Instead of using a different base in each dimension, as the Halton sampler did, a different generator matrix is used in each dimension. With well-chosen matrices for each sampled dimension, it's possible to generate very good low-discrepancy distributions of points. For example, (0, 2)-sequences can be defined using two specific generator matrices in base 2.

To see how generator matrices are used, consider an  $n$ -digit number  $a$  in base  $b$ , where the  $i$ th digit of  $a$  is  $d_i(a)$  and where we have an  $n \times n$  generator matrix  $C$ . Then the corresponding sample point  $x_a \in [0, 1]$  is defined by

$$x_a = [b^{-1} b^{-2} \dots b^n] \begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & \ddots & & c_{2,n} \\ \vdots & & \ddots & \vdots \\ c_{n,1} & \cdots & \cdots & c_{n,n} \end{bmatrix} \begin{bmatrix} d_1(a) \\ d_2(a) \\ \vdots \\ d_n(a) \end{bmatrix}, \quad [7.9]$$

where all arithmetic is performed in the ring  $\mathbb{Z}_b$  (in other words, when all operations are performed modulo  $b$ ). This construction gives a total of  $b^n$  points as  $a$  ranges from 0 to  $b^n - 1$ . If the generator matrix is the identity matrix, then this definition corresponds to the regular radical inverse, base  $b$ . (It's worth pausing to make sure you see this connection between Equations (7.7) and (7.9) before continuing.)

In this section, we will exclusively use  $b = 2$  and  $n = 32$ . While introducing a  $32 \times 32$  matrix to the sample generation algorithm may not seem like a step toward better performance, we'll see that in the end the sampling code can be mapped to an implementation that uses a small number of bit operations to perform this computation in an extremely efficient manner.

The first step toward high performance comes from the fact that we're working in base 2; as such, all entries of  $C$  are either 0 or 1 and thus we can represent either each row or each column of the matrix with a single unsigned 32-bit integer. We'll choose to represent columns of the matrix as `uint32_ts`; this choice leads to a very efficient algorithm for multiplying the  $d_i$  column vector by  $C$ .

Now consider the task of computing the  $C[d_i(a)]^T$  matrix-vector product; using the definition of matrix-vector multiplication, we have:

$$\begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & \ddots & & c_{2,n} \\ \vdots & & \ddots & \vdots \\ c_{n,1} & \cdots & \cdots & c_{n,n} \end{bmatrix} \begin{bmatrix} d_1(a) \\ d_2(a) \\ \vdots \\ d_n(a) \end{bmatrix} = d_1 \begin{bmatrix} c_{1,1} \\ c_{2,1} \\ \vdots \\ c_{n,1} \end{bmatrix} + \cdots + d_n \begin{bmatrix} c_{1,n} \\ c_{2,n} \\ \vdots \\ c_{n,n} \end{bmatrix}. \quad [7.10]$$

In other words, for each digit of  $d_i$  that has a value of 1, the corresponding column of  $C$  should be summed. This addition can in turn be performed very efficiently in  $\mathbb{Z}_2$ : in that setting, addition corresponds to the exclusive OR operation. (Consider the combinations of the two possible operand values—0 and 1—and the result of adding them mod 2, and compare to the values computed by exclusive OR with the same operand values.) Thus, the multiplication  $C[d_i(a)]^T$  is just a matter of exclusive ORing together the columns  $i$  of  $C$  where  $d_i(a)$ 's bit is 1. This computation is implemented in the `MultiplyGenerator()` function.

```
(Low Discrepancy Inline Functions) +≡
inline uint32_t MultiplyGenerator(const uint32_t *C, uint32_t a) {
    uint32_t v = 0;
    for (int i = 0; a != 0; ++i, a >>= 1)
        if (a & 1)
            v ^= C[i];
    return v;
}
```

Going back to Equation (7.9) now, if we denote the column vector from the product  $v = C[d_i(a)]^T$ , then consider now the vector product

$$x_a = \begin{bmatrix} 2^{-1} & 2^{-2} & \dots & 2^{-n} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \sum_{i=1}^{32} 2^{-i} v_i. \quad [7.11]$$

Because the entries of  $v$  are stored in a single **uint32\_t**, their value interpreted as a **uint32\_t** is

$$v = v_1 + 2v_2 + \dots = \sum_{i=1}^{32} 2^{i-1} v_i.$$

If we were to reverse the order of the bits in the **uint32\_t**, then we would have the value

$$v' = \sum_{i=1}^{32} 2^{32-i} v_i.$$

This is a more useful value: if we divide this value by  $2^{32}$ , we get Equation (7.11), which is  $x_a$ , the value we're trying to compute.

Thus, if we take the result of the `MultiplyGenerator()` function, reverse the order of the bits in the returned value (e.g., by using `ReverseBits32()`), and then divide that integer value by  $2^{32}$  to compute a floating-point value in  $[0, 1)$ , we've computed our sample value.

To save the small cost of reversing the bits, we can equivalently reverse the bits in all of the columns of the generator matrix  $C$  before passing it to `MultiplyGenerator()`. We will use that convention in the following.

To make  $(0, 2)$ -sequences useful in practice, we also need to be able to generate multiple different sets of 2D sample values for each image sample, and we would like to generate different sample values for each pixel. One approach to this problem would be to use carefully chosen nonoverlapping subsequences of the  $(0, 2)$ -sequence for each pixel.<sup>10</sup> Another approach is to randomly scramble the  $(0, 2)$ -sequence, giving a new  $(0, 2)$ -

---

<sup>10</sup> This approach is taken by the Sobol' sampler in Section 7.7.

sequence built by applying a random permutation to the base- $b$  digits of the values in the original sequence.

The scrambling approach we will use is due to Kollig and Keller (2002). It repeatedly partitions and shuffles the unit square  $[0, 1]^2$ . In each of the two dimensions, it first divides the square in half and then swaps the two halves with 50% probability. Then it splits each of the intervals  $[0, 0.5]$  and  $[0.5, 1]$  in half and randomly exchanges each of those two halves. This process continues recursively all of the bits of the base-2 representation have been processed. This process was carefully designed so that it preserves the low-discrepancy properties of the set of points; otherwise, the advantages of the  $(0, 2)$ -sequence would be lost from the scrambling. Figure 7.29 shows an unscrambled  $(0, 2)$ -sequence and two randomly scrambled variations of it.

Two things make the scrambling process efficient: first, because we are scrambling two sequences that are computed in base 2, the digits  $d_i$  of the sequences are all 0 or 1, and scrambling a particular digit is equivalent to exclusive-ORing it with 0 or 1. Second, the simplification is made such that at each level  $l$  of the recursive scrambling, the same decision will be made as to whether to swap each of the  $2^{l-1}$  pairs of subintervals or not. The result of these two design choices is that the scrambling can be encoded as a set of bits stored in a `uint32_t` and can be applied to the original digits via exclusive-OR operations.

The `SampleGeneratorMatrix()` function pulls these pieces together to generate sample values.

*(Low Discrepancy Inline Functions) +≡*

```
inline Float SampleGeneratorMatrix(const uint32_t *C, uint32_t a,
                                  uint32_t scramble = 0) {
    return (MultiplyGenerator(C, a) ^ scramble) * 0x1p-32f;
}
```

The `SampleGeneratorMatrix()` function is already fairly efficient, performing a handful of arithmetic operations each time through the loop in `MultiplyGenerator()` that runs for a number of iterations equal to the base-2 logarithm of the value `a`. Remarkably, it's possible to do even better by changing the order in which samples are generated, enumerating them in *Gray code* order.

With Gray codes, successive binary values differ in only a single bit; the third column of Table 7.4 shows the first eight integers in Gray code order. Note that not only does only a single bit change between any pair of values but also that in any power-of-two-sized number of values  $n$  starting from 0, the Gray code enumerates all of the values from 0 to  $n - 1$ , just in a different order than usual.

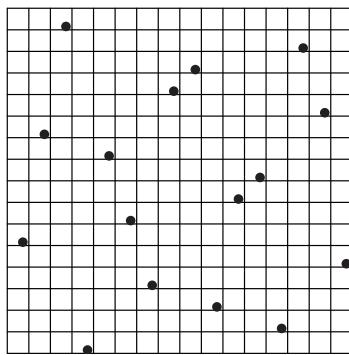
Computing the  $n$ th Gray code value can be done very efficiently.

*(Low Discrepancy Inline Functions) +≡*

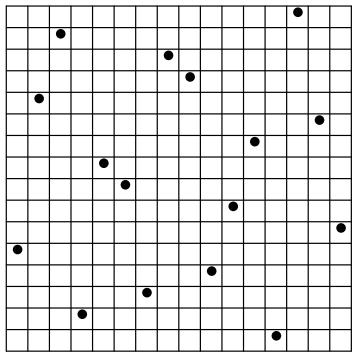
```
inline uint32_t GrayCode(uint32_t n) {
    return (n >> 1) ^ n;
}
```

Float 1062

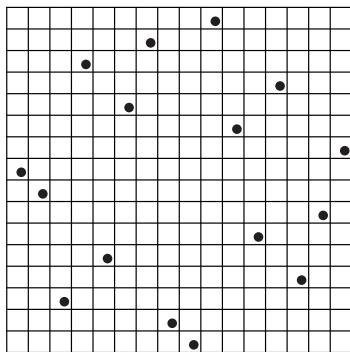
MultiplyGenerator() 458



(a)



(b)



(c)

**Figure 7.29:** (a) A low-discrepancy  $(0, 2)$ -sequence-based sampling pattern and (b, c) two randomly scrambled instances of it. Random scrambling of low-discrepancy patterns is an effective way to eliminate the artifacts that would be present in images if we used the same sampling pattern in every pixel, while still preserving the low-discrepancy properties of the point set being used.

**Table 7.4: The First Eight Integers, in Gray Code Order.** Each Gray code value  $g(n)$  differs by just a single bit from the previous one,  $g(n - 1)$ . The index of the bit that changes is given by the number of trailing zeros in the binary value  $n$ . Note that within any power-of-two-sized set of  $n$  values starting from 0, all of the integers between 0 and  $n - 1$  are represented, just in a different order than usual.

$n$ (base 10)	$n$ (binary)	$g(n)$	Changed Bit Index
0	000	000	n/a
1	001	001	0
2	010	011	1
3	011	010	0
4	100	110	2
5	101	111	0
6	110	101	1
7	111	100	0

By enumerating samples in Gray code order, we can take great advantage of the fact that only a single bit of  $g(n)$  changes between subsequent samples. Assume that we have computed the product  $C[d_i(a)]^T = v$  for some index  $a$ : if another value  $a'$  differs by just one bit from  $a$ , then we only need to add or subtract one column of  $C$  from  $v$  to find  $v' = C[d_I(a')]^T$  (recall Equation (7.10)). Even better, *both* addition and subtraction mod 2 can be performed with exclusive OR, so it doesn't matter which operation is needed; we only need to know which bit changed. As can be seen from Table 7.4, the index of the bit that changes going from  $g(i)$  to  $g(i + 1)$  is given by the number of trailing 0s in the binary representation of  $i + 1$ . Most CPU instruction sets can count trailing 0 bits in a single instruction.

Putting this all together, we can very efficiently generate a series of samples using a generator matrix in Gray code order. `GrayCodeSample()` takes a generator matrix  $C$ , a number of samples to generate  $n$ , and stores the corresponding samples in memory at the location pointed to by  $p$ .

*(Low Discrepancy Inline Functions) +≡*

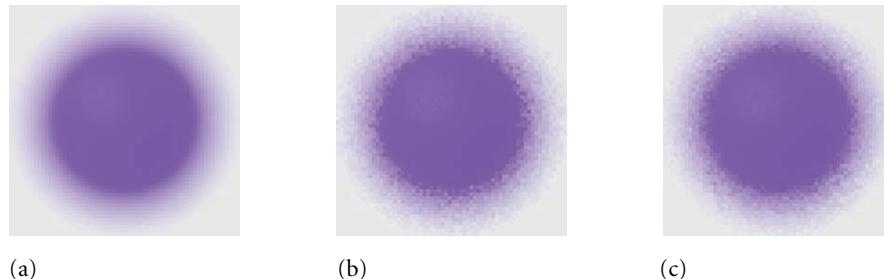
```
inline void GrayCodeSample(const uint32_t *C, uint32_t n,
                           uint32_t scramble, Float *p) {
    uint32_t v = scramble;
    for (uint32_t i = 0; i < n; ++i) {
        p[i] = v * 0x1p-32f; /* 1/2^32 */
        v ^= C[CountTrailingZeros(i + 1)];
    }
}
```

The x86 assembly code for heart of the inner loop (with the loop control logic elided) is wonderfully brief:

```
xorps      %xmm1, %xmm1
cvtsi2ssq %rax, %xmm1
mulss      %xmm0, %xmm1
movss      %xmm1, (%rcx,%rdx,4)
incq      %rdx
bsfl      %edx, %eax
xorl      $31, %eax
xorl      (%rdi,%rax,4), %esi
```

Even if one isn't an x86 assembly language aficionado, one can appreciate that it's an incredibly short sequence of instructions to generate each sample value.

There is a second version of `GrayCodeSample()` (not included here) for generating 2D samples; it takes a generator matrix for each dimension and fills in an array of `Point2f` values with the samples.



**Figure 7.30: Comparisons of the Stratified and  $(0, 2)$ -Sequence Samplers for Rendering Depth of Field.** (a) A reference image of the blurred edge of an out-of-focus sphere, (b) an image rendered using the `StratifiedSampler`, and (c) an image using the `ZeroTwoSequenceSampler`. The `ZeroTwoSequenceSampler`'s results are better than the stratified image, although the difference is smaller than the difference between stratified and random sampling.

## 7.5.2 SAMPLER IMPLEMENTATION

The `ZeroTwoSequenceSampler` generates samples for positions on the film plane, lens, and other 2D samples using scrambled  $(0, 2)$ -sequences, and generates 1D samples with scrambled van der Corput sequences. Figure 7.30 compares the result of using a  $(0, 2)$ -sequence for sampling the lens for the depth of field to using a stratified pattern.

```
(ZeroTwoSequenceSampler Declarations) ≡
class ZeroTwoSequenceSampler : public PixelSampler {
public:
    (ZeroTwoSequenceSampler Public Methods 462)
};
```

The constructor rounds the number of samples per pixel up to a power of 2 if necessary, since subsets of  $(0, 2)$ -sequences that are not a power of 2 in size are much less well distributed over  $[0, 1]^2$  than those that are.

```
(ZeroTwoSequenceSampler Method Definitions) ≡
ZeroTwoSequenceSampler::ZeroTwoSequenceSampler(int64_t samplesPerPixel,
                                                int nSampledDimensions)
: PixelSampler(RoundUpPow2(samplesPerPixel), nSampledDimensions) {
}
```

```
(ZeroTwoSequenceSampler Public Methods) ≡
int RoundCount(int count) const { return RoundUpPow2(count); }
```

Since the `ZeroTwoSequenceSampler` is a `PixelSampler`, its `StartPixel()` method must not only generate array sample values for the samples in the pixel but must also generate samples for a number of dimensions of non-array samples.

`PixelSampler` 427  
`RoundUpPow2()` 1064  
`StratifiedSampler` 434  
`ZeroTwoSequenceSampler` 462

```
<ZeroTwoSequenceSampler Method Definitions> +≡
    void ZeroTwoSequenceSampler::StartPixel(const Point2i &p) {
        <Generate 1D and 2D pixel sample components using (0, 2)-sequence 463>
        <Generate 1D and 2D array samples using (0, 2)-sequence 463>
        PixelSampler::StartPixel(p);
    }
```

Generating the samples for the non-array dimensions expected by the PixelSampler is a matter of filling in the appropriate vectors with the appropriate number of sample values.

```
<Generate 1D and 2D pixel sample components using (0, 2)-sequence> ≡ 463
    for (size_t i = 0; i < samples1D.size(); ++i)
        VanDerCorput(1, samplesPerPixel, &samples1D[i][0], rng);
    for (size_t i = 0; i < samples2D.size(); ++i)
        Sobol2D(1, samplesPerPixel, &samples2D[i][0], rng);
```

The sample vector dimensions with array samples are similar, though with multiple sample values in each dimension.

```
<Generate 1D and 2D array samples using (0, 2)-sequence> ≡ 463
    for (size_t i = 0; i < samples1DArraySizes.size(); ++i)
        VanDerCorput(samples1DArraySizes[i], samplesPerPixel,
                     &sampleArray1D[i][0], rng);
    for (size_t i = 0; i < samples2DArraySizes.size(); ++i)
        Sobol2D(samples2DArraySizes[i], samplesPerPixel,
                 &sampleArray2D[i][0], rng);
```

```
Float 1062
GrayCodeSample() 461
PixelSampler 427
PixelSampler::rng 428
PixelSampler::samples1D 427
PixelSampler::samples2D 427
Point2i 68
RNG 1065
RNG::UniformUInt32() 1066
Sampler::sampleArray1D 426
Sampler::sampleArray2D 426
Sampler::samples1DArraySizes
    426
Sampler::samples2DArraySizes
    426
Sampler::samplesPerPixel 422
Sampler::StartPixel() 422
Sobol2D() 464
VanDerCorput() 463
```

The VanDerCorput() function generates a number of scrambled 1D sample values using the Gray code-based sampling machinery. Although a specialized implementation of this function that took advantage of the structure of the identity matrix could be written, here we use the existing Gray code implementation, which is more than sufficiently efficient.

```
<Low Discrepancy Inline Functions> +≡
    inline void VanDerCorput(int nSamplesPerPixelSample, int nPixelSamples,
                            Float *samples, RNG &rng) {
        uint32_t scramble = rng.UniformUInt32();
        <Define CVanDerCorput Generator Matrix 464>
        int totalSamples = nSamplesPerPixelSample * nPixelSamples;
        GrayCodeSample(CVanDerCorput, totalSamples, scramble, samples);
        <Randomly shuffle 1D sample points 464>
    }
```

The generator matrix for the 1D van der Corput sequence is just the identity matrix but with each column's bits reversed, as per the earlier convention.

```
{Define CVanDerCorput Generator Matrix} ≡ 463
  const uint32_t CVanDerCorput[] = {
    0b10000000000000000000000000000000,
    0b10000000000000000000000000000000,
    0b10000000000000000000000000000000,
    0b10000000000000000000000000000000,
    {Remainder of Van Der Corput generator matrix entries}
  };
```

There is a subtle implementation detail that must be accounted for when using scrambled  $(0, 2)$ -sequences.<sup>11</sup> Often, integrators will use samples from more than one of the sampling patterns that the sampler creates in the process of computing the values of particular integrals. For example, they might use a sample from a 1D pattern to select one of the  $N$  light sources in the scene to sample illumination from and then might use a sample from a 2D pattern to select a sample point on that light source, if it is an area light.

Even if these two patterns are computed with random scrambling with different random scramble values for each one, some correlation can still remain between elements of these patterns, such that the  $i$ th element of the 1D pattern and the  $i$ th element of the 2D pattern are related. As such, in the earlier area lighting example, the distribution of sample points on each light source would not in general cover the entire light due to this correlation, leading to unusual rendering errors.

This problem can be solved easily enough by randomly shuffling the various dimensions individually after they are generated. After generating a scrambled 1D low-discrepancy sampling pattern, giving a well-distributed set of samples across all of the image samples for this pixel, this function shuffles these samples in two ways. Consider, for example, a pixel with 8 image samples, each of which has 4 1D samples for the integrator (giving a total of 32 integrator samples). First, it shuffles samples within each of the 8 groups of 4 samples, putting each set of 4 into a random order. Next, it shuffles each of the 8 groups of 4 samples as a block, with respect to the other blocks of 4 samples.

```
{Randomly shuffle 1D sample points} ≡ 463
  for (int i = 0; i < nPixelSamples; ++i)
    Shuffle(samples + i * nSamplesPerPixelSample,
            nSamplesPerPixelSample, 1, rng);
  Shuffle(samples, nPixelSamples, nSamplesPerPixelSample, rng);
```

The `Sobol2D()` function follows a similar structure to `VanDerCorput()` but uses two generator matrices to generate the first two dimensions of Sobol' points. Its implementation isn't included here.

```
(Low Discrepancy Declarations) +≡
  inline void Sobol2D(int nSamplesPerPixelSample, int nPixelSamples,
                      Point2f *samples, RNG &rng);
```

Point2f [68](#)  
RNG [1065](#)  
Shuffle() [438](#)  
VanDerCorput() [463](#)

---

<sup>11</sup> Indeed, the importance of this issue wasn't fully appreciated by the authors until after going through the process of debugging some unexpected noise patterns in rendered images when this sampler was being used.



(a)



(b)

**Figure 7.31:** When the `ZeroTwoSequenceSampler` is used for the area light sampling example, similar results are generated (a) with both 1 image sample and 16 light samples as well as (b) with 16 image samples and 1 light sample, thanks to the  $(0, 2)$ -sequence sampling pattern that ensures good distribution of samples over the pixel area in both cases. Compare these images to Figure 7.24, where the stratified pattern generates a much worse set of light samples when only 1 light sample is taken for each of the 16 image samples.

Figure 7.31 shows the result of using the  $(0, 2)$ -sequence for the area lighting example scene. Note that not only does it give a visibly better image than stratified patterns, but it also does well with one light sample per image sample, unlike the stratified sampler.

## \* 7.6 MAXIMIZED MINIMAL DISTANCE SAMPLER

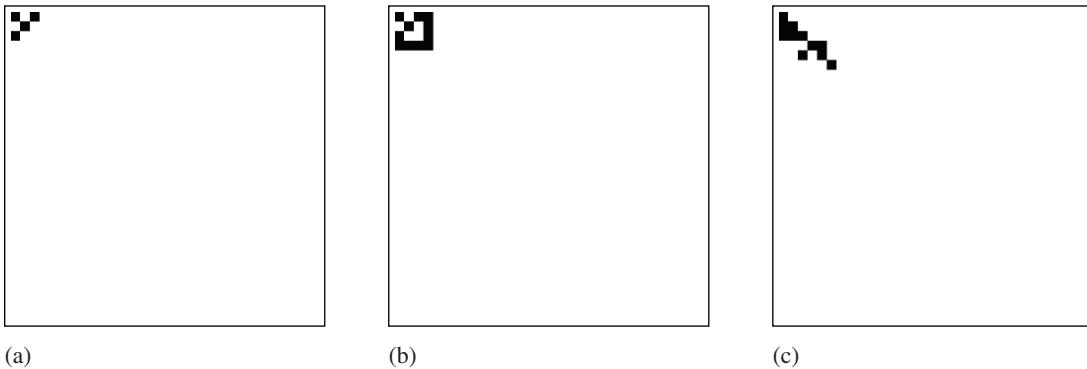
The  $(0, 2)$ -sequence sampler is more effective than the stratified sampler, thanks to being stratified over all elementary intervals. However, it still sometimes generates sample points that are close together. An alternative is to use a different pair of generator matrices that not only generate  $(0, 2)$ -sequences but that are also specially designed to maximize the distance between samples; this approach is implemented by the `MaxMinDistSampler`. (See the “Further Reading” section for more details about the origin of these generator matrices.)

*(MaxMinDistSampler Declarations) ≡*

```
class MaxMinDistSampler : public PixelSampler {
public:
    (MaxMinDistSampler Public Methods 466)
private:
    (MaxMinDistSampler Private Data 466)
};
```

`PixelSampler` 427

`ZeroTwoSequenceSampler` 462



**Figure 7.32:** Generator matrices for  $n = 8, 16$ , and  $64$  sample patterns for the `MaxMinDistSampler`. As before, all matrix elements are either 0 or 1, and 1 elements are shown as filled squares here.

There are 17 of these specialized matrices, one for each power-of-two number of samples up to  $2^{17}$  samples; a pointer to the appropriate one is stored in `CPixel` in the constructor.

*(MaxMinDistSampler Public Methods)  $\equiv$*

465

```
MaxMinDistSampler(int64_t samplesPerPixel, int nSampledDimensions)
    : PixelSampler(RoundUpPow2(samplesPerPixel), nSampledDimensions) {
    CPixel = CMaxMinDist[Log2Int(samplesPerPixel)];
}
```

*(MaxMinDistSampler Private Data)  $\equiv$*

465

```
const uint32_t *CPixel;
```

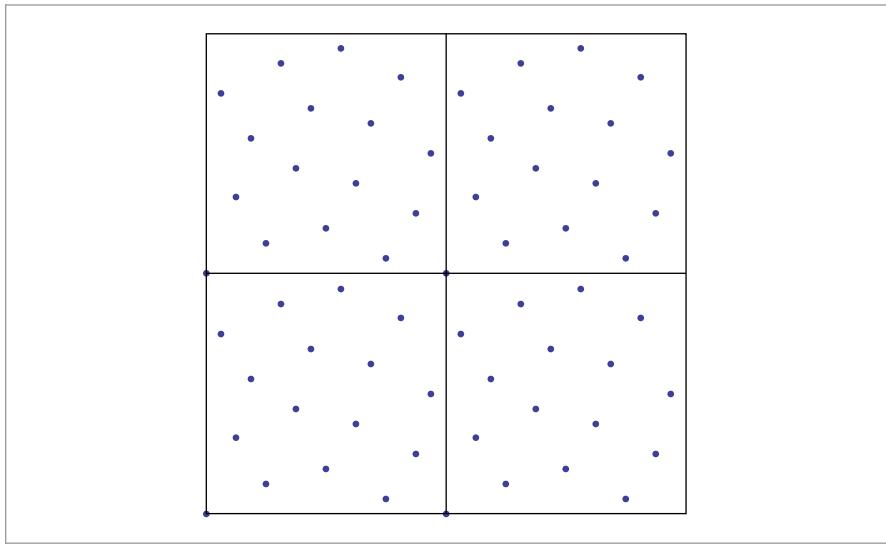
Figure 7.32 shows a few of these matrices and Figure 7.33 shows the points that one of them generates. Note that the same sampling pattern is used in each of the  $2 \times 2$  pixels shown there; when the matrices were found, distance between sample points was evaluated using *toroidal topology*—as if the unit square was rolled into a torus—to allow for high-quality sample tiling.

*(Low Discrepancy Declarations)  $+ \equiv$*

```
extern uint32_t CMaxMinDist[17][32];
```

The `MaxMinDistSampler` uses the generator matrix to compute the pixel samples. The first 2D sample dimension's value is set by uniformly stepping in the first dimension and the second comes from the generator matrix.

CMaxMinDist 466  
`Log2Int()` 1064  
`MaxMinDistSampler` 465  
`MaxMinDistSampler::CPixel` 466  
`PixelSampler` 427  
`Sampler::samplesPerPixel` 422



**Figure 7.33:** A grid of  $2 \times 2$  pixels, each sampled with 16 samples from the `MaxMinDistSampler`. Though the same sample points are used in each pixel, their placement has been optimized so that not only are they well distributed within each pixel, but when they are tiled across pixels, sample points also aren't too close to those in neighboring pixels.

*(MaxMinDistSampler Method Definitions)  $\equiv$*

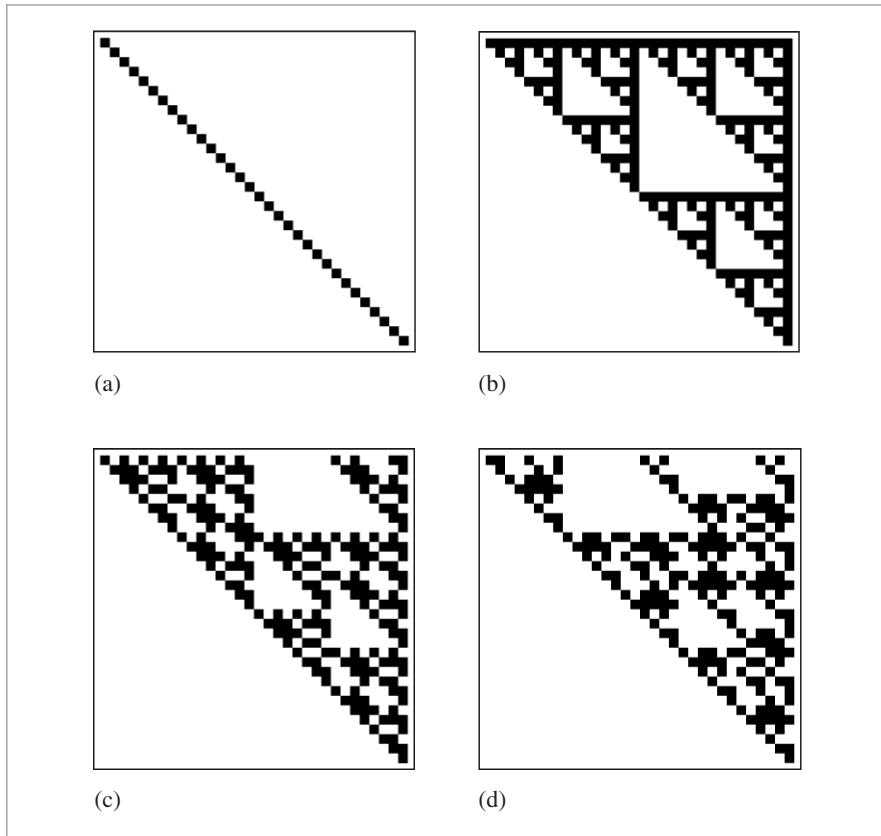
```
void MaxMinDistSampler::StartPixel(const Point2i &p) {
    Float invSPP = (Float)1 / samplesPerPixel;
    for (int i = 0; i < samplesPerPixel; ++i)
        samples2D[0][i] = Point2f(i * invSPP,
                                    SampleGeneratorMatrix(CPixel, i));
    Shuffle(&samples2D[0][0], samplesPerPixel, 1, rng);
    (Generate remaining samples for MaxMinDistSampler)
    PixelSampler::StartPixel(p);
}
```

CMaxMinDist 466  
 Float 1062  
 MaxMinDistSampler 465  
`MaxMinDistSampler::CPixel` 466  
`PixelSampler::samples2D` 427  
 Point2f 68  
 Point2i 68  
`SampleGeneratorMatrix()` 459  
`Sampler::samplesPerPixel` 422  
`Sampler::StartPixel()` 422  
 ZeroTwoSequenceSampler 462

The remaining dimensions are sampled using the first two Sobol' matrices, like the `ZeroTwoSequenceSampler`. We have found slightly better results with this approach (versus using the `CMaxMinDist` matrices) for samples in non-image dimensions of the sample vector. Therefore, the corresponding fragment *(Generate remaining samples for MaxMinDistSampler)* isn't included here.

## ★ 7.7 SOBOL' SAMPLER

The last Sampler in this chapter is based on a series of generator matrices due to Sobol'. The samples from the sequence that these matrices generate are distinguished by both being very efficient to implement—thanks to being entirely based on base-2



**Figure 7.34:** Generator matrices for the first four dimensions of the Sobol' sequence. Note their regular structure.

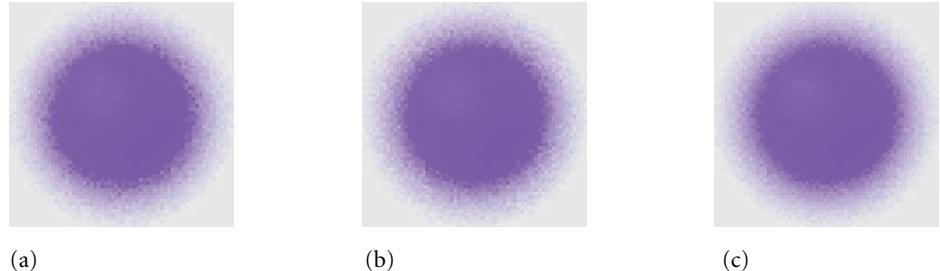
computations—while also being extremely well distributed over all  $n$  dimensions of the sample vector. Figure 7.34 shows the first few Sobol' generator matrices, and Figure 7.35 compares it to stratified and Halton points with the depth of field scene.

The weakness of the Sobol' points is that they are prone to structural grid artifacts before convergence; a sense of this issue can be seen in the image sample points shown in Figure 7.36 and in the images in Figure 7.37. In exchange for this weakness, Sobol' sequences are extremely well distributed over all  $n$  dimensions of the sample vector.

*(SobolSampler Declarations)* ≡

```
class SobolSampler : public GlobalSampler {
public:
    (SobolSampler Public Methods 470)
private:
    (SobolSampler Private Data 470)
};
```

GlobalSampler 429

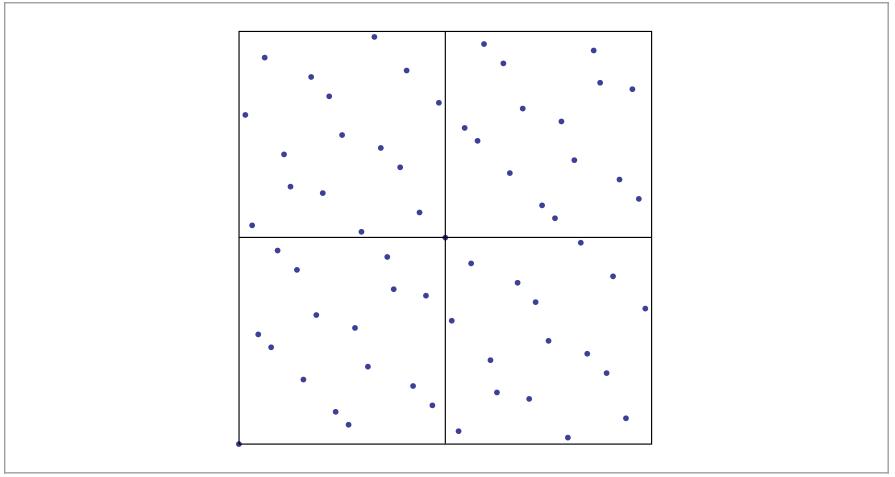


(a)

(b)

(c)

**Figure 7.35: Comparisons of the Stratified, Halton, and Sobol' Samplers for Rendering Depth of Field.** (a) An image rendered using the `StratifiedSampler`, (b) an image rendered using the `HaltonSampler`, and (c) an image using the `SobolSampler`. Both low-discrepancy samplers are better than the stratified sampler. In spite of the structured grid artifacts visible with this undersampled image with the `SobolSampler`, the Sobol' sequence often provides a faster rate of convergence than the Halton sequence.



**Figure 7.36:** A grid of  $2 \times 2$  pixels, sampled with 16 Sobol' samples each. Note that there is a fair amount of structure as well as many samples close to others. The very good distribution properties of the sequence over all  $n$  dimensions of the sample vector generally make up for these shortcomings.



**Figure 7.37:** Undersampled images rendered with (a) the Halton sampler and (b) the Sobol' sampler. Both exhibit visible structure, though with different visual characteristics. The Sobol' sequence in particular exhibits a clearly visible checkerboard structure.

The SobolSampler uniformly scales the first two dimensions by the smallest power of 2 that causes the  $[0, 1]^2$  sample domain to cover the image area to be sampled. As with the HaltonSampler, this specific scaling scheme is chosen in order to make it easier to compute the reverse mapping from pixel coordinates to the sample indices that land in each pixel.

*(SobolSampler Public Methods) ≡* 468

```
SobolSampler(int64_t samplesPerPixel, const Bounds2i &sampleBounds)
    : GlobalSampler(RoundUpPow2(samplesPerPixel)),
    sampleBounds(sampleBounds) {
    resolution = RoundUpPow2(std::max(sampleBounds.Diagonal().x,
        sampleBounds.Diagonal().y));
    log2Resolution = Log2Int(resolution);
}
```

*(SobolSampler Private Data) ≡* 468

```
const Bounds2i sampleBounds;
int resolution, log2Resolution;
```

The SobolIntervalToIndex() function returns the index of the sampleNum<sup>th</sup> sample in the pixel p, if the  $[0, 1]^2$  sampling domain has been scaled by  $2^{\log2Resolution}$  to cover the pixel sampling area.

*(Low Discrepancy Declarations) +≡*

```
inline uint64_t SobolIntervalToIndex(const uint32_t log2Resolution,
    uint64_t sampleNum, const Point2i &p);
```

The general approach used to derive the algorithm it implements is similar to that used by the Halton sampler in its GetIndexForSample() method. Here, scaling by a power of two means that the base-2 logarithm of the scale gives the number of digits of the  $C[d_i(a)]^T$  product that form the scaled sample's integer component. To find the values of  $a$  that give a particular integer value after scaling, we can compute the inverse of C: given

$$v = C[d_i(a)]^T,$$

then equivalently

$$C^{-1}v = [d_i(a)]^T.$$

We won't include the implementation of this method here.

*(SobolSampler Method Definitions) ≡*

```
int64_t SobolSampler::GetIndexForSample(int64_t sampleNum) const {
    return SobolIntervalToIndex(log2Resolution, sampleNum,
        Point2i(currentPixel - sampleBounds.pMin));
}
```

Bounds2::Diagonal() 80  
 Bounds2i 76  
 GlobalSampler 429  
 HaltonSampler 450  
 Log2Int() 1064  
 Point2i 68  
 RoundUpPow2() 1064  
 Sampler::currentPixel 425  
 SobolSample() 471  
 SobolSampler 468  
 SobolSampler::log2Resolution 470  
 SobolSampler::resolution 470  
 SobolSampler::sampleBounds 470

Computing the sample value for a given sample index and dimension is straightforward given the SobolSample() function.

```
<SobolSampler Method Definitions> +≡
    Float SobolSampler::SampleDimension(int64_t index, int dim) const {
        Float s = SobolSample(index, dim);
        <Remap Sobol' dimensions used for pixel samples 472>
        return s;
    }
```

The code for computing Sobol' sample values takes different paths for 32- and 64-bit floating-point values. Different generator matrices are used for these two cases, giving more bits of precision for 64-bit doubles.

```
<Low Discrepancy Inline Functions> +≡
    inline Float SobolSample(int64_t index, int dimension,
                             uint64_t scramble = 0) {
        #ifdef PBRT_FLOAT_AS_DOUBLE
            return SobolSampleDouble(index, dimension, scramble);
        #else
            return SobolSampleFloat(index, dimension, scramble);
        #endif
    }
```

The implementation of the `SobolSampleFloat()` function is quite similar to that of `MultiplyGenerator()`, with the differences that it takes a 64-bit index and that the matrices it uses have size  $32 \times 52$ . These larger matrices allow it to generate distinct sample values up to  $a = 2^{52} - 1$ , rather than  $2^{32} - 1$ , as with the  $32 \times 32$  matrices used previously.

```
<Low Discrepancy Inline Functions> +≡
    inline float SobolSampleFloat(int64_t a, int dimension,
                                 uint32_t scramble) {
        uint32_t v = scramble;
        for (int i = dimension * SobolMatrixSize; a != 0; a >= 1, ++i)
            if (a & 1)
                v ^= SobolMatrices32[i];
        return v * 0x1p-32f; /* 1/2^32 */
    }
```

Float 1062  
GlobalSampler 429  
MultiplyGenerator() 458  
NumSobolDimensions 471  
SobolMatrices32 471  
SobolMatrixSize 471  
SobolSample() 471  
SobolSampleDouble() 471  
SobolSampleFloat() 471

```
<Sobol Matrix Declarations> ≡
    static constexpr int NumSobolDimensions = 1024;
    static constexpr int SobolMatrixSize = 52;
    extern const uint32_t SobolMatrices32[NumSobolDimensions *
                                         SobolMatrixSize];
```

The `SobolSampleDouble()` function is similar, except that it uses 64-bit Sobol' matrices. It is not included in the text here.

Because the `SobolSampler` is a `GlobalSampler`, the values returned for the first two dimensions need to be adjusted so that they are offsets from the current pixel. Here, the sample

value is scaled up by the power-of-two scale computed in the constructor and then offset by the lower corner of the sample bounds to find the corresponding raster sample location. The current integer pixel coordinate is subtracted to get a result in [0, 1].

```
(Remap Sobol' dimensions used for pixel samples) ≡ 471
if (dim == 0 || dim == 1) {
    s = s * resolution + sampleBounds.pMin[dim];
    s = Clamp(s - currentPixel[dim], (Float)0, OneMinusEpsilon);
}
```

## 7.8 IMAGE RECONSTRUCTION

Given carefully chosen image samples, we need to convert the samples and their computed radiance values into pixel values for display or storage. According to signal processing theory, we need to do three things to compute final values for each of the pixels in the output image:

1. Reconstruct a continuous image function  $\tilde{L}$  from the set of image samples.
2. Prefilter the function  $\tilde{L}$  to remove any frequencies past the Nyquist limit for the pixel spacing.
3. Sample  $\tilde{L}$  at the pixel locations to compute the final pixel values.

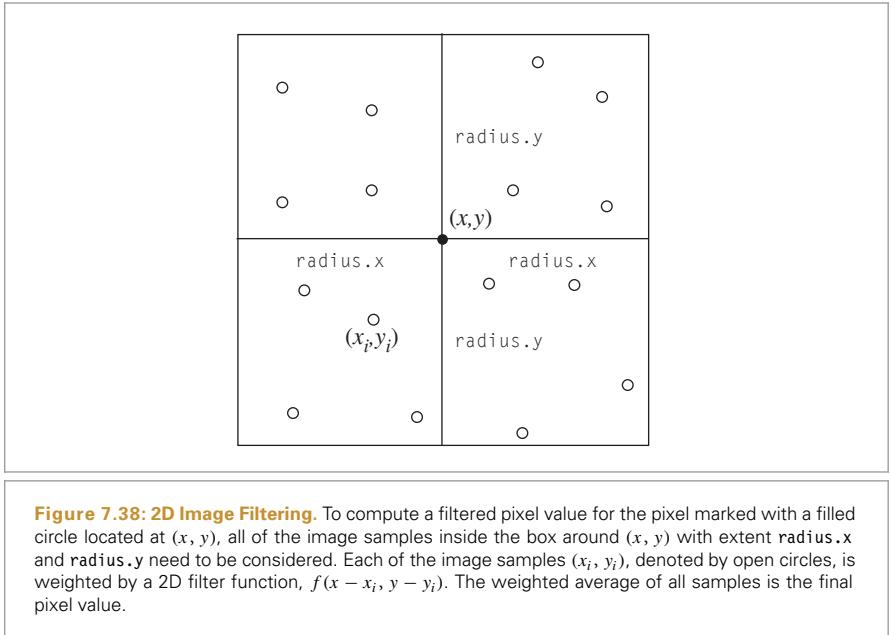
Because we know that we will be resampling the function  $\tilde{L}$  at only the pixel locations, it's not necessary to construct an explicit representation of the function. Instead, we can combine the first two steps using a single filter function.

Recall that if the original function had been uniformly sampled at a frequency greater than the Nyquist frequency and reconstructed with the sinc filter, then the reconstructed function in the first step would match the original image function perfectly—quite a feat since we only have point samples. But because the image function almost always will have higher frequencies than could be accounted for by the sampling rate (due to edges, etc.), we chose to sample it nonuniformly, trading off noise for aliasing.

The theory behind ideal reconstruction depends on the samples being uniformly spaced. While a number of attempts have been made to extend the theory to nonuniform sampling, there is not yet an accepted approach to this problem. Furthermore, because the sampling rate is known to be insufficient to capture the function, perfect reconstruction isn't possible. Recent research in the field of sampling theory has revisited the issue of reconstruction with the explicit acknowledgment that perfect reconstruction is not generally attainable in practice. This slight shift in perspective has led to powerful new reconstruction techniques. See, for example, Unser (2000) for a survey of these developments. In particular, the goal of research in reconstruction theory has shifted from perfect reconstruction to developing reconstruction techniques that can be shown to minimize error between the reconstructed function and the original function, *regardless of whether the original was band limited*.

While the reconstruction techniques used in pbrt are not directly built on these new approaches, they serve to explain the experience of practitioners that applying perfect

`Clamp()` 1062  
`Float` 1062  
`OneMinusEpsilon` 417  
`Sampler::currentPixel` 425  
`SobolSampler::resolution` 470



reconstruction techniques to samples taken for image synthesis generally does not result in the highest quality images.

To reconstruct pixel values, we will consider the problem of interpolating the samples near a particular pixel. To compute a final value for a pixel  $I(x, y)$ , interpolation results in computing a weighted average

$$I(x, y) = \frac{\sum_i f(x - x_i, y - y_i) w(x_i, y_i) L(x_i, y_i)}{\sum_i f(x - x_i, y - y_i)}, \quad [7.12]$$

where

- $L(x_i, y_i)$  is the radiance value of the  $i$ th sample located at  $(x_i, y_i)$
- $w(x_i, y_i)$  is the sample contribution weight returned by the camera. As described in Sections 6.4.7 and 13.6.6, the manner in which these weights are computed determines which radiometric quantity the film measures.
- $f$  is the filter function.

Figure 7.38 shows a pixel at location  $(x, y)$  that has a pixel filter with extent  $\text{radius.x}$  in the  $x$  direction and  $\text{radius.y}$  in the  $y$  direction. All of the samples inside the box given by the filter extent may contribute to the pixel's value, depending on the filter function's value for  $f(x - x_i, y - y_i)$ .

The sinc filter is not an appropriate choice here: recall that the ideal sinc filter is prone to ringing when the underlying function has frequencies beyond the Nyquist limit (Gibbs phenomenon), meaning edges in the image have faint replicated copies of the edge in nearby pixels. Furthermore, the sinc filter has *infinite support*: it doesn't fall off to zero at a finite distance from its center, so all of the image samples would need to be filtered

for each output pixel. In practice, there is no single best filter function. Choosing the best one for a particular scene takes a mixture of quantitative evaluation and qualitative judgment.

### 7.8.1 FILTER FUNCTIONS

All filter implementations in pbrt are derived from an abstract `Filter` class, which provides the interface for the  $f(x, y)$  functions used in filtering; see Equation (7.12). The `Film` class (described in the Section 7.9) stores a pointer to a `Filter` and uses it to filter image sample contributions when accumulating them into the final image. (Figure 7.39 shows comparisons of zoomed-in regions of images rendered using a variety of the filters from this section to reconstruct pixel values.) The `Filter` base class is defined in the files `core/filter.h` and `core/filter.cpp`.

*(Filter Declarations) ≡*

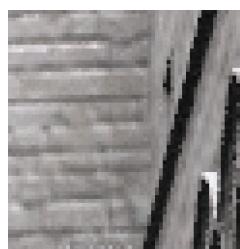
```
class Filter {
public:
    (Filter Interface 474)
    (Filter Public Data 475)
};
```

All filters are centered at the origin  $(0, 0)$  and define a radius beyond which they have a value of 0; this width may be different in the  $x$  and  $y$  directions. The constructor takes the radius values and stores them along with their reciprocals, for use by the filter implementations. The filter's overall extent in each direction (its *support*) is twice the value of its corresponding radius (Figure 7.40).

*(Filter Interface) ≡*

474

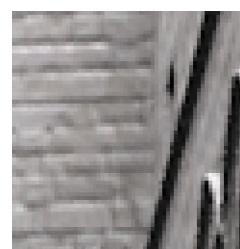
```
Filter(const Vector2f &radius)
: radius(radius),
invRadius(Vector2f(1 / radius.x, 1 / radius.y)) {}
```



(a)



(b)



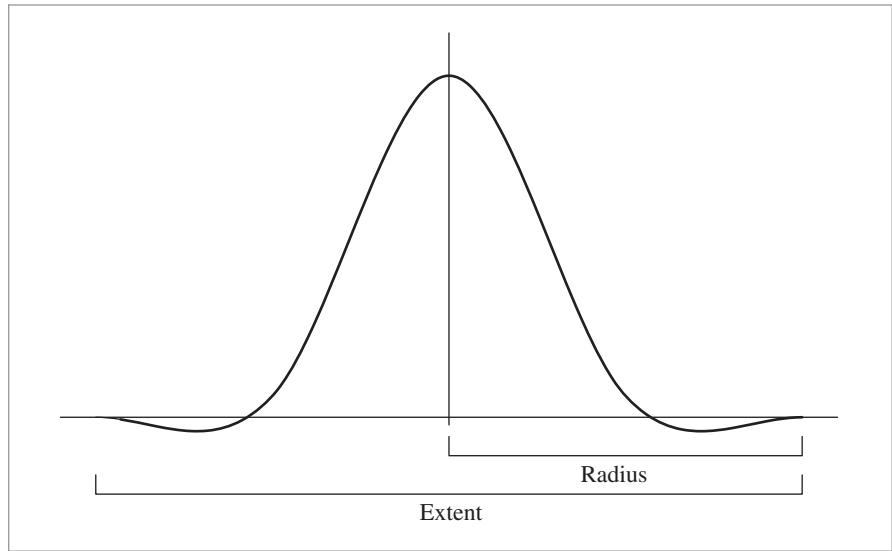
(c)

**Figure 7.39:** The pixel reconstruction filter used to convert the image samples into pixel values can have a noticeable effect on the character of the final image. Here, we see blowups of a region of the brick wall in the Sponza atrium scene, filtered with (a) the box filter, (b) Gaussian, and (c) Mitchell-Netravali filter. Note that the Mitchell filter gives the sharpest image, while the Gaussian blurs it. (Note artifacts on the top edges of arches, for example.)

Film 484

Filter 474

Vector2f 60



**Figure 7.40:** The extent of filters in pbrt is specified in terms of their radius from the origin to its cutoff point. The support of a filter is its total non-zero extent, here equal to twice its radius.

*{Filter Public Data}*  $\equiv$

474

```
const Vector2f radius, invRadius;
```

The sole method that *Filter* implementations need to provide is *Evaluate()*. It takes as a parameter a 2D point that gives the position of the sample point relative to the center of the filter. The filter's value at that point is returned. Code elsewhere in the system will never call the filter function with points outside of the filter's extent, so filter implementations don't need to check for this case.

*{Filter Interface}*  $+ \equiv$

474

```
virtual Float Evaluate(const Point2f &p) const = 0;
```

### Box Filter

One of the most commonly used filters in graphics is the *box filter* (and, in fact, when filtering and reconstruction aren't addressed explicitly, the box filter is the *de facto* result). The box filter equally weights all samples within a square region of the image. Although computationally efficient, it's just about the worst filter possible. Recall from the discussion in Section 7.1 that the box filter allows high-frequency sample data to leak into the reconstructed values. This causes postaliasing—even if the original sample values were at a high enough frequency to avoid aliasing, errors are introduced by poor filtering.

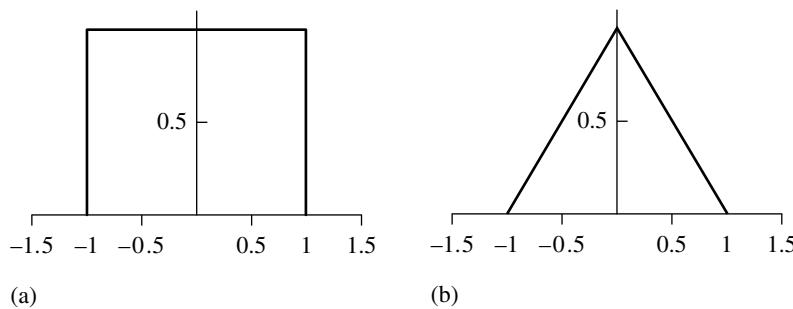
**Filter** 474

**Float** 1062

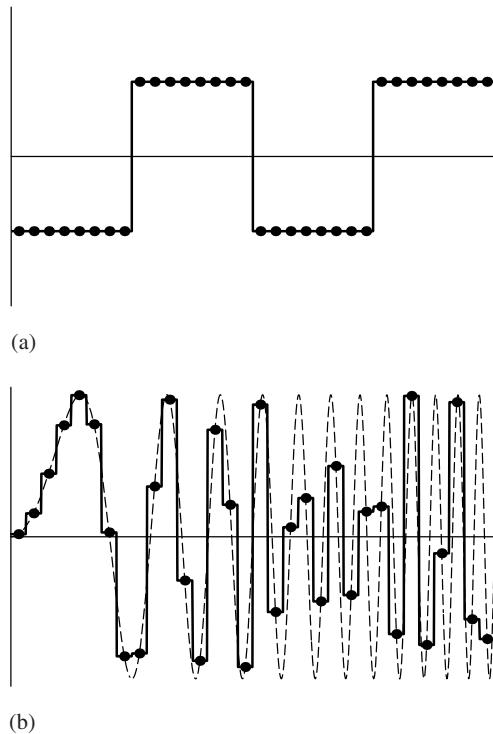
**Point2f** 68

**Vector2f** 60

Figure 7.41(a) shows a graph of the box filter, and Figure 7.42 shows the result of using the box filter to reconstruct two 1D functions. For the step function we used previously to illustrate the Gibbs phenomenon, the box does reasonably well. However, the results are much worse for a sinusoidal function that has increasing frequency along the *x* axis. Not only does the box filter do a poor job of reconstructing the function when the frequency is low, giving a discontinuous result even though the original function was smooth, but it



**Figure 7.41:** Graphs of the (a) box filter and (b) triangle filter. Although neither of these is a particularly good filter, they are both computationally efficient, easy to implement, and good baselines for evaluating other filters.



**Figure 7.42:** The box filter reconstructing (a) a step function and (b) a sinusoidal function with increasing frequency as  $x$  increases. This filter does well with the step function, as expected, but does an extremely poor job with the sinusoidal function.

also does an extremely poor job of reconstruction as the function's frequency approaches and passes the Nyquist limit.

```
(BoxFilter Declarations) ≡
class BoxFilter : public Filter {
public:
    BoxFilter(const Vector2f &radius) : Filter(radius) { }
    Float Evaluate(const Point2f &p) const;
};
```

Because the evaluation function won't be called with  $(x, y)$  values outside of the filter's extent, it can always return 1 for the filter function's value.

```
(BoxFilter Method Definitions) ≡
Float BoxFilter::Evaluate(const Point2f &p) const {
    return 1.;
}
```

### Triangle Filter

The triangle filter gives slightly better results than the box: samples at the filter center have a weight of 1, and the weight falls off linearly to the square extent of the filter. See Figure 7.41(b) for a graph of the triangle filter.

```
(TriangleFilter Declarations) ≡
class TriangleFilter : public Filter {
public:
    TriangleFilter(const Vector2f &radius) : Filter(radius) { }
    Float Evaluate(const Point2f &p) const;
};
```

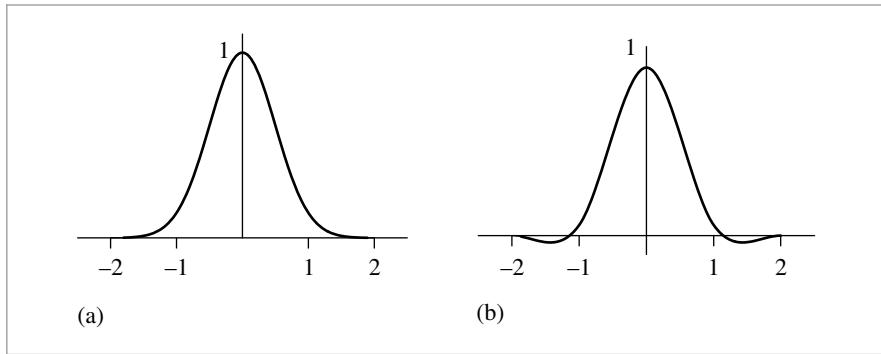
Evaluating the triangle filter is simple: the implementation just computes a linear function based on the width of the filter in both the  $x$  and  $y$  directions.

```
(TriangleFilter Method Definitions) ≡
Float TriangleFilter::Evaluate(const Point2f &p) const {
    return std::max((Float)0, radius.x - std::abs(p.x)) *
           std::max((Float)0, radius.y - std::abs(p.y));
}
```

BoxFilter 477  
 Filter 474  
 Filter::radius 475  
 Float 1062  
 Point2f 68  
 TriangleFilter 477  
 Vector2f 60

### Gaussian Filter

Unlike the box and triangle filters, the Gaussian filter gives a reasonably good result in practice. This filter applies a Gaussian bump that is centered at the pixel and radially symmetric around it. The Gaussian's value at the end of its extent is subtracted from the filter value, in order to make the filter go to 0 at its limit (Figure 7.43). The Gaussian does tend to cause slight blurring of the final image compared to some of the other filters, but this blurring can actually help mask any remaining aliasing in the image.



**Figure 7.43:** Graphs of (a) the Gaussian filter and (b) the Mitchell filter with  $B = \frac{1}{3}$  and  $C = \frac{1}{3}$ , each with a width of 2. The Gaussian gives images that tend to be a bit blurry, while the negative lobes of the Mitchell filter help to accentuate and sharpen edges in final images.

*(GaussianFilter Declarations) ≡*

```
class GaussianFilter : public Filter {
public:
    (GaussianFilter Public Methods 478)
private:
    (GaussianFilter Private Data 478)
    (GaussianFilter Utility Functions 479)
};
```

The 1D Gaussian filter function of radius  $r$  is

$$f(x) = e^{-\alpha x^2} - e^{-\alpha r^2},$$

where  $\alpha$  controls the rate of falloff of the filter. Smaller values cause a slower falloff, giving a blurrier image. The second term here ensures that the Gaussian goes to 0 at the end of its extent rather than having an abrupt cliff. For efficiency, the constructor precomputes the constant term for  $e^{-\alpha r^2}$  in each direction.

*(GaussianFilter Public Methods) ≡*

478

```
GaussianFilter(const Vector2f &radius, Float alpha)
    : Filter(radius), alpha(alpha),
      expX(std::exp(-alpha * radius.x * radius.x)),
      expY(std::exp(-alpha * radius.y * radius.y)) {}
```

Filter 474

Filter::radius 475

Float 1062

GaussianFilter 478

GaussianFilter::alpha 478

GaussianFilter::expX 478

GaussianFilter::expY 478

Vector2f 60

*(GaussianFilter Private Data) ≡*

478

```
const Float alpha;
const Float expX, expY;
```

Since a 2D Gaussian function is separable into the product of two 1D Gaussians, the implementation calls the `Gaussian()` function twice and multiplies the results.

```
(GaussianFilter Method Definitions) ≡
    Float GaussianFilter::Evaluate(const Point2f &p) const {
        return Gaussian(p.x, expX) * Gaussian(p.y, expY);
    }
```

*(GaussianFilter Utility Functions) ≡* 478

```
    Float Gaussian(Float d, Float expv) const {
        return std::max((Float)0, Float(std::exp(-alpha * d * d) - expv));
```

### Mitchell Filter

Filter design is notoriously difficult, mixing mathematical analysis and perceptual experiments. Mitchell and Netravali (1988) have developed a family of parameterized filter functions in order to be able to explore this space in a systematic manner. After analyzing test subjects' subjective responses to images filtered with a variety of parameter values, they developed a filter that tends to do a good job of trading off between *ringing* (phantom edges next to actual edges in the image) and *blurring* (excessively blurred results)—two common artifacts from poor reconstruction filters.

Note from the graph in Figure 7.43(b) that this filter function takes on negative values out by its edges; it has *negative lobes*. In practice these negative regions improve the sharpness of edges, giving crisper images (reduced blurring). If they become too large, however, ringing tends to start to enter the image. Also, because the final pixel values can therefore become negative, they will eventually need to be clamped to a legal output range.

Figure 7.44 shows this filter reconstructing the two test functions. It does extremely well with both of them: there is minimal ringing with the step function, and it does a very good job with the sinusoidal function, up until the point where the sampling rate isn't sufficient to capture the function's detail.

*(MitchellFilter Declarations) ≡*

```
class MitchellFilter : public Filter {
public:
    (MitchellFilter Public Methods 479)
```

*private:*

```
    const Float B, C;
};
```

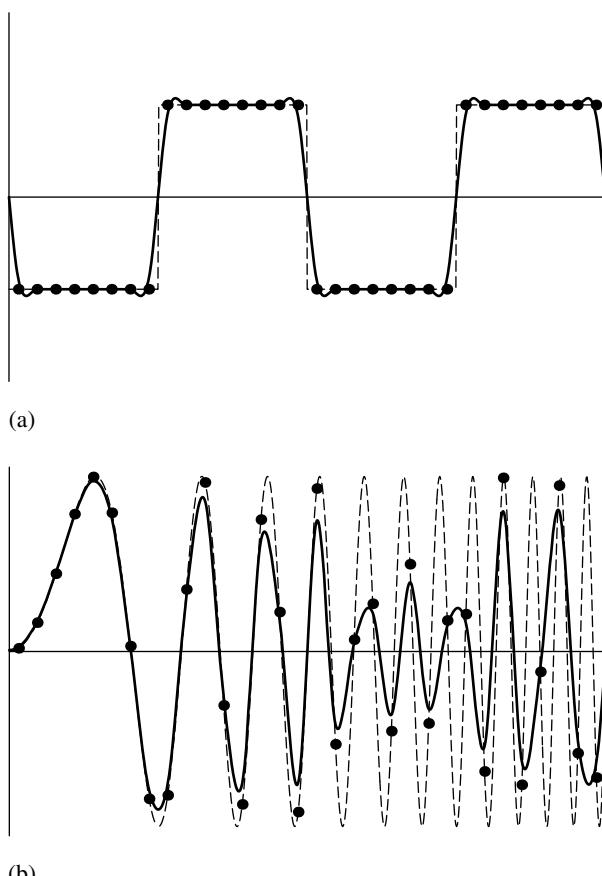
The Mitchell filter has two parameters called *B* and *C*. Although any values can be used for these parameters, Mitchell and Netravali recommend that they lie along the line  $B + 2C = 1$ .

*(MitchellFilter Public Methods) ≡* 479

```
MitchellFilter(const Vector2f &radius, Float B, Float C)
    : Filter(radius), B(B), C(C) {
```

Filter [474](#)  
 Filter::Evaluate() [475](#)  
 Float [1062](#)  
 GaussianFilter [478](#)  
 GaussianFilter::expX [478](#)  
 GaussianFilter::expY [478](#)  
 MitchellFilter [479](#)  
 Point2f [68](#)  
 Vector2f [60](#)

The Mitchell-Netravali filter is the product of 1D filter functions in the *x* and *y* directions and is therefore separable, like the Gaussian filter. (In fact, all of the provided filters in *pbrt* are separable.) Nevertheless, the *Filter::Evaluate()* interface does not enforce this requirement, giving more flexibility in implementing new filters in the future.



**Figure 7.44: The Mitchell–Netravali Filter Used to Reconstruct the Example Functions.** It does a good job with both of these functions, (a) introducing minimal ringing with the step function and (b) accurately representing the sinusoid until aliasing from undersampling starts to dominate.

*(MitchellFilter Method Definitions) ≡*

```
Float MitchellFilter::Evaluate(const Point2f &p) const {
    return Mitchell1D(p.x * invRadius.x) * Mitchell1D(p.y * invRadius.y);
}
```

The 1D function used in the Mitchell filter is an even function defined over the range  $[-2, 2]$ . This function is made by joining a cubic polynomial defined over  $[0, 1]$  with another cubic polynomial defined over  $[1, 2]$ . This combined polynomial is also reflected around the  $x = 0$  plane to give the complete function. These polynomials are controlled by the  $B$  and  $C$  parameters and are chosen carefully to guarantee  $C^0$  and  $C^1$  continuity at  $x = 0$ ,  $x = 1$ , and  $x = 2$ . The polynomials are

Filter::invRadius 475  
Float 1062  
MitchellFilter 479  
MitchellFilter::Mitchell1D() 481  
Point2f 68

$$f(x) = \frac{1}{6} \begin{cases} (12 - 9B - 6C)|x|^3 + (-18 + 12B + 6C)|x|^2 + (6 - 2B) & |x| < 1 \\ (-B - 6C)|x|^3 + (6B + 30C)|x|^2 + (-12B - 48C)|x| & 1 \leq |x| < 2 \\ 0 & \text{otherwise.} \end{cases}$$

*(MitchellFilter Public Methods)*  $\equiv$

```
Float Mitchell1D(Float x) const {
    x = std::abs(2 * x);
    if (x > 1)
        return ((-B - 6*C) * x*x*x + (6*B + 30*C) * x*x +
                (-12*B - 48*C) * x + (8*B + 24*C)) * (1.f/6.f);
    else
        return ((12 - 9*B - 6*C) * x*x*x +
                (-18 + 12*B + 6*C) * x*x +
                (6 - 2*B)) * (1.f/6.f);
}
```

479

### Windowed Sinc Filter

Finally, the LanczosSincFilter class implements a filter based on the sinc function. In practice, the sinc filter is often multiplied by another function that goes to 0 after some distance. This gives a filter function with finite extent, which is necessary for an implementation with reasonable performance. An additional parameter  $\tau$  controls how many cycles the sinc function passes through before it is clamped to a value of 0. Figure 7.45 shows a graph of three cycles of the sinc function, along with a graph of the windowing function we use, which was developed by Lanczos. The Lanczos window is just the central lobe of the sinc function, scaled to cover the  $\tau$  cycles:

$$w(x) = \frac{\sin \pi x / \tau}{\pi x / \tau}.$$

Figure 7.45 also shows the filter that we will implement here, which is the product of the sinc function and the windowing function.

Figure 7.46 shows the windowed sinc's reconstruction results for uniform 1D samples. Thanks to the windowing, the reconstructed step function exhibits far less ringing than the reconstruction using the infinite-extent sinc function (compare to Figure 7.11). The windowed sinc filter also does extremely well at reconstructing the sinusoidal function until prealiasing begins.

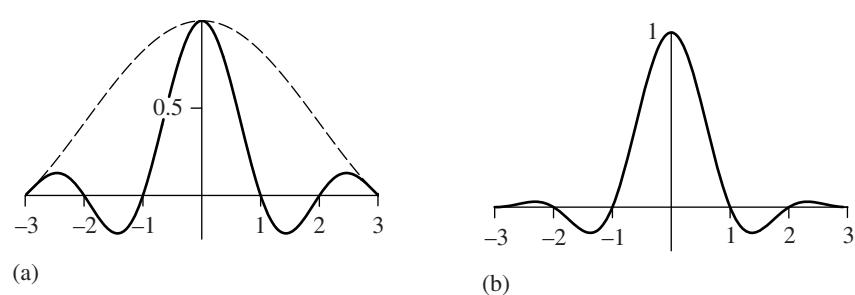
*(Sinc Filter Declarations)*  $\equiv$

```
class LanczosSincFilter : public Filter {
public:
    (LanczosSincFilter Public Methods 483)
private:
    const Float tau;
};
```

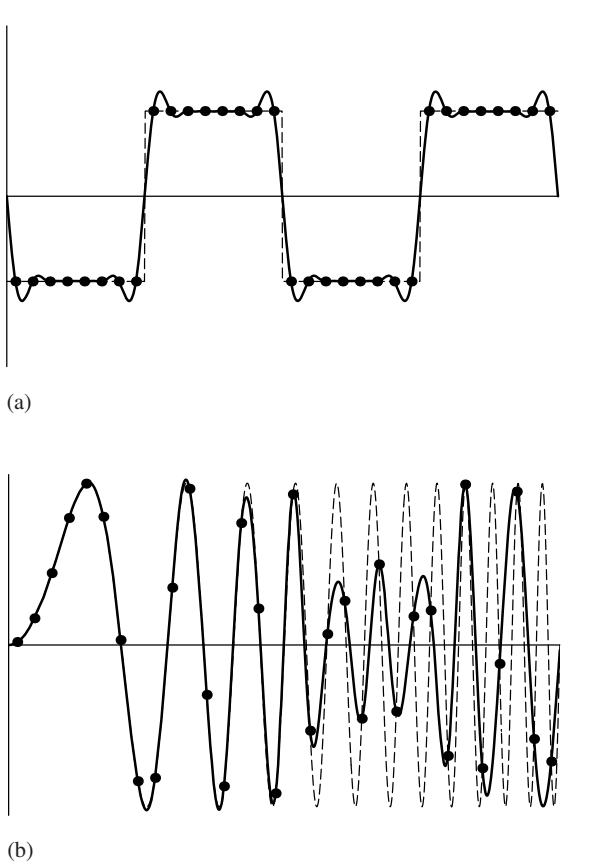
Filter 474

Float 1062

LanczosSincFilter 481



**Figure 7.45: Graphs of the Sinc Filter.** (a) The sinc function, truncated after three cycles (solid line) and the Lanczos windowing function (dashed line). (b) The product of these two functions, as implemented in the LanczosSincFilter.



**Figure 7.46: Results of Using the Windowed Sinc Filter to Reconstruct the Example Functions.** Here,  $\tau = 3$ . (a) Like the infinite sinc, it suffers from ringing with the step function, although there is much less ringing in the windowed version. (b) The filter does quite well with the sinusoid, however.

```
<LanczosSincFilter Public Methods> ≡ 481
    LanczosSincFilter(const Vector2f &radius, Float tau)
        : Filter(radius), tau(tau) { }

<Sinc Filter Method Definitions> ≡
    Float LanczosSincFilter::Evaluate(const Point2f &p) const {
        return WindowedSinc(p.x, radius.x) * WindowedSinc(p.y, radius.y);
    }
```

The implementation computes the value of the sinc function and then multiplies it by the value of the Lanczos windowing function.

```
<LanczosSincFilter Public Methods> +≡ 481
    Float Sinc(Float x) const {
        x = std::abs(x);
        if (x < 1e-5) return 1;
        return std::sin(Pi * x) / (Pi * x);
    }
```

```
<LanczosSincFilter Public Methods> +≡ 481
    Float WindowedSinc(Float x, Float radius) const {
        x = std::abs(x);
        if (x > radius) return 0;
        Float lanczos = Sinc(x / tau);
        return Sinc(x) * lanczos;
    }
```

## 7.9 FILM AND THE IMAGING PIPELINE

The type of film or sensor in a camera has a dramatic effect on the way that incident light is transformed into colors in an image. In pbrt, the `Film` class models the sensing device in the simulated camera. After the radiance is found for each camera ray, the `Film` implementation determines the sample's contribution to the pixels around the point on the film plane where the camera ray began and updates its representation of the image. When the main rendering loop exits, the `Film` writes the final image to a file.

```
Camera:::GenerateRayDifferential() 357
Film 484
Filter 474
Filter:::radius 475
Float 1062
LanczosSincFilter 481
LanczosSincFilter:::tau 481
LanczosSincFilter:::WindowedSinc() 483
Pi 1063
Point2f 68
Vector2f 60
```

For realistic camera models, Section 6.4.7 introduced the measurement equation, which describes how a sensor in a camera measures the amount of energy arriving over the sensor area over a period of time. For simpler camera models, we can consider the sensor to be measuring the average radiance over a small area over some period of time. The effect of the choice of which measurement to take is encapsulated in the weight for the ray returned by `Camera:::GenerateRayDifferential()`. Therefore, the `Film` implementation can proceed without having to account for these variations, as long as it scales the provided radiance values by these weights.

This section introduces a single `Film` implementation that applies the pixel reconstruction equation to compute final pixel values. For a physically based renderer, it's generally best for the resulting images to be stored in a floating-point image format. Doing so provides more flexibility in how the output can be used than if a traditional image format

with 8-bit unsigned integer values is used; floating-point formats avoid the substantial loss of information that comes from quantizing images to 8 bits.

In order to display such images on modern display devices, it is necessary to map these floating-point pixel values to discrete values for display. For example, computer monitors generally expect the color of each pixel to be described by an RGB color triple, not an arbitrary spectral power distribution. Spectra described by general basis function coefficients must therefore be converted to an RGB representation before they can be displayed. A related problem is that displays have a substantially smaller range of displayable radiance values than the range present in many real-world scenes. Therefore, pixel values must be mapped to the displayable range in a way that causes the final displayed image to appear as close as possible to the way it would appear on an ideal display device without this limitation. These issues are addressed by research into *tone mapping*; the “Further Reading” section has more information about this topic.

### 7.9.1 THE FILM CLASS

`Film` is defined in the files `core/film.h` and `core/film.cpp`.

*(Film Declarations)* ≡

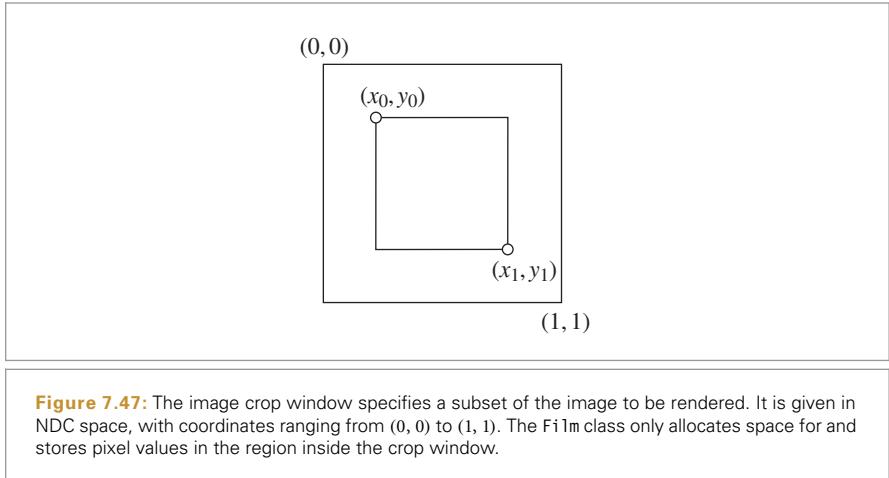
```
class Film {
public:
    (Film Public Methods)
    (Film Public Data 485)
private:
    (Film Private Data 486)
    (Film Private Methods 493)
};
```

A number of values are passed to the constructor: the overall resolution of the image in pixels; a crop window that may specify a subset of the image to render; the length of the diagonal of the film’s physical area, which is specified to the constructor in millimeters but is converted to meters here; a filter function; the filename for the output image and parameters that control how the image pixel values are stored in files.

*(Film Method Definitions)* ≡

```
Film::Film(const Point2i &resolution, const Bounds2f &cropWindow,
           std::unique_ptr<Filter> filt, Float diagonal,
           const std::string &filename, Float scale)
: fullResolution(resolution), diagonal(diagonal * .001),
  filter(std::move(filt)), filename(filename), scale(scale) {
    (Compute film image bounds 485)
    (Allocate film image storage 486)
    (Precompute filter weight table 487)
}
```

Bounds2f 76  
`Film` 484  
`Film::diagonal` 485  
`Film::filename` 485  
`Film::filter` 485  
`Film::fullResolution` 485  
`Film::scale` 496  
`Filter` 474  
`Float` 1062  
`Point2i` 68



*(Film Public Data)*  $\equiv$

```
const Point2i fullResolution;
const Float diagonal;
std::unique_ptr<Filter> filter;
const std::string filename;
```

484

In conjunction with the overall image resolution, the crop window gives the bounds of pixels that need to be actually stored and written out. Crop windows are useful for debugging or for breaking a large image into small pieces that can be rendered on different computers and reassembled later. The crop window is specified in NDC space, with each coordinate ranging from 0 to 1 (Figure 7.47). `Film::croppedPixelBounds` stores the pixel bounds from the upper-left to the lower-right corners of the crop window. Fractional pixel coordinates are rounded up; this ensures that if an image is rendered in pieces with abutting crop windows, each final pixel will be present in only one of the subimages.

*(Compute film image bounds)*  $\equiv$

```
croppedPixelBounds =
    Bounds2i(Point2i(std::ceil(fullResolution.x * cropWindow.pMin.x),
                     std::ceil(fullResolution.y * cropWindow.pMin.y)),
             Point2i(std::ceil(fullResolution.x * cropWindow.pMax.x),
                     std::ceil(fullResolution.y * cropWindow.pMax.y)));
```

484

*(Film Public Data)*  $+ \equiv$

```
Bounds2i croppedPixelBounds;
```

484

Given the pixel resolution of the (possibly cropped) image, the constructor allocates an array of `Pixel` structures, one for each pixel. The running weighted sums of spectral pixel contributions are represented using XYZ colors (Section 5.2.1) and are stored in the `xyz` member variable. `filterWeightSum` holds the sum of filter weight values for the sample contributions to the pixel. `splatXYZ` holds an (unweighted) sum of sample splats. The `pad` member is unused; its sole purpose is to ensure that the `Pixel` structure is 32 bytes in size,

`Bounds2i` 76

`Film` 484

`Film::croppedPixelBounds` 485

`Film::fullResolution` 485

`Filter` 474

`Float` 1062

`Pixel` 486

`Point2i` 68

rather than 28 as it would be otherwise (assuming 4-byte `Floats`; otherwise, it ensures a 64-byte structure). This padding ensures that a `Pixel` won't straddle a cache line, so that no more than one cache miss will be incurred when a `Pixel` is accessed (as long as the first `Pixel` in the array is allocated at the start of a cache line).

```
(Film Private Data) ≡ 484
struct Pixel {
    Float xyz[3] = { 0, 0, 0 };
    Float filterWeightSum = 0;
    AtomicFloat splatXYZ[3];
    Float pad;
};
std::unique_ptr<Pixel[]> pixels;
```

```
(Allocate film image storage) ≡ 484
pixels = std::unique_ptr<Pixel[]>(new Pixel[croppedPixelBounds.Area()]);
```

Two natural alternatives to using XYZ colors to store pixel values would be to use `Spectrum` values or to store RGB color. Here, it isn't worthwhile to store complete `Spectrum` values, even when doing full spectral rendering. Because the final colors written to the output file don't include the full set of `Spectrum` samples, converting to a tristimulus value here doesn't represent a loss of information versus storing `Spectrums` and converting to a tristimulus value on image output. Not storing complete `Spectrum` values in this case can save a substantial amount of memory if the `Spectrum` has a large number of samples. (If pbrt supported saving `SampledSpectrum` values to files, then this design choice would need to be revisited.)

We have chosen to use XYZ color rather than RGB to emphasize that XYZ is a display-independent representation of color, while RGB requires assuming a particular set of display response curves (Section 5.2.2). (In the end, we will, however, have to convert to RGB, since few image file formats store XYZ color.)

With typical filter settings, every image sample may contribute to 16 or more pixels in the final image. Particularly for simple scenes, where relatively little time is spent on ray intersection testing and shading computations, the time spent updating the image for each sample can be significant. Therefore, the `Film` precomputes a table of filter values so that we can avoid the expense of virtual function calls to the `Filter::Evaluate()` method as well as the expense of evaluating the filter and can instead use values from the table for filtering. The error introduced by not evaluating the filter at each sample's precise location isn't noticeable in practice.

The implementation here makes the reasonable assumption that the filter is defined such that  $f(x, y) = f(|x|, |y|)$ , so the table needs to hold values for only the positive quadrant of filter offsets. This assumption is true for all of the `Filters` currently available in pbrt and is true for most filters used in practice. This makes the table one-fourth the size and improves the coherence of memory accesses, leading to better cache performance.<sup>12</sup>

AtomicFloat 1086  
 Film 484  
 Film::croppedPixelBounds 485  
 Film::pixels 486  
 Filter 474  
 Filter::Evaluate() 475  
 Float 1062  
 Pixel 486  
 SampledSpectrum 319  
 Spectrum 315

---

<sup>12</sup> The implementation here could further take advantage of the fact that all filters currently in pbrt are separable, only allocating two 1D tables. However, to more easily allow different filter functions to be added, we don't assume separability here.

```
{Precompute filter weight table} ≡ 484
    int offset = 0;
    for (int y = 0; y < filterTableWidth; ++y) {
        for (int x = 0; x < filterTableWidth; ++x, ++offset) {
            Point2f p;
            p.x = (x + 0.5f) * filter->radius.x / filterTableWidth;
            p.y = (y + 0.5f) * filter->radius.y / filterTableWidth;
            filterTable[offset] = filter->Evaluate(p);
        }
    }
```

```
{Film Private Data} +≡ 484
    static constexpr int filterTableWidth = 16;
    Float filterTable[filterTableWidth * filterTableWidth];
```

The Film implementation is responsible for determining the range of integer pixel values that the Sampler is responsible for generating samples for. The area to be sampled is returned by the `GetSampleBounds()` method. Because the pixel reconstruction filter generally spans a number of pixels, the Sampler must generate image samples a bit outside of the range of pixels that will actually be output. This way, even pixels at the boundary of the image will have an equal density of samples around them in all directions and won't be biased with only values from toward the interior of the image. This detail is also important when rendering images in pieces with crop windows, since it eliminates artifacts at the edges of the subimages.

Computing the sample bounds involves accounting for the half-pixel offsets when converting from discrete to continuous pixel coordinates, expanding by the filter radius, and then rounding outward.

```
{Film Method Definitions} +≡
    Bounds2i Film::GetSampleBounds() const {
        Bounds2f floatBounds(
            Floor(Point2f(croppedPixelBounds.pMin) + Vector2f(0.5f, 0.5f) -
                  filter->radius),
            Ceil( Point2f(croppedPixelBounds.pMax) - Vector2f(0.5f, 0.5f) +
                  filter->radius));
        return (Bounds2i)floatBounds;
    }
```

Bounds2f 76  
 Bounds2i 76  
 Film 484  
 Film::filter 485  
 Filter::Evaluate() 475  
 Filter::radius 475  
 Float 1062  
 Point2f::Ceil() 71  
 Point2f::Floor() 71  
 Point2f 68  
 RealisticCamera 378  
 Sampler 421  
 Vector2f 60

`GetPhysicalExtent()` returns the actual extent of the film in the scene. This information is specifically needed by the `RealisticCamera`. Given the length of the film diagonal and the aspect ratio of the image, we can compute the size of the sensor in the  $x$  and  $y$  directions. If we denote the diagonal length by  $d$  and the width and height of the film sensor by  $x$  and  $y$ , then we know that  $x^2 + y^2 = d^2$ . We can define the aspect ratio  $a$  of the image by  $a = y/x$ , so  $y = ax$ , which gives  $x^2 + (a^2x^2) = d^2$ . Solving for  $x$  gives

$$x = \sqrt{\frac{d^2}{1 + a^2}}.$$

The implementation of `GetPhysicalExtent()` follows directly. The returned extent is centered around  $(0, 0)$ .

```
(Film Method Definitions) +≡
Bounds2f Film::GetPhysicalExtent() const {
    Float aspect = (Float)fullResolution.y / (Float)fullResolution.x;
    Float x = std::sqrt(diagonal * diagonal / (1 + aspect * aspect));
    Float y = aspect * x;
    return Bounds2f(Point2f(-x / 2, -y / 2), Point2f(x / 2, y / 2));
}
```

## 7.9.2 SUPPLYING PIXEL VALUES TO THE FILM

There are three ways that the sample contributions can be provided to the film. The first is driven by samples generated by the `Sampler` over tiles of the image. While the most straightforward interface would be to allow renderers to provide a film pixel location and a `Spectrum` with the contribution of the corresponding ray directly to the `Film`, it's not easy to provide a high-performance implementation of such a method in the presence of multi-threading, where multiple threads may end up trying to update the same portion of the image concurrently.

Therefore, `Film` defines an interface where threads can specify that they're generating samples in some extent of pixels with respect to the overall image. Given the sample bounds, `GetFilmTile()` in turn returns a pointer to a `FilmTile` object that stores contributions for the pixels in the corresponding region of the image. Ownership of the `FilmTile` and the data it stores is exclusive to the caller, so that thread can provide sample values to the `FilmTile` without worrying about contention with other threads. When it has finished work on the tile, the thread passes the completed tile back to the `Film`, which safely merges it into the final image.

```
(Film Method Definitions) +≡
std::unique_ptr<FilmTile> Film::GetFilmTile(
    const Bounds2i &sampleBounds) {
    <Bound image pixels that samples in sampleBounds contribute to 489>
    return std::unique_ptr<FilmTile>(new FilmTile(tilePixelBounds,
        filter->radius, filterTable, filterTableWidth));
}
```

Given a bounding box of the pixel area that samples will be generated in, there are two steps to compute the bounding box of image pixels that the sample values will contribute to. First, the effects of the discrete-to-continuous pixel coordinate transformation and the radius of the filter must be accounted for. Second, the resulting bound must be clipped to the overall image pixel bounds; pixels outside the image by definition don't need to be accounted for.

Bounds2f 76  
 Bounds2i 76  
 Film 484  
 Film::diagonal 485  
 Film::fullResolution 485  
 FilmTile 489  
 Float 1062  
 Point2f 68  
 Sampler 421  
 Spectrum 315

```

⟨Bound image pixels that samples in sampleBounds contribute to⟩ ≡ 488
    Vector2f halfPixel = Vector2f(0.5f, 0.5f);
    Bounds2f floatBounds = (Bounds2f)sampleBounds;
    Point2i p0 = (Point2i)Ceil(floatBounds.pMin - halfPixel -
                                filter->radius);
    Point2i p1 = (Point2i)Floor(floatBounds.pMax - halfPixel +
                                filter->radius) + Point2i(1, 1);
    Bounds2i tilePixelBounds =
        Intersect(Bounds2i(p0, p1), croppedPixelBounds);

```

```

⟨Film Declarations⟩ +≡
class FilmTile {
public:
    ⟨FilmTile Public Methods 489⟩
private:
    ⟨FilmTile Private Data 489⟩
};

```

The `FilmTile` constructor takes a 2D bounding box that gives the bounds of the pixels in the final image that it must provide storage for as well as additional information about the reconstruction filter being used, including a pointer to the filter function values tabulated in *(Precompute filter weight table)*.

```

⟨FilmTile Public Methods⟩ ≡ 489
    FilmTile(const Bounds2i &pixelBounds, const Vector2f &filterRadius,
             const Float *filterTable, int filterTableSize)
        : pixelBounds(pixelBounds), filterRadius(filterRadius),
          invFilterRadius(1 / filterRadius.x, 1 / filterRadius.y),
          filterTable(filterTable), filterTableSize(filterTableSize) {
            pixels = std::vector<FilmTilePixel>(std::max(0, pixelBounds.Area()));
        }

```

```

⟨FilmTile Private Data⟩ ≡ 489
    const Bounds2i pixelBounds;
    const Vector2f filterRadius, invFilterRadius;
    const Float *filterTable;
    const int filterTableSize;
    std::vector<FilmTilePixel> pixels;

```

Bounds2::Intersect() 78  
 Bounds2f 76  
 Bounds2i 76  
 FilmTile 489  
 FilmTilePixel 489  
 Float 1062  
 Point2::Ceil() 71  
 Point2::Floor() 71  
 Point2i 68  
 Spectrum 315  
 Vector2f 60

For each pixel, both a sum of the weighted contributions from the pixel samples (according to the reconstruction filter weights) and a sum of the filter weights is maintained.

```

⟨FilmTilePixel Declarations⟩ ≡
    struct FilmTilePixel {
        Spectrum contribSum = 0.f;
        Float filterWeightSum = 0.f;
    };

```

Once the radiance carried by a ray for a sample has been computed, the Integrator calls `FilmTile::AddSample()`. It takes a sample and corresponding radiance value as well as the weight for the sample's contribution originally returned by `Camera::GenerateRayDifferential()`. It updates the stored image using the reconstruction filter with the pixel filtering equation.

*(FilmTile Public Methods) +≡*

489

```
void AddSample(const Point2f &pFilm, const Spectrum &L,
              Float sampleWeight = 1.) {
    (Compute sample's raster bounds 490)
    (Loop over filter support and add sample to pixel arrays 491)
}
```

To understand the operation of `FilmTile::AddSample()`, first recall the pixel filtering equation:

$$I(x, y) = \frac{\sum_i f(x - x_i, y - y_i) w(x_i, y_i) L(x_i, y_i)}{\sum_i f(x - x_i, y - y_i)}.$$

It computes each pixel's value  $I(x, y)$  as the weighted sum of nearby samples' radiance values, using both a filter function  $f$  and the sample weight returned by the Camera  $w(x_i, y_i)$  to compute the contribution of the radiance value to the final pixel value. Because all of the Filters in pbrt have finite extent, this method starts by computing which pixels will be affected by the current sample. Then, turning the pixel filtering equation inside out, it updates two running sums for each pixel  $(x, y)$  that is affected by the sample. One sum accumulates the numerator of the pixel filtering equation, and the other accumulates the denominator. At the end of rendering, the final pixel values are computed by performing the division.

To find which pixels a sample potentially contributes to, `FilmTile::AddSample()` converts the continuous sample coordinates to discrete coordinates by subtracting 0.5 from  $x$  and  $y$ . It then offsets this value by the filter radius in each direction (Figure 7.48), transforms it to the tile coordinate space, and takes the ceiling of the minimum coordinates and the floor of the maximum, since pixels outside the bound of the extent are unaffected by the sample. Finally, the pixel bounds are clipped to the bounds of the pixels in the tile. While the sample may theoretically contribute to pixels outside the tile, any such pixels must be outside the image extent.

*(Compute sample's raster bounds) ≡*

490

```
Point2f pFilmDiscrete = pFilm - Vector2f(0.5f, 0.5f);
Point2i p0 = (Point2i)Ceil(pFilmDiscrete - filterRadius);
Point2i p1 = (Point2i)Floor(pFilmDiscrete + filterRadius) + Point2i(1, 1);
p0 = Max(p0, pixelBounds.pMin);
p1 = Min(p1, pixelBounds.pMax);
```

Given the bounds of pixels that are affected by this sample, it's now possible to loop over all of those pixels and accumulate the filtered sample weights at each of them.

```
Camera::
  GenerateRayDifferential()
  357

FilmTile::AddSample() 490

Filter 474

Float 1062

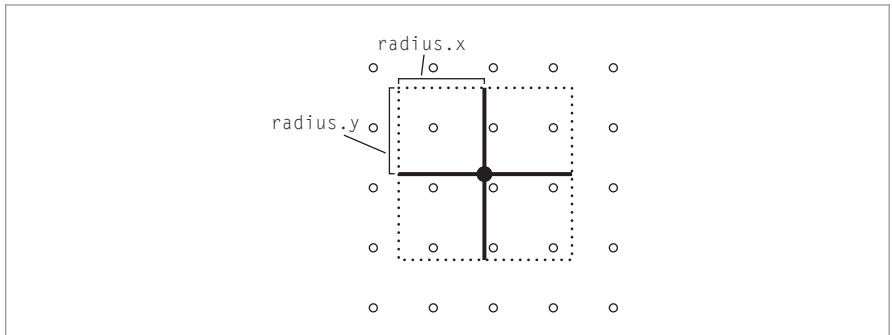
Integrator 25

Point2::Ceil() 71
Point2::Floor() 71
Point2::Max() 70
Point2::Min() 70

Point2f 68
Point2i 68

Spectrum 315

Vector2f 60
```



**Figure 7.48:** Given an image sample at some position on the image plane (solid dot), it is necessary to determine which pixel values (empty dots) are affected by the sample's contribution. This is done by taking the offsets in the  $x$  and  $y$  directions according to the pixel reconstruction filter's radius (solid lines) and finding the pixels inside this region.

```

<Loop over filter support and add sample to pixel arrays> ≡ 490
    <Precompute x and y filter table offsets 492>
    for (int y = p0.y; y < p1.y; ++y) {
        for (int x = p0.x; x < p1.x; ++x) {
            <Evaluate filter value at (x, y) pixel 492>
            <Update pixel values with filtered sample contribution 492>
        }
    }
}

```

Each discrete integer pixel  $(x, y)$  has an instance of the filter function centered around it. To compute the filter weight for a particular sample, it's necessary to find the offset from the pixel to the sample's position in discrete coordinates and evaluate the filter function. If we were evaluating the filter explicitly, the appropriate computation would be

```
filterWeight = filter->Evaluate(Point2i(x - pFilmDiscrete.x,
                                         y - pFilmDiscrete.y));
```

Instead, the implementation retrieves the appropriate filter weight from the table.

To find the filter weight for a pixel  $(x', y')$  given the sample position  $(x, y)$ , this routine computes the offset  $(x' - x, y' - y)$  and converts it into coordinates for the filter weights lookup table. This can be done directly by dividing each component of the sample offset by the filter radius in that direction, giving a value between 0 and 1, and then multiplying by the table size. This process can be further optimized by noting that along each row of pixels in the  $x$  direction, the difference in  $y$ , and thus the  $y$  offset into the filter table, is constant. Analogously, for each column of pixels, the  $x$  offset is constant. Therefore, before looping over the pixels here it's possible to precompute these indices and store them in two 1D arrays, saving repeated work in the loop.

*(Precompute x and y filter table offsets) ≡*

```

int *ifx = ALLOCA(int, p1.x - p0.x);
for (int x = p0.x; x < p1.x; ++x) {
    Float fx = std::abs((x - pFilmDiscrete.x) *
                         invFilterRadius.x * filterTableSize);
    ifx[x - p0.x] = std::min((int)std::floor(fx), filterTableSize - 1);
}
int *ify = ALLOCA(int, p1.y - p0.y);
for (int y = p0.y; y < p1.y; ++y) {
    Float fy = std::abs((y - pFilmDiscrete.y) *
                         invFilterRadius.y * filterTableSize);
    ify[y - p0.y] = std::min((int)std::floor(fy), filterTableSize - 1);
}
```

491

Now at each pixel, the  $x$  and  $y$  offsets into the filter table can be found for the pixel, leading to the offset into the array and thus the filter value.

*(Evaluate filter value at (x, y) pixel) ≡*

```

int offset = ify[y - p0.y] * filterTableSize + ifx[x - p0.x];
Float filterWeight = filterTable[offset];
```

491

For each affected pixel, we can now add its weighted spectral contribution and the filter weight to the appropriate value in the `pixels` array.

*(Update pixel values with filtered sample contribution) ≡*

```

FilmTilePixel &pixel = GetPixel(Point2i(x, y));
pixel.contribSum += L * sampleWeight * filterWeight;
pixel.filterWeightSum += filterWeight;
```

491

The `GetPixel()` method takes pixel coordinates with respect to the overall image and converts them to coordinates in the film tile before indexing into the `pixels` array. In addition to the version here, there is also a `const` variant of the method that returns a `const FilmTilePixel &`.

*(FilmTile Public Methods) +≡*

```

FilmTilePixel &GetPixel(const Point2i &p) {
    int width = pixelBounds.pMax.x - pixelBounds.pMin.x;
    int offset = (p.x - pixelBounds.pMin.x) +
                 (p.y - pixelBounds.pMin.y) * width;
    return pixels[offset];
}
```

489

Rendering threads present `FilmTiles` to be merged into the image stored by `Film` using the `MergeFilmTile()` method. Its implementation starts by acquiring a lock to a mutex in order to ensure that multiple threads aren't simultaneously modifying image pixel values. Note that because `MergeFilmTile()` takes a `std::unique_ptr` to the tile, ownership of the tile's memory is transferred when this method is called. Calling code should therefore no longer attempt to add contributions to a tile after calling this method. Storage for the

ALLOCA() 1071  
 FilmTile 489  
 FilmTile::GetPixel() 492  
 FilmTile::pixelBounds 489  
 FilmTile::pixels 489  
 FilmTilePixel 489  
 FilmTilePixel::contribSum 489  
 FilmTilePixel::filterWeightSum 489  
 Float 1062  
 Point2i 68

FilmTile is freed automatically at the end of the execution of MergeFilmTile() when the tile parameter goes out of scope.

```
(Film Method Definitions) +≡
void Film::MergeFilmTile(std::unique_ptr<FilmTile> tile) {
    std::lock_guard<std::mutex> lock(mutex);
    for (Point2i pixel : tile->GetPixelBounds()) {
        (Merge pixel into Film::pixels 493)
    }
}
```

*(Film Private Data) +≡*

484

```
std::mutex mutex;
```

When merging a tile's contributions in the final image, it's necessary for calling code to be able to find the bound of pixels that the tile has contributions for.

*(FilmTile Public Methods) +≡*

489

```
Bounds2i GetPixelBounds() const { return pixelBounds; }
```

For each pixel in the tile, it's just necessary to merge its contribution into the values stored in Film::pixels.

*(Merge pixel into Film::pixels) ≡*

493

```
Bounds2i 76
Film::croppedPixelBounds 485
Film::GetPixel() 493
Film::mutex 493
Film::pixels 486
FilmTile 489
FilmTile::GetPixel() 492
FilmTile::GetPixelBounds()
    493
FilmTile::pixelBounds 489
FilmTilePixel 489
FilmTilePixel::contribSum
    489
FilmTilePixel::
    filterWeightSum
    489
Float 1062
Integrator 25
Pixel 486
Pixel::filterWeightSum 486
Pixel::xyz 486
Point2i 68
Spectrum::ToXYZ() 324
```

*(Film Private Methods) ≡*

484

```
Pixel &GetPixel(const Point2i &p) {
    int width = croppedPixelBounds.pMax.x - croppedPixelBounds.pMin.x;
    int offset = (p.x - croppedPixelBounds.pMin.x) +
        (p.y - croppedPixelBounds.pMin.y) * width;
    return pixels[offset];
}
```

It's also useful for some Integrator implementations to be able to just provide values for all of the pixels in the entire image all at once. The SetImage() method allows this mode of operation. Note that number of elements in the array pointed to by the image parameter should be equal to croppedPixelBounds.Area(). The implementation of SetImage() is a straightforward matter of copying the given values after converting them to XYZ color.

```
(Film Method Definitions) +≡
void Film::SetImage(const Spectrum *img) const {
    int nPixels = croppedPixelBounds.Area();
    for (int i = 0; i < nPixels; ++i) {
        Pixel &p = pixels[i];
        img[i].ToXYZ(p.xyz);
        p.filterWeightSum = 1;
        p.splatXYZ[0] = p.splatXYZ[1] = p.splatXYZ[2] = 0;
    }
}
```

Some light transport algorithms (notably bidirectional path tracing, which is introduced in Section 16.3) require the ability to “splat” contributions to arbitrary pixels. Rather than computing the final pixel value as a weighted average of contributing splats, splats are simply summed. Generally, the more splats that are around a given pixel, the brighter the pixel will be. The `Pixel::splatXYZ` member variable is declared to be of `AtomicFloat` type, which allows multiple threads to concurrently update pixel values via the `AddSplat()` method without additional synchronization.

```
(Film Method Definitions) +≡
void Film::AddSplat(const Point2f &p, const Spectrum &v) {
    if (!InsideExclusive((Point2i)p, croppedPixelBounds))
        return;
    Float xyz[3];
    v.ToXYZ(xyz);
    Pixel &pixel = GetPixel((Point2i)p);
    for (int i = 0; i < 3; ++i)
        pixel.splatXYZ[i].Add(xyz[i]);
}
```

[AtomicFloat 1086](#)  
[AtomicFloat::Add\(\) 1087](#)  
[Bounds2::InsideExclusive\(\) 79](#)  
[Film 484](#)  
[Film::croppedPixelBounds 485](#)  
[Film::GetPixel\(\) 493](#)  
[Film::pixels 486](#)  
[Film::WriteImage\(\) 494](#)  
[Float 1062](#)  
[MLTIntegrator 1035](#)  
[Pixel 486](#)  
[Pixel::filterWeightSum 486](#)  
[Pixel::splatXYZ 486](#)  
[Pixel::xyz 486](#)  
[Point2f 68](#)  
[Point2i 68](#)  
[Spectrum 315](#)  
[Spectrum::ToXYZ\(\) 324](#)

### 7.9.3 IMAGE OUTPUT

After the main rendering loop exits, the Integrator’s `Render()` method generally calls the `Film::WriteImage()` method, which directs the film to do the processing necessary to generate the final image and store it in a file. This method takes a scale factor that is applied to the samples provided to the `AddSplat()` method. (See the end of Section 16.4.5 for further discussion of this scale factor’s use with the `MLTIntegrator`.)

```
(Film Method Definitions) +≡
void Film::WriteImage(Float splatScale) {
    (Convert image to RGB and compute final pixel values 495)
    (Write RGB image 496)
}
```

This method starts by allocating an array to store the final RGB pixel values. It then loops over all of the pixels in the image to fill in this array.

```

⟨Convert image to RGB and compute final pixel values⟩ ≡ 494
    std::unique_ptr<Float[]> rgb(new Float[3 * croppedPixelBounds.Area()]);
    int offset = 0;
    for (Point2i p : croppedPixelBounds) {
        ⟨Convert pixel XYZ color to RGB 495⟩
        ⟨Normalize pixel with weight sum 495⟩
        ⟨Add splat value at pixel 496⟩
        ⟨Scale pixel value by scale 496⟩
        ++offset;
    }
}

```

Given information about the response characteristics of the display device being used, the pixel values can be converted to device-dependent RGB values from the device-independent XYZ tristimulus values. This conversion is another change of spectral basis, where the new basis is determined by the spectral response curves of the red, green, and blue elements of the display device. Here, weights to convert from XYZ to the device RGB based on the sRGB primaries are used; sRGB is a standardized color space that is supported by virtually all 2015-era displays and printers.

```

⟨Convert pixel XYZ color to RGB⟩ ≡ 495
Pixel &pixel = GetPixel(p);
XYZToRGB(pixel.xyz, &rgb[3 * offset]);

```

As the RGB output values are being initialized, their final values from the pixel filtering equation are computed by dividing each pixel sample value by `Pixel::filterWeightSum`. This conversion can lead to RGB values where some components are negative; these are *out-of-gamut* colors that can't be represented with the chosen display primaries. Various approaches have been suggested to deal with this issue, ranging from clamping to 0, offsetting all components to lie within the gamut, or even performing a global optimization based on all of the pixels in the image. Reconstructed pixels may also end up with negative values due to negative lobes in the reconstruction filter function. Color components are clamped to 0 here to handle both of these cases.

```

⟨Normalize pixel with weight sum⟩ ≡ 495
    Float filterWeightSum = pixel.filterWeightSum;
    if (filterWeightSum != 0) {
        Float invWt = (Float)1 / filterWeightSum;
        rgb[3 * offset] = std::max((Float)0, rgb[3 * offset] * invWt);
        rgb[3 * offset+1] = std::max((Float)0, rgb[3 * offset + 1] * invWt);
        rgb[3 * offset+2] = std::max((Float)0, rgb[3 * offset + 2] * invWt);
    }
}
Pixel::xyz 486
Point2i 68
XYZToRGB() 327

```

It's also necessary to add in the contributions of splatted values for this pixel to the final value.

*(Add splat value at pixel) ≡*

```
Float splatRGB[3];
Float splatXYZ[3] = { pixel.splatXYZ[0], pixel.splatXYZ[1],
                      pixel.splatXYZ[2] };
XYZToRGB(splatXYZ, splatRGB);
rgb[3 * offset    ] += splatScale * splatRGB[0];
rgb[3 * offset + 1] += splatScale * splatRGB[1];
rgb[3 * offset + 2] += splatScale * splatRGB[2];
```

495

The final pixel value is scaled by a user-supplied factor (or by 1, if none was specified); this can be useful when writing images to 8-bit integer image formats to make the most of the limited dynamic range.

*(Scale pixel value by scale) ≡*

```
rgb[3 * offset    ] *= scale;
rgb[3 * offset + 1] *= scale;
rgb[3 * offset + 2] *= scale;
```

495

*(Film Private Data) +≡*

```
const Float scale;
```

484

The `WriteImage()` function, defined in Section A.2, handles the details of writing the image to a file. If writing to an 8-bit integer format, it applies gamma correction to the floating-point pixel values according to the sRGB standard before converting them to integers. (See the “Further Reading” section at the end of Chapter 10 for more information about gamma correction.)

*(Write RGB image) ≡*

```
::WriteImage(filename, &rgb[0], croppedPixelBounds, fullResolution);
```

494

## FURTHER READING

### Sampling Theory and Aliasing

One of the best books on signal processing, sampling, reconstruction, and the Fourier transform is Bracewell’s *The Fourier Transform and Its Applications* (2000). Glassner’s *Principles of Digital Image Synthesis* (1995) has a series of chapters on the theory and application of uniform and nonuniform sampling and reconstruction to computer graphics. For an extensive survey of the history of and techniques for interpolation of sampled data, including the sampling theorem, see Meijering (2002). Unser (2000) also surveyed recent developments in sampling and reconstruction theory including the recent move away from focusing purely on band-limited functions. For more recent work in this area, see Eldar and Michaeli (2009).

Crow (1977) first identified aliasing as a major source of artifacts in computer-generated images. Using nonuniform sampling to turn aliasing into noise was introduced by Cook (1986) and Dippé and Wold (1985); their work was based on experiments by Yellot (1983), who investigated the distribution of photoreceptors in the eyes of monkeys. Dippé and Wold also first introduced the pixel filtering equation to graphics and developed a Poisson sample pattern with a minimum distance between samples. Lee, Redner,

Film::filename 485  
 Film::fullResolution 485  
 Film::scale 496  
 Float 1062  
 Pixel::splatXYZ 486  
 WriteImage() 1068  
 XYZToRGB() 327

and Uselton (1985) developed a technique for adaptive sampling based on statistical tests that computed images to a given error tolerance. Mitchell investigated sampling patterns for ray tracing extensively. His 1987 and 1991 SIGGRAPH papers on this topic have many key insights.

Heckbert (1990a) wrote an article that explains possible pitfalls when using floating-point coordinates for pixels and develops the conventions used here.

Mitchell (1996b) investigated how much better stratified sampling patterns are than random patterns in practice. In general, the smoother the function being sampled, the more effective they are. For very quickly changing functions (e.g., pixel regions overlapped by complex geometry), sophisticated stratified patterns perform no better than unstratified random patterns. Therefore, for scenes with complex variation in the high-dimensional image function, the advantages of fancy sampling schemes compared to a simple stratified pattern are reduced.

Chiu, Shirley, and Wang (1994) suggested a *multijittered* 2D sampling technique based on randomly shuffling the  $x$  and  $y$  coordinates of a canonical jittered pattern that combines the properties of stratified and Latin hypercube approaches. More recently, Kensler (2013) showed that using the same permutation for both dimensions with their method gives much better results than independent permutations; he showed that this approach gives lower discrepancy than the Sobol' pattern while also maintaining the perceptual advantages of turning aliasing into noise due to using jittered samples.

Lagae and Dutré (2008c) surveyed the state of the art in generating Poisson disk sample patterns and compared the quality of the point sets that various algorithms generated. Of recent work in this area, see in particular the papers by Jones (2005), Dunbar and Humphreys (2006), Wei (2008), Li et al. (2010), and Ebeida et al. (2011, 2012). We note, however, the importance of Mitchell's (1991) observations that an  $n$ -dimensional Poisson disk distribution is not the ideal one for general integration problems in graphics; while it's useful for the projection of the first two dimensions on the image plane to have the Poisson-disk property, it's important that the other dimensions be more widely distributed than the Poisson-disk quality alone guarantees. Recently, Reinert et al. (2015) proposed a construction for  $n$ -dimensional Poisson disk samples that retain their characteristic sample separation under projection onto lower dimensional subsets.

`pbrt` doesn't include samplers that perform adaptive sampling—taking more samples in parts of the image with large variation. Though adaptive sampling has been an active area of research, our experience with the resulting algorithms has been that while most work well in some cases, few are robust across a wide range of scenes. Since initial work in adaptive sampling by Lee et al. (1985), Kajiya (1986), and Purgathofer (1987), a number of sophisticated and effective adaptive sampling methods have been developed in recent years. Notable work includes Hachisuka et al. (2008a), who adaptively sampled in the 5D domain of image location, time, and lens position, rather than just in image location, and introduced a novel multidimensional filtering method; Shinya (1993) and Egan et al. (2009), who developed adaptive sampling and reconstruction methods focused on rendering motion blur; and Overbeck et al. (2009), who developed adaptive sampling algorithms based on wavelets for image reconstruction. Recently, Belcour et al. (2013)

computed covariance of 5D imaging (image, time, and lens defocus) and applied adaptive sampling and high-quality reconstruction and Moon et al. (2014) have applied local regression theory to this problem.

Kirk and Arvo (1991) identified a subtle problem with adaptive sampling algorithms: in short, if a set of samples is both used to decide if more samples should be taken and is also added to the image, the end result is *biased* and doesn't converge to the correct result in the limit. Mitchell (1987) observed that standard image reconstruction techniques fail in the presence of adaptive sampling: the contribution of a dense clump of samples in part of the filter's extent may incorrectly have a large effect on the final value purely due to the number of samples taken in that region. He described a multi-stage box filter that addresses this issue.

Compressed sensing is a recent approach to sampling where the required sampling rate depends on the sparsity of the signal, not its frequency content. Sen and Darabi (2011) applied compressed sensing to rendering, allowing them to generate high-quality images at very low sampling rates.

### Low-Discrepancy Sampling

Shirley (1991) first introduced the use of discrepancy to evaluate the quality of sample patterns in computer graphics. This work was built upon by Mitchell (1992), Dobkin and Mitchell (1993), and Dobkin, Eppstein, and Mitchell (1996). One important observation in Dobkin et al.'s paper is that the box discrepancy measure used in this chapter and in other work that applies discrepancy to pixel sampling patterns isn't particularly appropriate for measuring a sampling pattern's accuracy at randomly oriented edges through a pixel and that a discrepancy measure based on random edges should be used instead. This observation explains why some theoretically good low-discrepancy patterns do not perform as well as expected when used for image sampling.

Mitchell's first paper on discrepancy introduced the idea of using deterministic low-discrepancy sequences for sampling, removing all randomness in the interest of lower discrepancy (Mitchell 1992). Such *quasi-random* sequences are the basis of quasi-Monte Carlo methods, which will be described in Chapter 13. The seminal book on quasi-random sampling and algorithms for generating low-discrepancy patterns was written by Niederreiter (1992). For a more recent treatment, see Dick and Pillichshammer's excellent book (2010).

Faure (1992) described a deterministic approach for computing permutations for scrambled radical inverses. The implementation of the `ComputeRadicalInversePermutations()` function in this chapter uses random permutations, which are simpler to implement and work nearly as well in practice. The algorithms used for computing sample indices within given pixels in Sections 7.4 and 7.7 were introduced by Grünschloß et al. (2012).

Keller and collaborators have investigated quasi-random sampling patterns for a variety of applications in graphics (Keller 1996, 1997, 2001). The  $(0, 2)$ -sequence sampling techniques used in the `ZeroTwoSequenceSampler` are based on a paper by Kollig and Keller (2002).  $(0, 2)$ -sequences are one instance of a general type of low-discrepancy sequence known as  $(t, s)$ -sequences and  $(t, m, s)$ -nets. These are discussed further by Niederreiter (1992) and Dick and Pillichshammer (2010). Some of Kollig and Keller's techniques are based on algorithms developed by Friedel and Keller (2000). Keller (2001, 2006) argued

`ComputeRadicalInverse  
Permutations()`  
**449**

`ZeroTwoSequenceSampler` **462**

that because low-discrepancy patterns tend to converge more quickly than others, they are the most efficient sampling approach for generating high-quality imagery.

The MaxMinDistSampler in Section 7.6 is based on generator matrices found by Grünschloß and collaborators (2008, 2009). Sobol' (1967) introduced the family of generator matrices used in Section 7.7; Wächter's Ph.D. dissertation discusses high-performance implementation of base-2 generator matrix operations (Wächter 2008). The Sobol' generator matrices our implementation uses are improved versions derived by Joe and Kuo (2008).

### Filtering and Reconstruction

Cook (1986) first introduced the Gaussian filter to graphics. Mitchell and Netravali (1988) investigated a family of filters using experiments with human observers to find the most effective ones; the MitchellFilter in this chapter is the one they chose as the best. Kajiya and Ullner (1981) investigated image filtering methods that account for the effect of the reconstruction characteristics of Gaussian falloff from pixels in CRTs, and, more recently, Betrisey et al. (2000) described Microsoft's ClearType technology for display of text on LCDs. Alim (2013) has recently applied reconstruction techniques that attempt to minimize the error between the reconstructed image and the original continuous image, even in the presence of discontinuities.

There has been quite a bit of research into reconstruction filters for image resampling applications. Although this application is not the same as reconstructing nonuniform samples for image synthesis, much of this experience is applicable. Turkowski (1990a) reported that the Lanczos windowed sinc filter gives the best results of a number of filters for image resampling. Meijering et al. (1999) tested a variety of filters for image resampling by applying a series of transformations to images such that if perfect resampling had been done the final image would be the same as the original. They also found that the Lanczos window performed well (as did a few others) and that truncating the sinc without a window gave some of the worst results. Other work in this area includes papers by Möller et al. (1997) and Machiraju and Yagel (1996).

Even with a fixed sampling rate, clever reconstruction algorithms can be useful to improve image quality. See, for example, Reshetov (2009), who used image gradients to find edges across multiple pixels to estimate pixel coverage for antialiasing and Guertin et al. (2014), who developed a filtering approach for motion blur.

Lee and Redner (1990) first suggested using a median filter, where the median of a set of samples is used to find each pixel's value, as a noise reduction technique. More recently, Lehtinen et al. (2011, 2012), Kalantari and Sen (2013), Rousselle et al. (2012, 2013), Delbracio et al. (2014), Munkberg et al. (2014), and Bauszat et al. (2015) have developed filtering techniques to reduce noise in images rendered using Monte Carlo algorithms. Kalantari et al. (2015) applied machine learning to the problem of finding effective denoising filters and demonstrated impressive results.

Jensen and Christensen (1995) observed that it can be more effective to separate out the contributions to pixel values based on the type of illumination they represent; low-frequency indirect illumination can be filtered differently from high-frequency direct illumination, thus reducing noise in the final image. They developed an effective filtering technique based on this observation. An improvement to this approach was developed by

Keller and collaborators with the *discontinuity buffer* (Keller 1998; Wald et al. 2002). In addition to filtering slowly varying quantities like indirect illumination separately from more quickly varying quantities like surface reflectance, the discontinuity buffer uses geometric quantities like the surface normal at nearby pixels to determine whether their corresponding values can be reasonably included at the current pixel. Kontkanen et al. (2004) built on these approaches to build a filtering approach for indirect illumination when using the irradiance caching algorithm.

Lessig et al. (2014) proposed a general framework for constructing quadrature rules tailored to specific integration problems such as stochastic ray tracing, spherical harmonics projection, and scattering by surfaces. When targeting band-limited functions, their approach subsumes the frequency-space approach presented in this chapter. An excellent tutorial about the underlying theory of *reproducing kernel bases* is provided in the article's supplemental material.

### Perceptual Issues

A number of different approaches have been developed for mapping out-of-gamut colors to the displayable range; see Rougeron and Péroche's survey article for discussion of this issue and references to various approaches (Rougeron and Péroche 1998). This topic was also covered by Hall (1989).

Tone reproduction—algorithms for displaying high-dynamic-range images on low-dynamic-range display devices—became an active area of research starting with the work of Tumblin and Rushmeier (1993). The survey article of Devlin et al. (2002) summarizes most of the work in this area through 2002, giving pointers to the original papers. See Reinhard et al.'s book (2010) on high dynamic range imaging, which includes comprehensive coverage of this topic through 2010. More recently, Reinhard et al. (2012) have developed tone reproduction algorithms that consider both accurate brightness and color reproduction together, also accounting for the display and viewing environment.

The human visual system generally causes the brain to perceive that surfaces have the color of the underlying surface, regardless of the illumination spectrum—for example, white paper is perceived to be white, even under the yellow-ish illumination of an incandescent lightbulb. A number of methods have been developed to process photographs to perform *white balancing* to eliminate the tinge of light source colors; see Gijsenij et al. (2011) for a survey. White balancing is challenging, since the only information available to white balancing algorithms is the final pixel values. In a renderer, the problem is easier, as information is available directly about the light sources and the surface reflection properties; Wilkie and Weidlich (2009) developed an efficient method to perform accurate white balancing in a renderer with limited computational overhead.

For background information on properties of the human visual system, Wandell's book on vision is an excellent starting point (Wandell 1995). Ferwerda (2001) presented an overview of the human visual system for applications in graphics, and Malacara (2002) gave a concise overview of color theory and basic properties of how the human visual system processes color.

## EXERCISES

- ② 7.1 It's possible to implement a specialized version of `ScrambledRadicalInverse()` for base 2, along the lines of the implementation in `RadicalInverse()`. Determine how to map the random digit permutation to a single bitwise operation and implement this approach. Compare the values computed to those generated by the current implementation to ensure your method is correct and measure how much faster yours is by writing a small benchmark program.
- ② 7.2 Currently, the third through fifth dimensions of each sample vector are consumed for time and lens samples, even though not all scenes need these sample values. Because lower dimensions in the sample vector are often better distributed than later ones, this can cause an unnecessary reduction in image quality.  
Modify pbrt so that the camera can report its sample requirements and then use this information when samples are requested to initialize `CameraSamples`. Don't forget to update the value of `GlobalSampler::arrayStartDim`. Render images with the `DirectLightingIntegrator` and compare results to the current implementation. Do you see an improvement? How do results differ with different samplers? How do you explain any differences you see across samplers?
- ② 7.3 Implement the improved multi-jittered sampling method introduced by Kensler (2013) as a new Sampler in pbrt. Compare image quality and rendering time to rendering with the `StratifiedSampler`, the `HaltonSampler`, and the `SobolSampler`.
- ② 7.4 Keller (2004) and Dammertz and Keller (2008b) described the application of *rank-1 lattices* to image synthesis. Rank-1 lattices are another way of efficiently generating high-quality low-discrepancy sequences of sample points. Read their papers and implement a Sampler based on this approach. Compare results to the other samplers in pbrt.

- ② 7.5 With pbrt's current `FilmTile` implementation, the pixel values in an image may change by small amounts if an image is rerendered, due to threads finishing tiles in different orders over subsequent runs. For example, a pixel that had a final value that came from samples from three different image sampling tiles,  $v_1 + v_2 + v_3$ , may sometimes have its value computed as  $(v_1 + v_2) + v_3$  and sometimes as  $v_1 + (v_2 + v_3)$ . Due to floating-point round-off, these two values may be different. While these differences aren't normally a problem, they wreak havoc with automated testing scripts that might want to verify that a believed-to-be-innocuous change to the system didn't actually cause any differences in rendered images.

`CameraSample` 357  
`DirectLightingIntegrator` 851  
`Film::MergeFilmTile()` 493  
`FilmTile` 489  
`GlobalSampler::arrayStartDim` 431  
`HaltonSampler` 450  
`RadicalInverse()` 444  
`ScrambledRadicalInverse()` 450  
`SobolSampler` 468  
`StratifiedSampler` 434

Modify `Film::MergeFilmTile()` so that it merges tiles in a consistent order so that final pixel values don't suffer from this inconsistency. (For example, your implementation might buffer up `FilmTiles` and only merge a tile when all neighboring tiles above and to its left have already been merged.) Ensure that

your implementation doesn't introduce any meaningful performance regression. Measure the additional memory usage due to longer lived `FilmTiles`; how does it relate to total memory usage?

- ② 7.6 As mentioned in Section 7.9, the `Film::AddSplat()` method doesn't use a filter function but instead just splats the sample to the single pixel it's closest to, effectively using a box filter. In order to apply an arbitrary filter, the filter must be normalized so that it integrates to one over its domain; this constraint isn't currently required of `Filters` by `pbrt`. Modify the computation of `filterTable` in the `Film` constructor so that the tabulated function is normalized. (Don't forget that the table only stores one-quarter of the function's extent when computing the normalization factor.) Then modify the implementation of the `AddSplat()` method to use this filter. Investigate the execution time and image quality differences that result.
- ① 7.7 Modify `pbrt` to create images where the value stored in the `Film` for each camera ray is proportional to the time taken to compute the ray's radiance. (A 1-pixel-wide box filter is probably the most useful filter for this exercise.) Render images of a variety of scenes with this technique. What insight about the system's performance do the resulting images bring? You may need to scale pixel values or take their logarithm to see meaningful variation when you view them.
- ② 7.8 One of the advantages of the linearity assumption in radiometry is that the final image of a scene is the same as the sum of individual images that account for each light source's contribution (assuming a floating-point image file format is used that doesn't clip pixel radiance values). An implication of this property is that if a renderer creates a separate image for each light source, it is possible to write interactive lighting design tools that make it possible to quickly see the effects of scaling the contributions of individual lights in the scene without needing to rerender it from scratch. Instead, a light's individual image can be scaled and the final image regenerated by summing all of the light images again. (This technique was first applied for opera lighting design by Dorsey, Sillion, and Greenberg (1991).) Modify `pbrt` to output a separate image for each of the lights in the scene, and write an interactive lighting design tool that uses them in this manner.
- ③ 7.9 Mitchell and Netravali (1988) noted that there is a family of reconstruction filters that use both the value of a function and its derivative at the point to do substantially better reconstruction than if just the value of the function is known. Furthermore, they report that they have derived closed-form expressions for the screen space derivatives of Lambertian and Phong reflection models, although they do not include these expressions in their paper. Investigate derivative-based reconstruction, and extend `pbrt` to support this technique. Because it will likely be difficult to derive expressions for the screen space derivatives for general shapes and BSDF models, investigate approximations based on finite differencing. Techniques built on the ideas behind the ray differentials of Section 10.1 may be fruitful for this effort.

`Film` 484

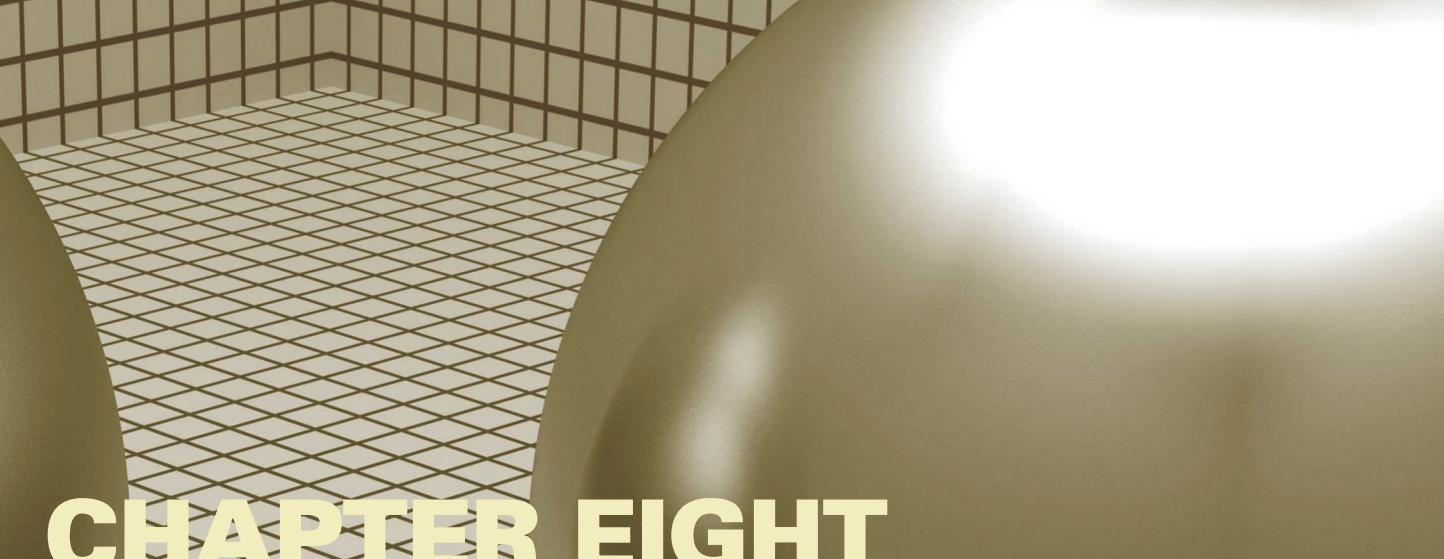
`Film::AddSplat()` 494

`Filter` 474

- ③ 7.10 Image-based rendering is the general name for a set of techniques that use one or more images of a scene to synthesize new images from viewpoints different from the original ones. One such approach is light field rendering, where a set of images from a densely spaced set of positions is used (Levoy and Hanrahan 1996; Gortler et al. 1996). Read these two papers on light fields, and modify pbrt to directly generate light fields of scenes, without requiring that the renderer be run multiple times, once for each camera position. It will probably be necessary to write a specialized Camera, Sampler, and Film to do this. Also, write an interactive light field viewer that loads light fields generated by your implementation and generates new views of the scene.
- ③ 7.11 Rather than just storing spectral values in an image, it's often useful to store additional information about the objects in the scene that were visible at each pixel. See, for example, the SIGGRAPH papers by Perlin (1985a) and Saito and Takahashi (1990). For example, if the 3D position, surface normal, and BRDF of the object at each pixel are stored, then the scene can be efficiently rerendered after moving the light sources (Gershbein and Hanrahan 2000). Alternatively, if each sample stores information about all of the objects visible along its camera ray, rather than just the first one, new images from shifted viewpoints can be rerendered (Shade et al. 1998). Investigate representations for deep frame buffers and algorithms that use them; extend pbrt to support the creation of images like these, and develop tools that operate on them.
- ② 7.12 Implement a median filter for image reconstruction: for each pixel, store the median of all of the samples within a filter extent around it. This task is complicated by the fact that filters in the current Film implementation must be *linear*—the value of the filter function is determined solely by the position of the sample with respect to the pixel position, and the value of the sample has no impact on the value of the filter function. Because the implementation assumes that filters are linear, and because it doesn't store sample values after adding their contribution to the image, implementing the median filter will require generalizing the Film or developing a new Film implementation.
- Render images using integrators like the PathIntegrator that have objectionable image noise with regular image filters. How successful is the median filter at reducing noise? Are there visual shortcomings to using the median filter? Can you implement this approach without needing to store all of the image sample values before computing final pixel values?
- ② 7.13 An alternative to the median filter is to discard the sample with the lowest contribution and the sample with the largest contribution in a pixel's filter region. This approach uses more of the information gathered during sampling. Implement this approach and compare the results to the median filter.
- ③ 7.14 Implement the discontinuity buffer, as described by Keller and collaborators (Keller 1998; Wald et al. 2002). You will probably need to modify the interface to the Integrators so that they can separately return direct and indirect illumination contributions and then pass these separately to the Film. Render images showing its effectiveness when rendering images with indirect illumination.

- ③ 7.15 Implement one of the recent adaptive sampling and reconstruction techniques such as the ones described by Hachisuka et al. (2008a), Egan et al. (2009), Overbeck et al. (2009), or Moon et al. (2014). How much more efficiently do they generate images of equal quality than just uniformly sampling at a high rate? How do they affect running time for simple scenes where adaptive sampling isn't needed?
- ③ 7.16 Investigate current research in tone reproduction algorithms (see, e.g., Reinhard et al. 2010; 2012), and implement one or more of these algorithms. Use your implementation with a number of scenes rendered by pbrt, and discuss the improvements you see compared to viewing the images without tone reproduction.

This page intentionally left blank



# CHAPTER EIGHT

## REFLECTION MODELS

This chapter defines a set of classes for describing the way that light scatters at surfaces. Recall that in Section 5.6.1 we introduced the bidirectional reflectance distribution function (BRDF) abstraction to describe light reflection at a surface, the BTDF to describe transmission at a surface, and the BSDF to encompass both of these effects. In this chapter, we will start by defining a generic interface to these surface reflection and transmission functions.

Scattering from many surfaces is often best described as a spatially varying mixture of multiple BRDFs and BTDFs; in Chapter 9, we will introduce a BSDF object that combines multiple BRDFs and BTDFs to represent overall scattering from the surface. The current chapter sidesteps the issue of reflection and transmission properties that vary over the surface; the texture classes of Chapter 10 will address that problem. BRDFs and BTDFs explicitly only model scattering from light that enters and exits a surface at a single point. For surfaces that exhibit meaningful subsurface light transport, we will introduce the BSSRDF class, which models subsurface scattering, in Section 11.4 after some of the related theory is introduced in Chapter 11.

Surface reflection models come from a number of sources:

- *Measured data*: Reflection distribution properties of many real-world surfaces have been measured in laboratories. Such data may be used directly in tabular form or to compute coefficients for a set of basis functions.
- *Phenomenological models*: Equations that attempt to describe the qualitative properties of real-world surfaces can be remarkably effective at mimicking them. These types of BSDFs can be particularly easy to use, since they tend to have intuitive parameters that modify their behavior (e.g., “roughness”).
- *Simulation*: Sometimes, low-level information is known about the composition of a surface. For example, we might know that a paint is comprised of colored particles of some average size suspended in a medium or that a particular fabric is comprised

of two types of threads, each with known reflectance properties. In these cases, light scattering from the microgeometry can be simulated to generate reflection data. This simulation can be done either during rendering or as a preprocess, after which it may be fit to a set of basis functions for use during rendering.

- *Physical (wave) optics:* Some reflection models have been derived using a detailed model of light, treating it as a wave and computing the solution to Maxwell’s equations to find how it scatters from a surface with known properties. These models tend to be computationally expensive, however, and usually aren’t appreciably more accurate than models based on geometric optics are for rendering applications.
- *Geometric optics:* As with simulation approaches, if the surface’s low-level scattering and geometric properties are known, then closed-form reflection models can sometimes be derived directly from these descriptions. Geometric optics makes modeling light’s interaction with the surface more tractable, since complex wave effects like polarization can be ignored.

The “Further Reading” section at the end of this chapter gives pointers to a variety of such reflection models.

Before we define the relevant interfaces, a brief review of how they fit into the overall system is in order. If a `SamplerIntegrator` is used, the `SamplerIntegrator::Li()` method implementation is called for each ray. After finding the closest intersection with a geometric primitive, it calls the surface shader that is associated with the primitive. The surface shader is implemented as a method of `Material` subclasses and is responsible for deciding what the BSDF is at a particular point on the surface; it returns a `BSDF` object that holds BRDFs and BTDFs that it has allocated and initialized to represent scattering at that point. The integrator then uses the BSDF to compute the scattered light at the point, based on the incoming illumination at the point. (The process where a `BDPTIntegrator`, `MLTIntegrator`, or `SPPMIntegrator` is used rather than a `SamplerIntegrator` is broadly similar.)

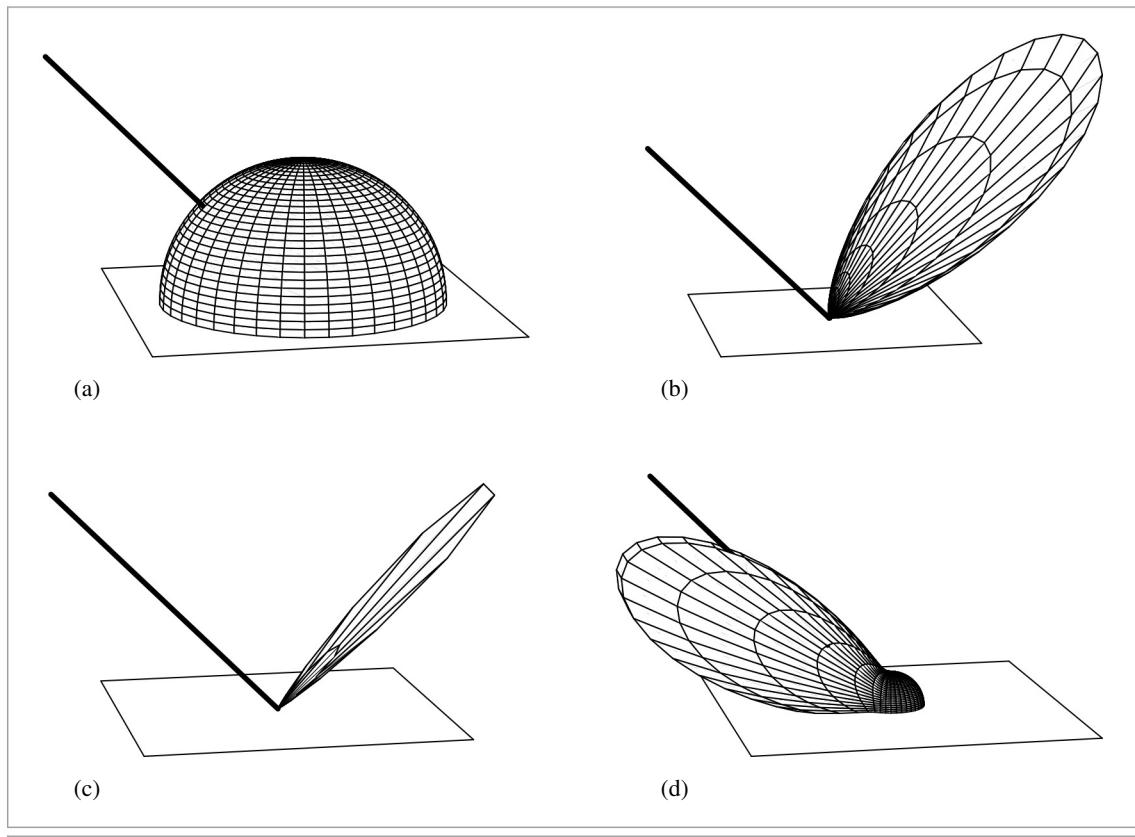
## Basic Terminology

In order to be able to compare the visual appearance of different reflection models, we will introduce some basic terminology for describing reflection from surfaces.

Reflection from surfaces can be split into four broad categories: *diffuse*, *glossy specular*, *perfect specular*, and *retro-reflective* (Figure 8.1). Most real surfaces exhibit reflection that is a mixture of these four types. Diffuse surfaces scatter light equally in all directions. Although a perfectly diffuse surface isn’t physically realizable, examples of near-diffuse surfaces include dull chalkboards and matte paint. Glossy specular surfaces such as plastic or high-gloss paint scatter light preferentially in a set of reflected directions—they show blurry reflections of other objects. Perfect specular surfaces scatter incident light in a single outgoing direction. Mirrors and glass are examples of perfect specular surfaces. Finally, retro-reflective surfaces like velvet or the Earth’s moon scatter light primarily back along the incident direction. Images throughout this chapter will show the differences between these various types of reflection when used in rendered scenes.

Given a particular category of reflection, the reflectance distribution function may be *isotropic* or *anisotropic*. Most objects are isotropic: if you choose a point on the surface

<code>BDPTIntegrator</code>	992
<code>BSDF</code>	572
<code>Material</code>	577
<code>MLTIntegrator</code>	1035
<code>SamplerIntegrator</code>	25
<code>SamplerIntegrator::Li()</code>	31
<code>SPPMIntegrator</code>	973



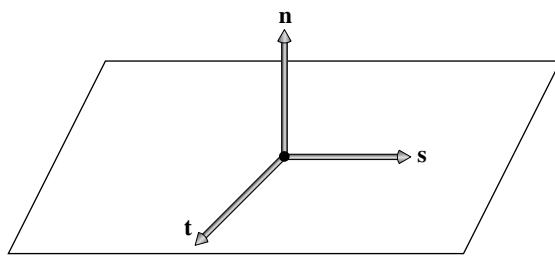
**Figure 8.1:** Reflection from a surface can be generally categorized by the distribution of reflected light from an incident direction (heavy lines): (a) diffuse, (b) glossy specular, (c) perfect specular, and (d) retro-reflective distributions.

and rotate it around its normal axis at that point, the distribution of light reflected doesn't change. In contrast, anisotropic materials reflect different amounts of light as you rotate them in this way. Examples of anisotropic surfaces include brushed metal, many types of cloth, and compact disks.

### Geometric Setting

Reflection computations in pbrt are evaluated in a reflection coordinate system where the two tangent vectors and the normal vector at the point being shaded are aligned with the  $x$ ,  $y$ , and  $z$  axes, respectively (Figure 8.2). All direction vectors passed to and returned from the BRDF and BTDF routines will be defined with respect to this coordinate system. It is important to understand this coordinate system in order to understand the BRDF and BTDF implementations in this chapter.

The shading coordinate system also gives a frame for expressing directions in spherical coordinates  $(\theta, \phi)$ ; the angle  $\theta$  is measured from the given direction to the  $z$  axis, and  $\phi$  is the angle formed with the  $x$  axis after projection of the direction onto the  $xy$  plane.



**Figure 8.2: The Basic BSDF Interface Setting.** The shading coordinate system is defined by the orthonormal basis vectors ( $s, t, n$ ). We will orient these vectors such that they lie along the  $x, y$ , and  $z$  axes in this coordinate system. Direction vectors  $\omega$  in world space are transformed into the shading coordinate system before any of the BRDF or BTDF methods are called.

Given a direction vector  $\omega$  in this coordinate system, it is easy to compute quantities like the cosine of the angle that it forms with the normal direction:

$$\cos \theta = (\mathbf{n} \cdot \omega) = ((0, 0, 1) \cdot \omega) = \omega_z.$$

We will provide utility functions to compute this value and some useful variations; their use helps clarify BRDF and BTDF implementations.

*(BSDF Inline Functions)*  $\equiv$

```
inline Float CosTheta(const Vector3f &w) { return w.z; }
inline Float Cos2Theta(const Vector3f &w) { return w.z * w.z; }
inline Float AbsCosTheta(const Vector3f &w) { return std::abs(w.z); }
```

The value of  $\sin^2 \theta$  can be computed using the trigonometric identity  $\sin^2 \theta + \cos^2 \theta = 1$ , though we need to be careful to avoid taking the square root of a negative number in the rare case that  $1 - \text{Cos2Theta}(w)$  is less than zero due to floating-point round-off error.

*(BSDF Inline Functions)*  $+ \equiv$

```
inline Float Sin2Theta(const Vector3f &w) {
    return std::max((Float)0, (Float)1 - Cos2Theta(w));
}
inline Float SinTheta(const Vector3f &w) {
    return std::sqrt(Sin2Theta(w));
}
```

The tangent of the angle  $\theta$  can be computed via the identity  $\tan \theta = \sin \theta / \cos \theta$ .

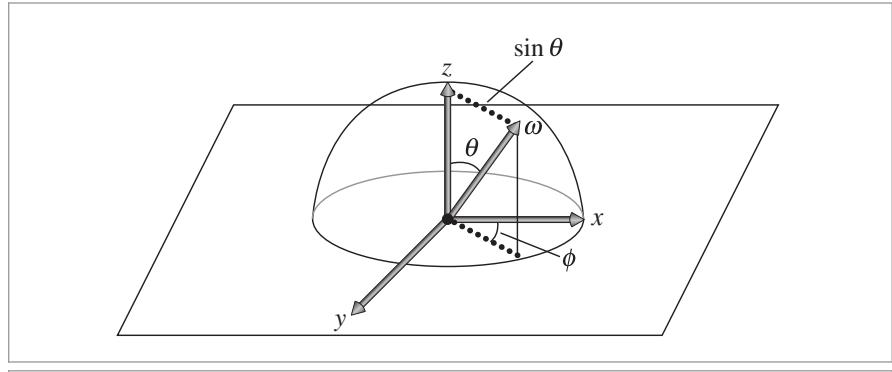
*(BSDF Inline Functions)*  $+ \equiv$

```
inline Float TanTheta(const Vector3f &w) {
    return SinTheta(w) / CosTheta(w);
}
inline Float Tan2Theta(const Vector3f &w) {
    return Sin2Theta(w) / Cos2Theta(w);
}
```

Cos2Theta() 510

Float 1062

Vector3f 60



**Figure 8.3:** The values of  $\sin \phi$  and  $\cos \phi$  can be computed using the circular coordinate equations  $x = r \cos \phi$  and  $y = r \sin \phi$ , where  $r$ , the length of the dashed line, is equal to  $\sin \theta$ .

We can similarly use the shading coordinate system to simplify the calculations for the sine and cosine of the  $\phi$  angle (Figure 8.3). In the plane of the point being shaded, the vector  $\omega$  has coordinates  $(x, y)$ , which are given by  $r \cos \phi$  and  $r \sin \phi$ , respectively. The radius  $r$  is  $\sin \theta$ , so

$$\begin{aligned}\cos \phi &= \frac{x}{r} = \frac{x}{\sin \theta} \\ \sin \phi &= \frac{y}{r} = \frac{y}{\sin \theta}.\end{aligned}$$

```
{BSDF Inline Functions} +≡
inline Float CosPhi(const Vector3f &w) {
    Float sinTheta = SinTheta(w);
    return (sinTheta == 0) ? 1 : Clamp(w.x / sinTheta, -1, 1);
}
inline Float SinPhi(const Vector3f &w) {
    Float sinTheta = SinTheta(w);
    return (sinTheta == 0) ? 0 : Clamp(w.y / sinTheta, -1, 1);
}
```

```
{BSDF Inline Functions} +≡
inline Float Cos2Phi(const Vector3f &w) {
    return CosPhi(w) * CosPhi(w);
}
inline Float Sin2Phi(const Vector3f &w) {
    return SinPhi(w) * SinPhi(w);
}

Clamp() 1062
Float 1062
SinTheta() 510
Vector3f 60
```

The cosine of the angle  $\Delta\phi$  between two vectors in the shading coordinate system can be found by zeroing the  $z$  coordinate of the two vectors to get 2D vectors and then normalizing them. The dot product of these two vectors gives the cosine of the angle

between them. The implementation below rearranges the terms a bit for efficiency so that only a single square root operation needs to be performed.

*(BSDF Inline Functions) +≡*

```
inline Float CosDPhi(const Vector3f &wa, const Vector3f &wb) {
    return Clamp((wa.x * wb.x + wa.y * wb.y) /
        std::sqrt((wa.x * wa.x + wa.y * wa.y) *
        (wb.x * wb.x + wb.y * wb.y)), -1, 1);
}
```

There are important conventions and implementation details to keep in mind when reading the code in this chapter and when adding BRDFs and BTDFs to `pbrt`:

- The incident light direction  $\omega_i$  and the outgoing viewing direction  $\omega_o$  will both be normalized and outward facing after being transformed into the local coordinate system at the surface.
- By convention in `pbrt`, the surface normal  $n$  always points to the “outside” of the object, which makes it easy to determine if light is entering or exiting transmissive objects: if the incident light direction  $\omega_i$  is in the same hemisphere as  $n$ , then light is entering; otherwise, it is exiting. Therefore, one detail to keep in mind is that the normal may be on the opposite side of the surface than one or both of the  $\omega_i$  and  $\omega_o$  direction vectors. Unlike many other renderers, `pbrt` does not flip the normal to lie on the same side as  $\omega_o$ .
- The local coordinate system used for shading may not be exactly the same as the coordinate system returned by the `Shape::Intersect()` routines from Chapter 3; they can be modified between intersection and shading to achieve effects like bump mapping. See Chapter 9 for examples of this kind of modification.
- Finally, BRDF and BTDF implementations should not concern themselves with whether  $\omega_i$  and  $\omega_o$  lie in the same hemisphere. For example, although a reflective BRDF should in principle detect if the incident direction is above the surface and the outgoing direction is below and always return no reflection in this case, here we will expect the reflection function to instead compute and return the amount of light reflected using the appropriate formulas for their reflection model, ignoring the detail that they are not in the same hemisphere. Higher level code in `pbrt` will ensure that only reflective or transmissive scattering routines are evaluated as appropriate. The value of this convention will be explained in Section 9.1.

## 8.1 BASIC INTERFACE

We will first define the interface for the individual BRDF and BTDF functions. BRDFs and BTDFs share a common base class, `BxDF`. Because both have the exact same interface, sharing the same base class reduces repeated code and allows some parts of the system to work with `BxDFs` generically without distinguishing between BRDFs and BTDFs.

`BxDF` 513

`Clamp()` 1062

`Float` 1062

`Shape::Intersect()` 129

`Vector3f` 60

```
{BxDF Declarations} ≡
    class BxDF {
        public:
            (BxDF Interface 513)
            (BxDF Public Data 513)
    };
```

The BSDF class, which will be introduced in Section 9.1, holds a collection of BxDF objects that together describe the scattering at a point on a surface. Although we are hiding the implementation details of the BxDF behind a common interface for reflective and transmissive materials, some of the light transport algorithms in Chapters 14 through 16 will need to distinguish between these two types. Therefore, all BxDFs have a BxDF::type member that holds flags from BxDFType. For each BxDF, the flags should have at least one of BSDF\_REFLECTION or BSDF\_TRANSMISSION set and exactly one of the diffuse, glossy, and specular flags. Note that there is no retro-reflective flag; retro-reflection is treated as glossy reflection in this categorization.

```
{BSDF Declarations} ≡
enum BxDFType {
    BSDF_REFLECTION = 1 << 0,
    BSDF_TRANSMISSION = 1 << 1,
    BSDF_DIFFUSE = 1 << 2,
    BSDF_GLOSSY = 1 << 3,
    BSDF_SPECULAR = 1 << 4,
    BSDF_ALL = BSDF_DIFFUSE | BSDF_GLOSSY | BSDF_SPECULAR |
                BSDF_REFLECTION | BSDF_TRANSMISSION,
};
```

```
{BxDF Interface} ≡
BxDF(BxDFType type) : type(type) {}
```

```
{BxDF Public Data} ≡
const BxDFType type;
```

513

513

The `MatchesFlags()` utility method determines if the BxDF matches the user-supplied type flags:

```
{BxDF Interface} +≡
bool MatchesFlags(BxDFType t) const {
    return (type & t) == type;
}
```

513

The key method that BxDFs provide is `BxDF::f()`. It returns the value of the distribution function for the given pair of directions. This interface implicitly assumes that light in different wavelengths is decoupled—energy at one wavelength will not be reflected at a different wavelength. By making this assumption, the effect of the reflection function can be represented directly with a `Spectrum`. Supporting fluorescent materials where this assumption is not true would require that this method return an  $n \times n$  matrix that

`BSDF` 572  
`BSDF_DIFFUSE` 513  
`BSDF_GLOSSY` 513  
`BSDF_REFLECTION` 513  
`BSDF_SPECULAR` 513  
`BSDF_TRANSMISSION` 513  
`BxDF` 513  
`BxDF::f()` 514  
`BxDFType` 513  
`BxDFType` 513  
`Spectrum` 315

encoded the transfer of energy between spectral samples (where  $n$  is the number of samples in the Spectrum representation).

*(BxDF Interface) +≡* 513  
`virtual Spectrum f(const Vector3f &wo, const Vector3f &wi) const = 0;`

Not all BxDFs can be evaluated with the `f()` method. For example, perfectly specular objects like a mirror, glass, or water only scatter light from a single incident direction into a single outgoing direction. Such BxDFs are best described with delta distributions that are zero except for the single direction where light is scattered. These BxDFs need special handling in pbrt, so we will also provide the method `BxDF::Sample_f()`. This method is used both for handling scattering that is described by delta distributions as well as for randomly sampling directions from BxDFs that scatter light along multiple directions; this second application will be explained in the discussion of Monte Carlo BSDF sampling in Section 14.1.

`BxDF::Sample_f()` computes the direction of incident light  $\omega_i$  given an outgoing direction  $\omega_o$  and returns the value of the BxDF for the pair of directions. For delta distributions, it is necessary for the BxDF to choose the incident light direction in this way, since the caller has no chance of generating the appropriate  $\omega_i$  direction.<sup>1</sup> The `sample` and `pdf` parameters aren't needed for delta distribution BxDFs, so they will be explained later, in Section 14.1, when we provide implementations of this method for nonspecular reflection functions.

*(BxDF Interface) +≡* 513  
`virtual Spectrum Sample_f(const Vector3f &wo, Vector3f *wi,  
 const Point2f &sample, Float *pdf,  
 BxDFType *sampledType = nullptr) const;`

### 8.1.1 REFLECTANCE

It can be useful to take the aggregate behavior of the 4D BRDF or BTDF, defined as a function over pairs of directions, and reduce it to a 2D function over a single direction, or even to a constant value that describes its overall scattering behavior.

The *hemispherical-directional reflectance* is a 2D function that gives the total reflection in a given direction due to constant illumination over the hemisphere, or, equivalently, total reflection over the hemisphere due to light from a given direction.<sup>2</sup> It is defined as

$$\rho_{hd}(\omega_o) = \int_{\mathcal{H}^2(\mathbf{n})} f_r(p, \omega_o, \omega_i) |\cos \theta_i| d\omega_i. \quad (8.1)$$

The `BxDF::rho()` method computes the reflectance function  $\rho_{hd}$ . Some BxDFs can compute this value in closed form, although most use Monte Carlo integration to compute an approximation to it. For those BxDFs, the `nSamples` and `samples` parameters are used by the implementation of the Monte Carlo algorithm; they are explained in Section 14.1.5.

<sup>1</sup> Delta distributions in reflection functions have some additional subtle implications for light transport algorithms. Sections 14.1.3 and 14.4.5 describe the issues in detail.

<sup>2</sup> The fact that these two quantities are equal is due to the reciprocity of reflection functions. BTDFs are generally not reciprocal; see Section 16.1.3.

BxDF [513](#)  
`BxDF::rho()` [515](#)  
`BxDF::Sample_f()` [806](#)  
`BxDFType` [513](#)  
`Float` [1062](#)  
`Point2f` [68](#)  
`Spectrum` [315](#)  
`Vector3f` [60](#)

*(BxDF Interface)* +≡

```
virtual Spectrum rho(const Vector3f &wo, int nSamples,
                     const Point2f *samples) const;
```

The *hemispherical-hemispherical reflectance* of a surface, denoted by  $\rho_{hh}$ , is a spectral value that gives the fraction of incident light reflected by a surface when the incident light is the same from all directions. It is

$$\rho_{hh} = \frac{1}{\pi} \int_{\mathcal{H}^2(n)} \int_{\mathcal{H}^2(n)} f_r(p, \omega_o, \omega_i) |\cos \theta_o \cos \theta_i| d\omega_o d\omega_i.$$

The `BxDF::rho()` method computes  $\rho_{hh}$  if no direction  $\omega_o$  is provided. The remaining parameters are again used when computing a Monte Carlo estimate of the value of  $\rho_{hh}$ , if needed.

*(BxDF Interface)* +≡

```
virtual Spectrum rho(int nSamples, const Point2f *samples1,
                     const Point2f *samples2) const;
```

### 8.1.2 BxDF SCALING ADAPTER

It is also useful to take a given BxDF and scale its contribution with a `Spectrum` value. The `ScaledBxDF` wrapper holds a `BxDF *` and a `Spectrum` and implements this functionality. This class is used by the `MixMaterial` (defined in Section 9.2.3), which creates BSDFs based on a weighted combination of two other materials.

*(BxDF Declarations)* +≡

```
class ScaledBxDF : public BxDF {
public:
    (ScaledBxDF Public Methods 515)
private:
    BxDF *bxdf;
    Spectrum scale;
};
```

*(ScaledBxDF Public Methods)* ≡

```
ScaledBxDF(BxDF *bxdf, const Spectrum &scale)
    : BxDF(BxDFTyp(bxdf->type)), bxd(bxdf), scale(scale) { }
```

The implementations of the `ScaledBxDF` methods are straightforward; we'll only include `f()` here.

*(BxDF Method Definitions)* ≡

```
Spectrum ScaledBxDF::f(const Vector3f &wo, const Vector3f &wi) const {
    return scale * bxd->f(wo, wi);
}
```

BSDF 572  
 BxDF 513  
 BxDF::f() 514  
 BxDFTyp 513  
 MixMaterial 582  
 Point2f 68  
 ScaledBxDF 515  
 ScaledBxDF::bxdf 515  
 ScaledBxDF::scale 515  
 Spectrum 315  
 Vector3f 60

## 8.2 SPECULAR REFLECTION AND TRANSMISSION

The behavior of light at perfectly smooth surfaces is relatively easy to characterize analytically using both the physical and geometric optics models. These surfaces exhibit perfect specular reflection and transmission of incident light; for a given  $\omega_i$  direction, all light is scattered in a single outgoing direction  $\omega_o$ . For specular reflection, this direction is the outgoing direction that makes the same angle with the normal as the incoming direction:

$$\theta_i = \theta_o,$$

and where  $\phi_o = \phi_i + \pi$ . For transmission, we again have  $\phi_o = \phi_i + \pi$ , and the outgoing direction  $\theta_t$  is given by *Snell's law*, which relates the angle  $\theta_t$  between the transmitted direction and the surface normal  $\mathbf{n}$  to the angle  $\theta_i$  between the incident ray and the surface normal  $\mathbf{n}$ . (One of the exercises at the end of this chapter is to derive Snell's law using Fermat's principle from optics.) Snell's law is based on the *index of refraction* for the medium that the incident ray is in and the index of refraction for the medium it is entering. The index of refraction describes how much more slowly light travels in a particular medium than in a vacuum. We will use the Greek letter  $\eta$ , pronounced "eta," to denote the index of refraction. Snell's law is

$$\eta_i \sin \theta_i = \eta_t \sin \theta_t. \quad [8.2]$$

In general, the index of refraction varies with the wavelength of light. Thus, incident light generally scatters in multiple directions at the boundary between two different media, an effect known as *dispersion*. This effect can be seen when incident white light is split into spectral components by a prism. Common practice in graphics is to ignore this wavelength dependence, since this effect is generally not crucial for visual accuracy and ignoring it simplifies light transport calculations substantially. Alternatively, the paths of multiple beams of light (e.g., at a series of discrete wavelengths) can be tracked through the environment in which a dispersive object is found. The "Further Reading" section at the end of Chapter 14 has pointers to more information on this topic.

Figure 8.4 shows the effect of perfect specular reflection and transmission.

### 8.2.1 FRESNEL REFLECTANCE

In addition to the reflected and transmitted directions, it is also necessary to compute the fraction of incoming light that is reflected or transmitted. For physically accurate reflection or refraction, these terms are directionally dependent and cannot be captured by constant per-surface scaling amounts. The *Fresnel equations* describe the amount of light reflected from a surface; they are the solution to Maxwell's equations at smooth surfaces.

Given the index of refraction and the angle which the incident ray makes with the surface normal, the Fresnel equations specify the material's corresponding reflectance for two different polarization states of the incident illumination. Because the visual effect of polarization is limited in most environments, in pbrt we will make the common assumption that light is unpolarized; that is, it is randomly oriented with respect to the light wave. With this simplifying assumption, the Fresnel reflectance is the average of the squares of the parallel and perpendicular polarization terms.



(a)

(b)

**Figure 8.4:** Dragon model rendered with (a) perfect specular reflection and (b) perfect specular refraction. Image (b) excludes the effects of external and internal reflection; the resulting energy loss produces conspicuous dark regions. (*Model courtesy of Christian Schüller.*)

At this point, it is necessary to draw a distinction among several important classes of materials:

1. The first class is *dielectrics*, which are materials that don't conduct electricity. They have real-valued indices of refraction (usually in the range 1-3) and transmit<sup>3</sup> a portion of the incident illumination. Examples of dielectrics are glass, mineral oil, water, and air.

2. The second class consists of *conductors* such as metals. Valence electrons can freely move within their atomic lattice, allowing electric currents to flow from one place to another. This fundamental atomic property translates into a profoundly different behavior when a conductor is subjected to electromagnetic radiation such as visible light: the material is opaque and reflects back a significant portion of the illumination.

A portion of the light is also transmitted into the interior of the conductor, where it is rapidly absorbed: total absorption typically occurs within the top  $0.1 \mu\text{m}$  of the material, hence only extremely thin metal films are capable of transmitting appreciable amounts of light. We ignore this effect in pbrt and only model the reflection component of conductors.

In contrast to dielectrics, conductors have a complex-valued index of refraction  $\bar{\eta} = \eta + ik$ .

3. Semiconductors such as silicon or germanium are the third class though we will not consider them in this book.

---

<sup>3</sup> Note that a dielectric can be filled with particles that absorb most or all of the transmitted light (e.g., petroleum). A dielectric such as water can also be turned into an electrolyte solution by adding ions that cause it to conduct electricity. Both of these aspects are unrelated to a material's intrinsic classification as a dielectric or conductor.

**Table 8.1:** Indices of refraction for a variety of objects, giving the ratio of the speed of light in a vacuum to the speed of light in the medium. These are generally wavelength-dependent quantities; these values are averages over the visible wavelengths.

Medium	Index of refraction $\eta$
Vacuum	1.0
Air at sea level	1.00029
Ice	1.31
Water (20°C)	1.333
Fused quartz	1.46
Glass	1.5–1.6
Sapphire	1.77
Diamond	2.42

Both conductors and dielectrics are governed by the same set of Fresnel equations. Despite this, we prefer to create a special evaluation function for dielectrics to benefit from the particularly simple form that these equations take on when the indices of refraction are guaranteed to be real-valued.

To compute the Fresnel reflectance at the interface of two dielectric media, we need to know the indices of refraction for the two media. Table 8.1 has the indices of refraction for a number of dielectric materials. The Fresnel reflectance formulae for dielectrics are

$$r_{\parallel} = \frac{\eta_t \cos \theta_i - \eta_i \cos \theta_t}{\eta_t \cos \theta_i + \eta_i \cos \theta_t},$$

$$r_{\perp} = \frac{\eta_i \cos \theta_i - \eta_t \cos \theta_t}{\eta_i \cos \theta_i + \eta_t \cos \theta_t},$$

where  $r_{\parallel}$  is the Fresnel reflectance for parallel polarized light and  $r_{\perp}$  is the reflectance for perpendicular polarized light,  $\eta_i$  and  $\eta_t$  are the indices of refraction for the incident and transmitted media, and  $\omega_i$  and  $\omega_t$  are the incident and transmitted directions.  $\omega_t$  can be computed with Snell's law (see Section 8.2.3).

The cosine terms should all be greater than or equal to zero; for the purposes of computing these values, the geometric normal should be flipped to be on the same side as  $\omega_i$  and  $\omega_t$  when computing  $\cos \theta_i$  and  $\cos \theta_t$ , respectively.

For unpolarized light, the Fresnel reflectance is

$$F_r = \frac{1}{2}(r_{\parallel}^2 + r_{\perp}^2).$$

Due to conservation of energy, the energy transmitted by a dielectric is  $1 - F_r$ .

The function `FrDielectric()` computes the Fresnel reflection formula for dielectric materials and unpolarized light. The quantity  $\cos \theta_i$  is passed in with the parameter `cosThetaI`.

`FrDielectric()` 519

```

⟨BxDF Utility Functions⟩ ≡
    Float FrDielectric(Float cosThetaI, Float etaI, Float etaT) {
        cosThetaI = Clamp(cosThetaI, -1, 1);
        ⟨Potentially swap indices of refraction 519⟩
        ⟨Compute cosThetaT using Snell's law 520⟩
        Float Rparl = ((etaT * cosThetaI) - (etaI * cosThetaT)) /
            ((etaT * cosThetaI) + (etaI * cosThetaT));
        Float Rperp = ((etaI * cosThetaI) - (etaT * cosThetaT)) /
            ((etaI * cosThetaI) + (etaT * cosThetaT));
        return (Rparl * Rparl + Rperp * Rperp) / 2;
    }
}

```

To find the cosine of the transmitted angle,  $\cos\theta_{\text{T}}$ , it is first necessary to determine if the incident direction is on the outside of the medium or inside it, so that the two indices of refraction can be interpreted appropriately.

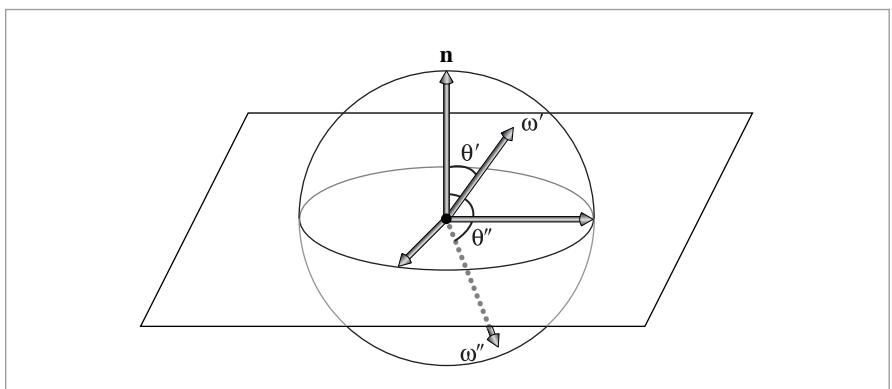
The sign of the cosine of the incident angle indicates on which side of the medium the incident ray lies (Figure 8.5). If the cosine is between 0 and 1, the ray is on the outside, and if the cosine is between  $-1$  and 0, the ray is on the inside. The parameters  $\text{etaI}$  and  $\text{etaT}$  are adjusted such that  $\text{etaI}$  has the index of refraction of the incident medium, and thus it is ensured that  $\cos\theta_{\text{I}}$  is nonnegative.

```

⟨Potentially swap indices of refraction⟩ ≡
    bool entering = cosThetaI > 0.f;
    if (!entering) {
        std::swap(etaI, etaT);
        cosThetaI = std::abs(cosThetaI);
    }
}

```

519



**Figure 8.5:** The cosine of the angle  $\theta$  between a direction  $\omega$  and the geometric surface normal indicates whether the direction is pointing outside the surface (in the same hemisphere as the normal) or inside the surface. In the standard reflection coordinate system, this test just requires checking the  $z$  component of the direction vector. Here,  $\omega'$  is in the upper hemisphere, with a positive-valued cosine, while  $\omega''$  is in the lower hemisphere.

Clamp() 1062

Float 1062

Once the indices of refraction are determined, we can compute the sine of the angle between the transmitted direction and the surface normal,  $\sin \theta_t$ , using Snell's law (Equation (8.2)). Finally, the cosine of this angle is found using the identity  $\sin^2 \theta + \cos^2 \theta = 1$ .

```
(Compute cosThetaT using Snell's law) ≡ 519
Float sinThetaI = std::sqrt(std::max((Float)0,
    1 - cosThetaI * cosThetaI));
Float sinThetaT = etaI / etaT * sinThetaI;
(Handle total internal reflection 520)
Float cosThetaT = std::sqrt(std::max((Float)0,
    1 - sinThetaT * sinThetaT));
```

When light is traveling from one medium to another medium with a lower index of refraction, none of the light at incident angles near grazing passes into the other medium. The largest angle at which this happens is called the *critical angle*; when  $\theta_i$  is greater than the critical angle, *total internal reflection* occurs, and all of the light is reflected. That case is detected here by a value of  $\sin \theta_t$  greater than one; in that case, the Fresnel equations are unnecessary.

```
(Handle total internal reflection) ≡ 520
if (sinThetaT >= 1)
    return 1;
```

We now focus on the general case of a complex index of refraction  $\bar{\eta} = \eta + ik$ , where some of the incident light is potentially absorbed by the material and turned into heat. In addition to the real part, the general Fresnel formula now also depends on the imaginary part  $k$  that is referred to as the *absorption coefficient*.

Figure 8.6 shows a plot of the index of refraction and absorption coefficient for gold; both of these are wavelength-dependent quantities. The directory `scenes/spds/metals` in the `pbrt` distribution has wavelength-dependent data for  $\eta$  and  $k$  for a variety of metals. Figure 9.4 in the next chapter shows a model rendered with a metal material.

The Fresnel reflectance at the boundary between a conductor and a dielectric medium is given by

$$r_{\perp} = \frac{a^2 + b^2 - 2a \cos \theta + \cos^2 \theta}{a^2 + b^2 + 2a \cos \theta + \cos^2 \theta}, \quad (8.3)$$

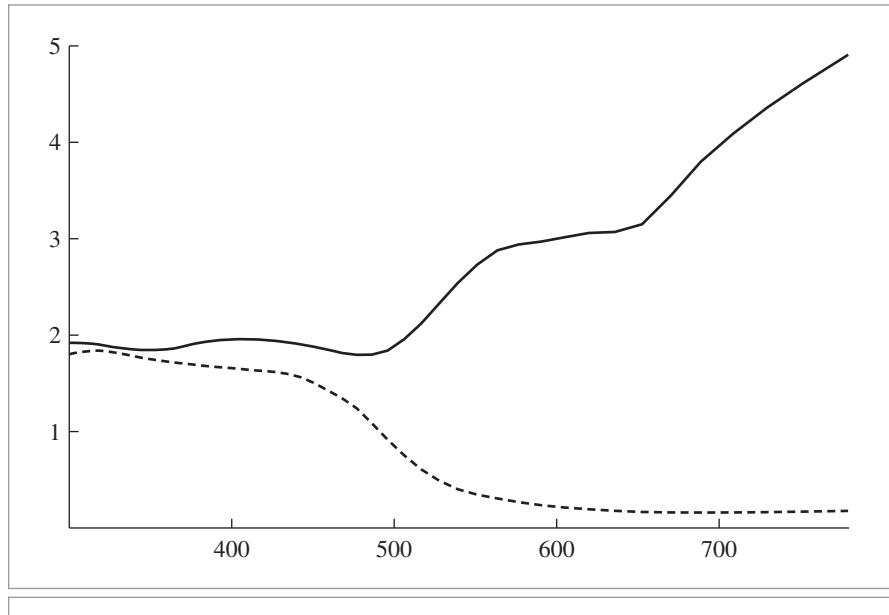
$$r_{\parallel} = r_{\perp} \frac{\cos^2 \theta (a^2 + b^2) - 2a \cos \theta \sin^2 \theta + \sin^4 \theta}{\cos^2 \theta (a^2 + b^2) + 2a \cos \theta \sin^2 \theta + \sin^4 \theta}, \quad (8.4)$$

where

$$a^2 + b^2 = \sqrt{(\eta^2 - k^2 - \sin^2 \theta)^2 + 4\eta^2 k^2},$$

and  $\eta + ik = \bar{\eta}_t/\bar{\eta}_i$  is the relative index of refraction computed using a complex division operation. However, generally  $\bar{\eta}_i$  will be a dielectric so that a normal real division can be used instead.

Float 1062



**Figure 8.6: Absorption Coefficient and Index of Refraction of Gold.** This plot shows the spectrally varying values of the absorption coefficient  $k$  (solid line) and the index of refraction  $\eta$  (dashed line) for gold, where the horizontal axis is wavelength in nm.

This computation is implemented by the `FrConductor()` function<sup>4</sup>; its implementation corresponds directly to Equations (8.3) and (8.4) and so isn't included here.

```
(Reflection Declarations) ≡
Spectrum FrConductor(Float cosThetaI, const Spectrum &etaI,
                      const Spectrum &etaT, const Spectrum &k);
```

For convenience, we will define an abstract `Fresnel` class that provides an interface for computing Fresnel reflection coefficients. Using implementations of this interface helps simplify the implementation of subsequent BRDFs that may need to support both forms.

```
(BxDF Declarations) +≡
class Fresnel {
public:
    (Fresnel Interface 522)
};
```

The only method provided by the `Fresnel` interface is `Fresnel::Evaluate()`. Given the cosine of the angle made by the incoming direction and the surface normal, it returns the amount of light reflected by the surface.

---

`Float` 1062  
`FrDielectric()` 519  
`Fresnel` 521  
`Fresnel::Evaluate()` 522  
`Spectrum` 315

<sup>4</sup> Note that this is a slight misnomer since the function technically subsumes the dielectric case when  $k = 0$ . That said, we chose this name to indicate that the function should only be used when dealing with conductors, since it is more expensive to evaluate than `FrDielectric()`.

*(Fresnel Interface)* ≡ 521  
 virtual Spectrum Evaluate(Float cosI) const = 0;

## Fresnel Conductors

`FresnelConductor` implements this interface for conductors.

*(BxDF Declarations)* +≡  
 class `FresnelConductor` : public `Fresnel` {  
 public:  
*(FresnelConductor Public Methods 522)*  
 private:  
 Spectrum etaI, etaT, k;  
};

Its constructor stores the given index of refraction  $\eta$  and absorption coefficient  $k$ .

*(FresnelConductor Public Methods)* ≡ 522  
`FresnelConductor`(const Spectrum &etaI, const Spectrum &etaT,  
 const Spectrum &k) : etaI(etaI), etaT(etaT), k(k) {}

The evaluation routine for `FresnelConductor` is also simple; it just calls the `FrConductor()` function defined earlier. Note that it takes the absolute value of `cosThetaI` before calling `FrConductor()`, since `FrConductor()` expects that the cosine will be measured with respect to the normal on the same side of the surface as  $\omega_i$ , or, equivalently, that the absolute value of  $\cos \theta_i$  should be used.

*(BxDF Method Definitions)* +≡  
 Spectrum `FresnelConductor::Evaluate`(Float cosThetaI) const {  
 return `FrConductor`(std::abs(cosThetaI), etaI, etaT, k);  
}

## Fresnel Dielectrics

`FresnelDielectric` similarly implements the `Fresnel` interface for dielectric materials.

*(BxDF Declarations)* +≡  
 class `FresnelDielectric` : public `Fresnel` {  
 public:  
*(FresnelDielectric Public Methods 522)*  
 private:  
 Float etaI, etaT;  
};

Its constructor stores the indices of refraction on the exterior and interior sides of the surface.

*(FresnelDielectric Public Methods)* ≡ 522  
`FresnelDielectric`(Float etaI, Float etaT) : etaI(etaI), etaT(etaT) {}

The evaluation routine for `FresnelDielectric` analogously calls `FrDielectric()`.

Float 1062  
`FrConductor()` 521  
`FrDielectric()` 519  
`Fresnel` 521  
`FresnelConductor` 522  
`FresnelDielectric` 522  
`FresnelDielectric::etaI` 522  
`FresnelDielectric::etaT` 522  
Spectrum 315

```
(BxDF Method Definitions) +≡
    Spectrum FresnelDielectric::Evaluate(Float cosThetaI) const {
        return FrDielectric(cosThetaI, etaI, etaT);
    }
```

### A Special Fresnel Interface

The `FresnelNoOp` implementation of the `Fresnel` interface returns 100% reflection for all incoming directions. Although this is physically implausible, it is a convenient capability to have available.

```
(BxDF Declarations) +≡
class FresnelNoOp : public Fresnel {
public:
    Spectrum Evaluate(Float) const { return Spectrum(1.); }
};
```

## 8.2.2 SPECULAR REFLECTION

We can now implement the `SpecularReflection` class, which describes physically plausible specular reflection, using the `Fresnel` interface to compute the fraction of light that is reflected. First, we will derive the BRDF that describes specular reflection. Since the `Fresnel` equations give the fraction of light reflected,  $F_r(\omega)$ , then we need a BRDF such that

$$L_o(\omega_o) = \int f_r(\omega_o, \omega_i) L_i(\omega_i) |\cos \theta_i| d\omega_i = F_r(\omega_r) L_i(\omega_r),$$

where  $\omega_r = R(\omega_o, \mathbf{n})$  is the specular reflection vector for  $\omega_o$  reflected about the surface normal  $\mathbf{n}$ . (Recall that  $\theta_r = \theta_o$  for specular reflection, and therefore  $F_r(\omega_o) = F_r(\omega_r)$ .)

Such a BRDF can be constructed using the Dirac delta distribution. Recall from Section 7.1 that the delta distribution has the useful property that

$$\int f(x) \delta(x - x_0) dx = f(x_0). \quad [8.5]$$

The delta distribution requires special handling compared to standard functions, however. In particular, numerical integration of integrals with delta distributions must explicitly account for the delta distribution. Consider the integral in Equation (8.5): if we tried to evaluate it using the trapezoid rule or some other numerical integration technique, by definition of the delta distribution there would be zero probability that any of the evaluation points  $x_i$  would have a nonzero value of  $\delta(x_i)$ . Rather, we must allow the delta distribution to determine the evaluation point itself. We will encounter delta distributions in light transport integrals both from specular BxDFs as well as from some of the light sources in Chapter 12.

Intuitively, we want the specular reflection BRDF to be zero everywhere except at the perfect reflection direction, which suggests the use of the delta distribution. A first guess might be to use delta functions to restrict the incident direction to the specular reflection direction  $\omega_r$ . This would yield a BRDF of

$$f_r(\omega_o, \omega_i) = \delta(\omega_i - \omega_r) F_r(\omega_i).$$

Although this seems appealing, plugging it into the scattering equation, Equation (5.9), reveals a problem:

$$\begin{aligned} L_o(\omega_o) &= \int \delta(\omega_i - \omega_r) F_r(\omega_i) L_i(\omega_i) |\cos \theta_i| d\omega_i \\ &= F_r(\omega_r) L_i(\omega_r) |\cos \theta_r|. \end{aligned}$$

This is not correct because it contains an extra factor of  $\cos \theta_r$ . However, we can divide out this factor to find the correct BRDF for perfect specular reflection:

$$f_r(p, \omega_o, \omega_i) = F_r(\omega_r) \frac{\delta(\omega_i - \omega_r)}{|\cos \theta_r|},$$

*(BxDF Declarations)* +≡

```
class SpecularReflection : public BxDF {
public:
    (SpecularReflection Public Methods 524)
private:
    (SpecularReflection Private Data 524)
};
```

The `SpecularReflection` constructor takes a `Spectrum` that is used to scale the reflected color and a `Fresnel` object pointer that describes dielectric or conductor Fresnel properties.

*(SpecularReflection Public Methods)* ≡

524

```
SpecularReflection(const Spectrum &R, Fresnel *fresnel)
: BxDF(BxDFType(BSDF_REFLECTION | BSDF_SPECULAR), R(R),
fresnel(fresnel) { }
```

*(SpecularReflection Private Data)* ≡

524

```
const Spectrum R;
const Fresnel *fresnel;
```

The rest of the implementation is straightforward. No scattering is returned from `SpecularReflection::f()`, since for an arbitrary pair of directions the delta function returns no scattering.<sup>5</sup>

*(SpecularReflection Public Methods)* +≡

524

```
Spectrum f(const Vector3f &wo, const Vector3f &wi) const {
    return Spectrum(0.f);
}
```

However, we do implement the `Sample_f()` method, which selects an appropriate direction according to the delta distribution. It sets the output variable `wi` to be the reflection of the supplied direction `wo` about the surface normal. The `*pdf` value is set to be one;

BSDF\_REFLECTION 513  
BSDF\_SPECULAR 513  
BxDF 513  
BxDFType 513  
Fresnel 521  
Spectrum 315  
SpecularReflection 524  
SpecularReflection::f() 524  
Vector3f 60

<sup>5</sup> If the caller happened to pass a vector and its perfect mirror direction, this function still returns zero. Although this might be a slightly confusing interface to these reflection functions, we still get the correct result in the end because reflection functions involving singularities with delta distributions receive special handling by the light transport routines (see Chapter 14).

Section 14.1.3 discusses some subtleties about the mathematical quantity that this value of one represents.

```
<BxDF Method Definitions> +≡
Spectrum SpecularReflection::Sample_f(const Vector3f &wo,
    Vector3f *wi, const Point2f &sample, Float *pdf,
    BxDFType *sampledType) const {
    (Compute perfect specular reflection direction 526)
    *pdf = 1;
    return fresnel->Evaluate(CosTheta(*wi)) * R / AbsCosTheta(*wi);
}
```

The desired incident direction is the reflection of  $\omega_0$  around the surface normal,  $R(\omega_0, \mathbf{n})$ . This direction can be computed fairly easily using vector geometry. First, note that the incoming direction, the reflection direction, and the surface normal all lie in the same plane. We can decompose vectors  $\omega$  that lie in a plane into a sum of two components: one parallel to  $\mathbf{n}$ , which we'll denote by  $\omega_{\parallel}$ , and one perpendicular,  $\omega_{\perp}$ .

These vectors are easily computed: if  $\mathbf{n}$  and  $\omega$  are normalized, then  $\omega_{\parallel}$  is  $(\cos \theta)\mathbf{n} = (\mathbf{n} \cdot \omega)\mathbf{n}$  (Figure 8.7). Because  $\omega_{\parallel} + \omega_{\perp} = \omega$ ,

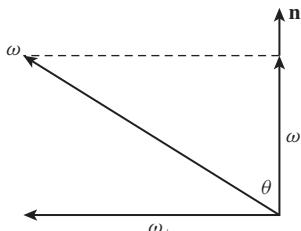
$$\omega_{\perp} = \omega - \omega_{\parallel} = \omega - (\mathbf{n} \cdot \omega)\mathbf{n}.$$

Figure 8.8 shows the setting for computing the reflected direction  $\omega_r$ . We can see that both vectors have the same  $\omega_{\parallel}$  component, and the value of  $\omega_{r\perp}$  is the negation of  $\omega_{o\perp}$ . Therefore, we have

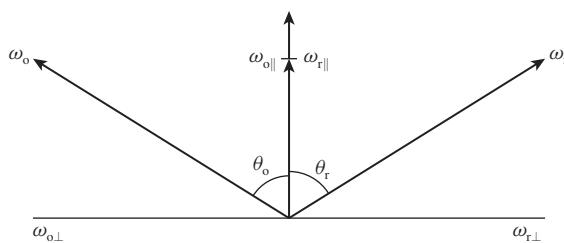
$$\begin{aligned}\omega_r &= \omega_{r\perp} + \omega_{r\parallel} = -\omega_{o\perp} + \omega_{o\parallel} \\ &= -(\omega_o - (\mathbf{n} \cdot \omega_o)\mathbf{n}) + (\mathbf{n} \cdot \omega_o)\mathbf{n} \\ &= -\omega_o + 2(\mathbf{n} \cdot \omega_o)\mathbf{n}.\end{aligned}\quad [8.6]$$

The `Reflect()` function implements this computation.

AbsCosTheta() 510
BxDFType 513
CosTheta() 510
Float 1062
Fresnel::Evaluate() 522
Point2f 68
Spectrum 315
SpecularReflection 524
SpecularReflection::fresnel 524
SpecularReflection::R 524
Vector3f 60



**Figure 8.7:** The parallel projection of a vector  $\omega$  on to the normal  $\mathbf{n}$  is given by  $\omega_{\parallel} = (\cos \theta)\mathbf{n} = (\mathbf{n} \cdot \omega)\mathbf{n}$ . The perpendicular component is given by  $\omega_{\perp} = (\sin \theta)\mathbf{n}$  but is more easily computed by  $\omega_{\perp} = \omega - \omega_{\parallel}$ .



**Figure 8.8:** Because the angles  $\theta_o$  and  $\theta_r$  are equal, the parallel component of the perfect reflection direction  $\omega_{r\parallel}$  is the same as the incident direction's:  $\omega_{r\parallel} = \omega_{o\parallel}$ . Its perpendicular component is just the incident direction's perpendicular component, negated.

*(BSDF Inline Functions)*  $\equiv$

```
inline Vector3f Reflect(const Vector3f &wo, const Vector3f &n) {
    return -wo + 2 * Dot(wo, n) * n;
}
```

In the BRDF coordinate system,  $\mathbf{n} = (0, 0, 1)$ , and this expression is substantially simpler.

*(Compute perfect specular reflection direction)*  $\equiv$

525, 817

```
*wi = Vector3f(-wo.x, -wo.y, wo.z);
```

### 8.2.3 SPECULAR TRANSMISSION

We will now derive the BTDF for specular transmission. Snell's law is the basis of the derivation. Not only does Snell's law give the direction for the transmitted ray, but it can also be used to show that radiance along a ray changes as the ray goes between media with different indices of refraction.

Consider incident radiance arriving at the boundary between two media, with indices of refraction  $\eta_i$  and  $\eta_o$  for the incoming and outgoing media, respectively (Figure 8.9). We use  $\tau$  to denote the fraction of incident energy that is transmitted to the outgoing direction as given by the Fresnel equations, so  $\tau = 1 - F_r(\omega_i)$ . The amount of transmitted differential flux, then, is

$$d\Phi_o = \tau d\Phi_i.$$

If we use the definition of radiance, Equation (5.2), we have

$$L_o \cos \theta_o dA d\omega_o = \tau (L_i \cos \theta_i dA d\omega_i).$$

Expanding the solid angles to spherical angles, we have

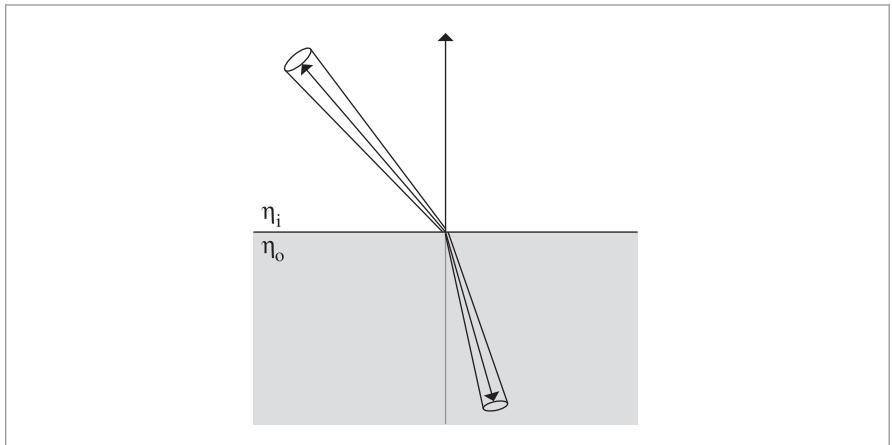
$$L_o \cos \theta_o dA \sin \theta_o d\theta_o d\phi_o = \tau L_i \cos \theta_i dA \sin \theta_i d\theta_i d\phi_i. \quad [8.7]$$

Dot() 63

Vector3f 60

We can now differentiate Snell's law with respect to  $\theta$ , which gives the relation

$$\eta_o \cos \theta_o d\theta_o = \eta_i \cos \theta_i d\theta_i.$$



**Figure 8.9:** The amount of transmitted radiance at the boundary between media with different indices of refraction is scaled by the squared ratio of the two indices of refraction. Intuitively, this can be understood as the result of the radiance's differential solid angle being compressed or expanded as a result of transmission.

Rearranging terms, we have

$$\frac{\cos \theta_o d\phi_o}{\cos \theta_i d\theta_i} = \frac{\eta_i}{\eta_o}.$$

Substituting this relationship and Snell's law into Equation (8.7) and then simplifying, we have

$$L_o \eta_i^2 d\phi_o = \tau L_i \eta_o^2 d\phi_i.$$

Because  $\phi_i = \phi_o + \pi$  and therefore  $d\phi_i = d\phi_o$ , we have the final relationship:

$$L_o = \tau L_i \frac{\eta_o^2}{\eta_i^2}. \quad (8.8)$$

As with the BRDF for specular reflection, we need to divide out a  $\cos \theta_i$  term to get the right BTDF for specular transmission:

$$f_r(\omega_o, \omega_i) = \frac{\eta_o^2}{\eta_i^2} (1 - F_r(\omega_i)) \frac{\delta(\omega_i - T(\omega_o, \mathbf{n}))}{|\cos \theta_i|},$$

where  $T(\omega_o, \mathbf{n})$  is the specular transmission vector that results from specular transmission of  $\omega_o$  through an interface with surface normal  $\mathbf{n}$ .

The  $1 - F_r(\omega_i)$  term in this equation corresponds to an easily observed effect: transmission is stronger at near-perpendicular angles. For example, if you look straight down into a clear lake, you can see far into the water, but at grazing angles most of the light is reflected as if from a mirror.

The SpecularTransmission class is almost exactly the same as SpecularReflection except that the sampled direction is the direction for perfect specular transmission. Figure 8.10



**Figure 8.10:** When the BRDF for specular reflection and the BTDF for specular transmission are modulated with the Fresnel formula for dielectrics, the realistic angle-dependent variation of the amount of reflection and transmission gives a visually accurate representation of the glass. (Model courtesy of Christian Schüller.)

shows an image of the dragon model using specular reflection and transmission BRDF and BTDF to model glass.

```
(BxDF Declarations) +≡
class SpecularTransmission : public BxDF {
public:
    (SpecularTransmission Public Methods 528)
private:
    (SpecularTransmission Private Data 529)
};
```

The SpecularTransmission constructor stores the indices of refraction on both sides of the surface, where etaA is the index of refraction above the surface (where the side the surface normal lies in is “above”), etaB is the index of refraction below the surface, and T gives a transmission scale factor. The TransportMode parameter indicates whether the incident ray that intersected the point where the BxDF was computed started from a light source or whether it was started from the camera. This distinction has implications for how the BxDF’s contribution is computed.

```
(SpecularTransmission Public Methods) ≡
SpecularTransmission(const Spectrum &T, Float etaA, Float etaB,
TransportMode mode)
: BxDF(BxDFType(BSDF_TRANSMISSION | BSDF_SPECULAR)), T(T), etaA(etaA),
etaB(etaB), fresnel(etaA, etaB), mode(mode) {
}
```

528

BSDF\_SPECULAR 513  
 BSDF\_TRANSMISSION 513  
 BxDF 513  
 BxDFType 513  
 Float 1062  
 Spectrum 315  
 SpecularTransmission 528  
 SpecularTransmission::etaA 529  
 SpecularTransmission::etaB 529  
 SpecularTransmission::fresnel 529  
 SpecularTransmission::mode 529  
 SpecularTransmission::T 529  
 TransportMode 960

Because conductors do not transmit light, a `FresnelDielectric` object is always used for the Fresnel computations.

*(SpecularTransmission Private Data)*  $\equiv$

```
const Spectrum T;
const Float etaA, etaB;
const FresnelDielectric fresnel;
const TransportMode mode;
```

528

As with `SpecularReflection`, zero is always returned from `SpecularTransmission::f()`, since the BTDF is a scaled delta distribution.

*(SpecularTransmission Public Methods)*  $+\equiv$

```
Spectrum f(const Vector3f &wo, const Vector3f &wi) const {
    return Spectrum(0.f);
}
```

528

Equation (8.8) describes how radiance changes as a ray passes from one medium to another. However, it turns out that while this scaling should be applied for rays starting at light sources, it must *not* be applied for rays starting from the camera. This issue is discussed in more detail in Section 16.1, and the fragment that applies this scaling, *(Account for non-symmetry with transmission to different medium)*, is defined there.

*(BxDF Method Definitions)*  $+\equiv$

```
Spectrum SpecularTransmission::Sample_f(const Vector3f &wo,
                                         Vector3f *wi, const Point2f &sample, Float *pdf,
                                         BxDFType *sampledType) const {
    (Figure out which  $\eta$  is incident and which is transmitted 529)
    (Compute ray direction for specular transmission 529)
    *pdf = 1;
    Spectrum ft = T * (Spectrum(1.) - fresnel.Evaluate(CosTheta(*wi)));
    (Account for non-symmetry with transmission to different medium 961)
    return ft / AbsCosTheta(*wi);
}
```

The method first determines whether the incident ray is entering or exiting the refractive medium. pbrt uses the convention that the surface normal, and thus the  $(0, 0, 1)$  direction in local reflection space, is oriented such that it points toward the outside of the object. Therefore, if the  $z$  component of the  $\omega_o$  direction is greater than zero, the incident ray is coming from outside of the object.

*(Figure out which  $\eta$  is incident and which is transmitted)*  $\equiv$

```
bool entering = CosTheta(wo) > 0;
Float etaI = entering ? etaA : etaB;
Float etaT = entering ? etaB : etaA;
```

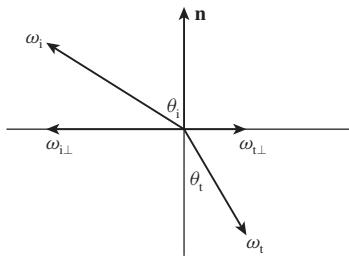
529, 817

*(Compute ray direction for specular transmission)*  $\equiv$

```
if (!Refract(wo, Faceforward(Normal3f(0, 0, 1), wo), etaI / etaT, wi))
    return 0;
```

529, 817

AbsCosTheta() 510  
 BxDFType 513  
 CosTheta() 510  
 Faceforward() 72  
 Float 1062  
 Fresnel::Evaluate() 522  
 FresnelDielectric 522  
 Normal3f 71  
 Point2f 68  
 Refract() 531  
 Spectrum 315  
 SpecularReflection 524  
 SpecularTransmission 528  
 SpecularTransmission::etaA 529  
 SpecularTransmission::etaB 529  
 SpecularTransmission::f() 529  
 SpecularTransmission::T 529  
 TransportMode 960  
 Vector3f 60



**Figure 8.11: The Geometry of Specular Transmission.** Given an incident direction  $\omega$  and surface normal  $n$  with angle  $\theta$  between them, the specularly transmitted direction makes an angle  $\theta_t$  with the surface normal. This direction,  $\omega_t$ , can be computed by using Snell's law to find its perpendicular component  $\omega_{t\perp}$  and then computing the  $\omega_{t\parallel}$  that gives a normalized result  $\omega_t$ .

To derive the expression that gives the transmitted direction vector, we can follow a similar approach to the one we used earlier for specular reflection. Figure 8.11 shows the setting. This time, however, we'll start with the perpendicular component: if the incident vector is normalized and has perpendicular component  $\omega_{i\perp}$ , then we know from trigonometry and the definition of  $\omega_\perp$  that the length of  $\omega_{i\perp}$  is equal to  $\sin \theta_i$ . Snell's law tells us that  $\sin \theta_t = \eta_i / \eta_t \sin \theta_i$ . Negating the direction of  $\omega_{i\perp}$  and adjusting the length accordingly, we have

$$\omega_{t\perp} = \frac{\eta_i}{\eta_t} (-\omega_{i\perp}).$$

Equivalently, because  $\omega_\perp = \omega - \omega_\parallel$ ,

$$\omega_{t\perp} = \frac{\eta_i}{\eta_t} \left( -\omega_i + (\omega_i \cdot n)n \right).$$

Now for  $\omega_{t\parallel}$ : we know that  $\omega_{t\parallel}$  is parallel to  $n$  but facing in the opposite direction, and we also know that  $\omega_t$  should be normalized. Putting these together,

$$\omega_{t\parallel} = - \left( \sqrt{1 - \|\omega_{t\perp}\|^2} \right) n.$$

The full vector  $\omega_t$ , then, is

$$\omega_t = \omega_{t\perp} + \omega_{t\parallel} = \frac{\eta_i}{\eta_t} (-\omega_i) + \left[ \frac{\eta_i}{\eta_t} (\omega_i \cdot n) - \sqrt{1 - \|\omega_{t\perp}\|^2} \right] n.$$

Because  $\|\omega_{t\perp}\| = \sin \theta_t$ , the term under the square root is  $1 - \sin^2 \theta_t = \cos^2 \theta_t$ , which gives the final result:

$$\omega_t = \frac{\eta_i}{\eta_t} (-\omega_i) + \left[ \frac{\eta_i}{\eta_t} (\omega_i \cdot n) - \cos \theta_t \right] n. \quad [8.9]$$

The `Refract()` function computes the refracted direction `wt` given an incident direction `wi`, surface normal `n` in the same hemisphere was `wi`, and `eta`, the ratio of indices of refraction in the incident and transmitted media, respectively. The Boolean return value indicates whether a valid refracted ray was returned in `*wt`; it is `false` in the case of total internal refraction.

```
(BSDF Inline Functions) +≡
    inline bool Refract(const Vector3f &wi, const Normal3f &n, Float eta,
                        Vector3f *wt) {
        (Compute cos θt using Snell's law 531)
        *wt = eta * -wi + (eta * cosThetaI - cosThetaT) * Vector3f(n);
        return true;
    }
```

Squaring both sides of Snell's law lets us compute  $\cos \theta_t$ :

$$\eta_i^2 \sin^2 \theta_i = \eta_t^2 \sin^2 \theta_t \quad \sin^2 \theta_t = \frac{\eta_i^2}{\eta_t^2} \sin^2 \theta_i$$

$$1 - \cos^2 \theta_t = \frac{\eta_i^2}{\eta_t^2} \sin^2 \theta_i \quad \cos \theta_t = \sqrt{1 - \frac{\eta_i^2}{\eta_t^2} \sin^2 \theta_i}$$

*(Compute cos θ<sub>t</sub> using Snell's law) ≡*

531

```
Float cosThetaI = Dot(n, wi);
Float sin2ThetaI = std::max(0.f, 1.f - cosThetaI * cosThetaI);
Float sin2ThetaT = eta * eta * sin2ThetaI;
(Handle total internal reflection for transmission 531)
Float cosThetaT = std::sqrt(1 - sin2ThetaT);
```

We need to handle the case of total internal reflection here as well. If the squared value of  $\sin \theta_t$  is greater than or equal to one, total internal reflection has occurred, so false is returned.<sup>6</sup>

*(Handle total internal reflection for transmission) ≡*

531

```
if (sin2ThetaT >= 1) return false;
```

## 8.2.4 FRESNEL-MODULATED SPECULAR REFLECTION AND TRANSMISSION

For better efficiency in some of the Monte Carlo light transport algorithms to come in Chapters 14 through 16, it's useful to have a single BxDF that represents both specular reflection and specular transmission together, where the relative weightings of the types of scattering are modulated by the dielectric Fresnel equations. Such a BxDF is provided in `FresnelSpecular`.

*(BxDF Declarations) +≡*

```
class FresnelSpecular : public BxDF {
public:
    (FresnelSpecular Public Methods 532)
private:
    (FresnelSpecular Private Data 532)
};
```

BxDF 513

Dot() 63

Float 1062

Normal3f 71

Vector3f 60

<sup>6</sup> The first version of pbrt had a test  $> 1$  rather than  $\geq 1$  here. Though the difference between the two may seem innocuous, this discrepancy led to not-a-number values occasionally being computed due to the  $z$  component of  $\omega_i$  being zero (in the tangent plane of the surface) and thus to the  $1/\cos \theta$  term being infinite.

```
(FresnelSpecular Public Methods) ≡ 531
FresnelSpecular(const Spectrum &R, const Spectrum &T, Float etaA,
    Float etaB, TransportMode mode)
: BxDF(BxDFType(BSDF_REFLECTION | BSDF_TRANSMISSION | BSDF_SPECULAR)),
    R(R), T(T), etaA(etaA), etaB(etaB), fresnel(etaA, etaB),
    mode(mode) { }
```

Since we only focus on the dielectric case, a `FresnelDielectric` object is always used for the Fresnel computations.

```
(FresnelSpecular Private Data) ≡ 531
const Spectrum R, T;
const Float etaA, etaB;
const FresnelDielectric fresnel;
const TransportMode mode;
```

```
(FresnelSpecular Public Methods) +≡ 531
Spectrum f(const Vector3f &wo, const Vector3f &wi) const {
    return Spectrum(0.f);
}
```

Because some of the implementation details depend on principles of Monte Carlo integration that are introduced in Chapters 13 and 14, the implementation of the `Sample_f()` method is in Section 14.1.3.

## 8.3 LAMBERTIAN REFLECTION

One of the simplest BRDFs is the Lambertian model. It models a perfect diffuse surface that scatters incident illumination equally in all directions. Although this reflection model is not physically plausible, it is a reasonable approximation to many real-world surfaces such as matte paint.

```
(BxDF Declarations) +≡
class LambertianReflection : public BxDF {
public:
    (LambertianReflection Public Methods 532)
private:
    (LambertianReflection Private Data 532)
};
```

The `LambertianReflection` constructor takes a reflectance spectrum  $R$ , which gives the fraction of incident light that is scattered.

```
(LambertianReflection Public Methods) ≡ 532
LambertianReflection(const Spectrum &R)
: BxDF(BxDFType(BSDF_REFLECTION | BSDF_DIFFUSE)), R(R) { }

(LambertianReflection Private Data) ≡ 532
const Spectrum R;
```

BSDF\_DIFFUSE 513  
 BSDF\_REFLECTION 513  
 BSDF\_SPECULAR 513  
 BSDF\_TRANSMISSION 513  
 BxDF 513  
 BxDFType 513  
 Float 1062  
 FresnelDielectric 522  
 FresnelSpecular 531  
 FresnelSpecular::etaA 532  
 FresnelSpecular::etaB 532  
 FresnelSpecular::fresnel 532  
 FresnelSpecular::mode 532  
 FresnelSpecular::R 532  
 FresnelSpecular::T 532  
 LambertianReflection 532  
 LambertianReflection::R 532  
 Spectrum 315  
 TransportMode 960  
 Vector3f 60

The reflection distribution function for `LambertianReflection` is quite straightforward, since its value is constant. However, the value  $R/\pi$  must be returned, rather than the reflectance  $R$  supplied to the constructor. This can be seen by equating  $R$  to Equation (8.1), which defined  $\rho_{hd}$ , and solving for the BRDF's value.

*(BxDF Method Definitions) +≡*

```
Spectrum LambertianReflection::f(const Vector3f &wo,
                                 const Vector3f &wi) const {
    return R * InvPi;
}
```

The directional-hemispherical and hemispherical-hemispherical reflectance values for a Lambertian BRDF are trivial to compute analytically, so the derivations are omitted in the text.

*(LambertianReflection Public Methods) +≡*

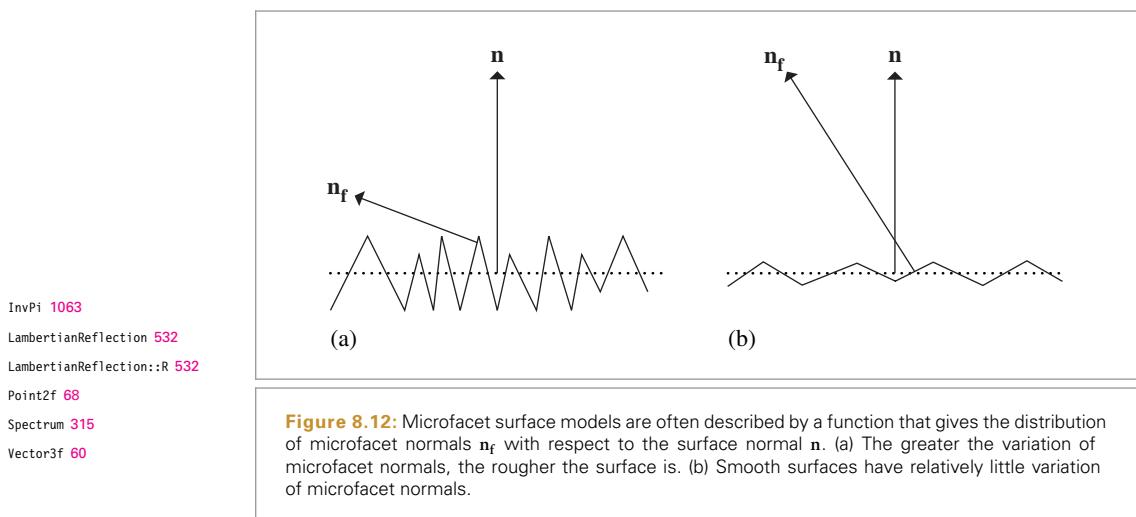
532

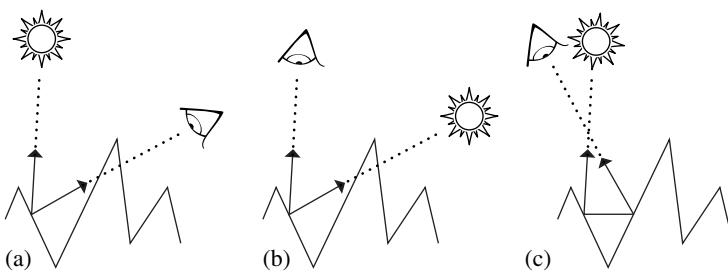
```
Spectrum rho(const Vector3f &, int, const Point2f *) const { return R; }
Spectrum rho(int, const Point2f *, const Point2f *) const { return R; }
```

It's also useful to be able to represent perfect Lambertian transmission through a surface; this BTDF is implemented in `LambertianTransmission`. Its implementation closely follows `LambertianReflection` and thus isn't included here.

## 8.4 MICROFACET MODELS

Many geometric-optics-based approaches to modeling surface reflection and transmission are based on the idea that rough surfaces can be modeled as a collection of small *microfacets*. Surfaces comprised of microfacets are often modeled as heightfields, where the distribution of facet orientations is described statistically. Figure 8.12 shows cross sections of a relatively rough surface and a much smoother microfacet surface. When the





**Figure 8.13: Three Important Geometric Effects to Consider with Microfacet Reflection Models.** (a) *Masking*: the microfacet of interest isn't visible to the viewer due to occlusion by another microfacet. (b) *Shadowing*: analogously, light doesn't reach the microfacet. (c) *Interreflection*: light bounces among the microfacets before reaching the viewer.

distinction isn't clear, we'll use the term *microsurface* to describe microfacet surfaces and *macrosurface* to describe the underlying smooth surface (e.g., as represented by a Shape).

Microfacet-based BRDF models work by statistically modeling the scattering of light from a large collection of microfacets. If we assume that the differential area being illuminated,  $dA$ , is relatively large compared to the size of individual microfacets, then a large number of microfacets are illuminated and it's their aggregate behavior that determines the observed scattering.

The two main components of microfacet models are a representation of the distribution of facets and a BRDF that describes how light scatters from individual microfacets. Given these, the task is to derive a closed-form expression giving the BRDF that describes scattering from such a surface. Perfect mirror reflection is most commonly used for the microfacet BRDF, though specular transmission is useful for modeling many translucent materials, and the Oren–Nayar model (described in the next section) treats microfacets as Lambertian reflectors.

To compute reflection from such a model, local lighting effects at the microfacet level need to be considered (Figure 8.13). Microfacets may be occluded by another facet, may lie in the shadow of a neighboring microfacet, or interreflection may cause a microfacet to reflect more light than predicted by the amount of direct illumination and the low-level microfacet BRDF. Particular microfacet-based BRDF models consider each of these effects with varying degrees of accuracy. The general approach is to make the best approximations possible, while still obtaining an easily evaluated expression.

#### 8.4.1 OREN–NAYAR DIFFUSE REFLECTION

Oren and Nayar (1994) observed that real-world objects do not exhibit perfect Lambertian reflection. Specifically, rough surfaces generally appear brighter as the illumination direction approaches the viewing direction. They developed a reflection model that describes rough surfaces by V-shaped microfacets described by a spherical Gaussian distri-



(a)



(b)

**Figure 8.14:** Dragon model rendered (a) with standard diffuse reflection from the `LambertianReflection` model and (b) with the `OrenNayar` model with a  $\sigma$  parameter of 20 degrees. Note the increase in reflection at the silhouette edges and the generally less-drawn-out transitions at light terminator edges with the Oren–Nayar model. (Model courtesy of Christian Schüller.)

bution with a single parameter  $\sigma$ , the standard deviation of the microfacet orientation angle.

Under the V-shape assumption, interreflection can be accounted for by only considering the neighboring microfacet; Oren and Nayar took advantage of this to derive a BRDF that models the aggregate reflection of the collection of grooves.

The resulting model, which accounts for shadowing, masking, and interreflection among the microfacets, does not have a closed-form solution, so they found the following approximation that fit it well:

$$f_r(\omega_i, \omega_o) = \frac{R}{\pi} (A + B \max(0, \cos(\phi_i - \phi_o)) \sin \alpha \tan \beta),$$

where if  $\sigma$  is in radians,

$$A = 1 - \frac{\sigma^2}{2(\sigma^2 + 0.33)}$$

$$B = \frac{0.45\sigma^2}{\sigma^2 + 0.09}$$

$$\alpha = \max(\theta_i, \theta_o)$$

$$\beta = \min(\theta_i, \theta_o).$$

LambertianReflection 532

OrenNayar 536

The implementation here precomputes and stores the values of the  $A$  and  $B$  parameters in the constructor to save work in evaluating the BRDF later. Figure 8.14 compares the difference between rendering with ideal diffuse reflection and with the Oren–Nayar model.

*(OrenNayar Public Methods)*  $\equiv$

```
OrenNayar(const Spectrum &R, Float sigma)
    : BxDF(BxDFType(BSDF_REFLECTION | BSDF_DIFFUSE)), R(R) {
    sigma = Radians(sigma);
    Float sigma2 = sigma * sigma;
    A = 1.f - (sigma2 / (2.f * (sigma2 + 0.33f)));
    B = 0.45f * sigma2 / (sigma2 + 0.09f);
}
```

*(OrenNayar Private Data)*  $\equiv$

```
const Spectrum R;
Float A, B;
```

Application of trigonometric identities can substantially improve the efficiency of the evaluation routine compared to a direct translation of the underlying equations. The implementation starts by computing the values of  $\sin \theta_i$  and  $\sin \theta_o$ .

*(BxDF Method Definitions)*  $+ \equiv$

```
Spectrum OrenNayar::f(const Vector3f &wo, const Vector3f &wi) const {
    Float sinThetaI = SinTheta(wi);
    Float sinTheta0 = SinTheta(wo);
    (Compute cosine term of Oren–Nayar model 536)
    (Compute sine and tangent terms of Oren–Nayar model 537)
    return R * InvPi * (A + B * maxCos * sinAlpha * tanBeta);
}
```

To compute the  $\max(0, \cos(\phi_i - \phi_o))$  term, we can apply the trigonometric identity

$$\cos(a - b) = \cos a \cos b + \sin a \sin b,$$

such that we just need to compute the sines and cosines of  $\phi_i$  and  $\phi_o$ .

*(Compute cosine term of Oren–Nayar model)*  $\equiv$

536

```
Float maxCos = 0;
if (sinThetaI > 1e-4 && sinTheta0 > 1e-4) {
    Float sinPhiI = SinPhi(wi), cosPhiI = CosPhi(wi);
    Float sinPhi0 = SinPhi(wo), cosPhi0 = CosPhi(wo);
    Float dCos = cosPhiI * cosPhi0 + sinPhiI * sinPhi0;
    maxCos = std::max((Float)0, dCos);
}
```

BSDF\_DIFFUSE 513  
 BSDF\_REFLECTION 513  
 BxDF 513  
 BxDFType 513  
 CosPhi() 511  
 Float 1062  
 InvPi 1063  
 OrenNayar 536  
 OrenNayar::A 536  
 OrenNayar::B 536  
 OrenNayar::R 536  
 SinPhi() 511  
 SinTheta() 510  
 Spectrum 315  
 Vector3f 60

Finally, the  $\sin \alpha$  and  $\tan \beta$  terms are found. Note that whichever of  $\omega_i$  or  $\omega_o$  has a larger value for  $\cos \theta$  (i.e., a larger  $z$  component) has a smaller value for  $\theta$ . We can set  $\sin \alpha$  using the appropriate sine value computed at the beginning of the method. The tangent can then be computed using the identity  $\tan \theta = \sin \theta / \cos \theta$ .

*{Compute sine and tangent terms of Oren–Nayar model} ≡*

```
Float sinAlpha, tanBeta;
if (AbsCosTheta(wi) > AbsCosTheta(wo)) {
    sinAlpha = sinTheta0;
    tanBeta = sinThetaI / AbsCosTheta(wi);
} else {
    sinAlpha = sinThetaI;
    tanBeta = sinTheta0 / AbsCosTheta(wo);
}
```

536

### 8.4.2 MICROFACET DISTRIBUTION FUNCTIONS

Reflection models based on microfacets that exhibit perfect specular reflection and transmission have been effective at modeling light scattering from a variety of glossy materials, including metals, plastic, and frosted glass. Before we discuss the radiometric details of these models, we'll first introduce abstractions that encapsulate their geometric properties. The code here includes implementations of two widely used microfacet models. All of this code is in the files `core/microfacet.h` and `core/microfacet.cpp`.

`MicrofacetDistribution` defines the interface provided by microfacet implementations as well as some common functionality for them.

*{MicrofacetDistribution Declarations} ≡*

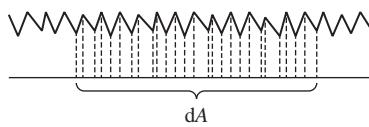
```
class MicrofacetDistribution {
public:
    {MicrofacetDistribution Public Methods 538}
protected:
    {MicrofacetDistribution Protected Methods 808}
    {MicrofacetDistribution Protected Data 808}
};
```

One important characteristic of a microfacet surface is represented by the distribution function  $D(\omega_h)$ , which gives the differential area of microfacets with the surface normal  $\omega_h$  (recall Figure 8.12, which shows how surface roughness and the microfacet normal distribution function are related). In `pbrt`, microfacet distribution functions are defined in the same BSDF coordinate system as `BxDFs`; as such, a perfectly smooth surface could be described by a delta distribution that was non-zero only when  $\omega_h$  was equal to the surface normal:  $D(\omega_h) = \delta(\omega_h - (0, 0, 1))$ .

Microfacet distribution functions must be *normalized* to ensure that they are physically plausible. Intuitively, if we consider incident rays on the microsurface along the normal direction  $\mathbf{n}$ , then each ray must intersect the microfacet surface exactly once. More formally, given a differential area of the microsurface,  $dA$ , then the projected area of the microfacet faces above that area must be equal to  $dA$  (Figure 8.15). Mathematically, this

`AbsCosTheta()` 510

`Float` 1062



**Figure 8.15:** Given a differential area on a surface  $dA$ , then the microfacet normal distribution function  $D(\omega_h)$  must be normalized such that the projected surface area of the microfacets above the area is equal to  $dA$ .

corresponds to the following requirement:<sup>7</sup>

$$\int_{\mathcal{H}^2(\mathbf{n})} D(\omega_h) \cos \theta_h d\omega_h = 1.$$

The method `MicrofacetDistribution::D()` corresponds to the function  $D(\omega_h)$ ; implementations return the differential area of microfacets oriented with the given normal vector  $\omega$ .

*(MicrofacetDistribution Public Methods)*  $\equiv$

537

```
virtual float D(const Vector3f &wh) const = 0;
```

A widely used microfacet distribution function based on a Gaussian distribution of microfacet slopes is due to Beckmann and Spizzichino (1963); our implementation is in the `BeckmannDistribution` class.

*(MicrofacetDistribution Declarations)*  $+ \equiv$

```
class BeckmannDistribution : public MicrofacetDistribution {
public:
    (BeckmannDistribution Public Methods)
private:
    (BeckmannDistribution Private Methods)
    (BeckmannDistribution Private Data 539)
};
```

The traditional definition of the Beckmann–Spizzichino model is

$$D(\omega_h) = \frac{e^{-\tan^2 \theta_h / \alpha^2}}{\pi \alpha^2 \cos^4 \theta_h}, \quad [8.10]$$

where if  $\sigma$  is the RMS slope of the microfacets, then  $\alpha = \sqrt{2}\sigma$ .

It's useful to define an anisotropic distribution, where the normal distribution also varies depending on the azimuthal orientation of  $\omega_h$ . For example, given a  $\alpha_x$  for microfacets oriented perpendicular to the  $x$  axis and  $\alpha_y$  for the  $y$  axis, then  $\alpha$  values for intermediate orientations can be interpolated by constructing an ellipse through these values.

Float 1062

MicrofacetDistribution 537

MicrofacetDistribution::D() 538

Vector3f 60

<sup>7</sup> A common error in normalizing microfacet distributions is to perform this integral over solid angle instead of projected solid angle (i.e., to leave out the  $\cos \theta_h$  term), which does not guarantee the existence of a heightfield with the right distribution.

The corresponding anisotropic microfacet distribution function is

$$D(\omega_h) = \frac{e^{-\tan^2 \theta_h (\cos^2 \phi_h / \alpha_x^2 + \sin^2 \phi_h / \alpha_y^2)}}{\pi \alpha_x \alpha_y \cos^4 \theta_h}. \quad [8.11]$$

Note that the original isotropic variant of the Beckmann–Spizzichino model falls out when  $\alpha_x = \alpha_y$ .

The `alphax` and `alphay` member variables are set in the `BeckmannDistribution` constructor, which is straightforward and therefore not included here.

```
{BeckmannDistribution Private Data} ≡
    const Float alphax, alphay;
```

538

The `BeckmannDistribution::D()` method is a direct translation of Equation (8.11). The only additional implementation detail is that infinite values of  $\tan^2 \theta$  must be handled specially. This case is actually valid—it happens at perfectly grazing directions. In this case, the code below ends up attempting to compute 0/0, which results in a “not a number” (NaN) value, which would eventually lead to a NaN value returned for the current pixel sample’s radiance. Therefore, zero is explicitly returned for this case, as that is the value that  $D(\omega_h)$  converges to as  $\tan \theta_h$  goes to infinity.

```
{MicrofacetDistribution Method Definitions} ≡
    Float BeckmannDistribution::D(const Vector3f &wh) const {
        Float tan2Theta = Tan2Theta(wh);
        if (std::isinf(tan2Theta)) return 0.;
        Float cos4Theta = Cos2Theta(wh) * Cos2Theta(wh);
        return std::exp(-tan2Theta * (Cos2Phi(wh) / (alphax * alphax) +
                                       Sin2Phi(wh) / (alphay * alphay))) /
               (Pi * alphax * alphay * cos4Theta);
    }
```

Another useful microfacet distribution function is due to Trowbridge and Reitz (1975).<sup>8</sup> Its anisotropic variant is given by

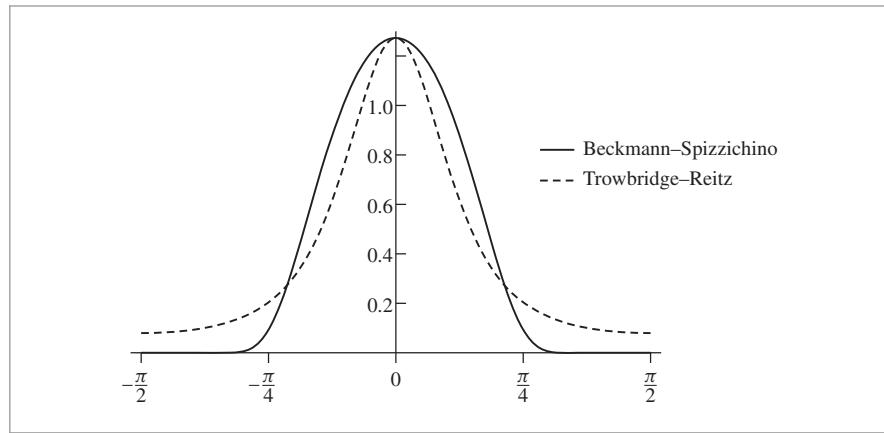
$$D(\omega_h) = \frac{1}{\pi \alpha_x \alpha_y \cos^4 \theta_h \left(1 + \tan^2 \theta_h (\cos^2 \phi_h / \alpha_x^2 + \sin^2 \phi_h / \alpha_y^2)\right)^2}. \quad [8.12]$$

In comparison to the Beckmann–Spizzichino model, Trowbridge–Reitz has higher tails—it falls off to zero more slowly for directions far from the surface normal. This characteristic matches the properties of many real-world surfaces well. See Figure 8.16 for a graph of these two microfacet distribution functions.

```
BeckmannDistribution::alphax
539
BeckmannDistribution::alphay
539
Cos2Phi() 511
Cos2Theta() 510
Float 1062
Pi 1063
Tan2Theta() 510
Vector3f 60
```

---

<sup>8</sup> This model was independently derived by Walter et al. (2007), who dubbed it “GGX.”



**Figure 8.16:** Graphs of isotropic Beckmann–Spizzichino and Trowbridge–Reitz microfacet distribution functions as a function of  $\theta$  for  $\alpha = 0.5$ . Note that Trowbridge–Reitz has higher tails for values of  $\theta$  with larger magnitudes.

*(MicrofacetDistribution Declarations)* +≡

```
class TrowbridgeReitzDistribution : public MicrofacetDistribution {
public:
    (TrowbridgeReitzDistribution Public Methods 540)
private:
    (TrowbridgeReitzDistribution Private Methods)
    (TrowbridgeReitzDistribution Private Data 540)
};
```

It can be convenient to specify the BRDF's roughness with a scalar parameter in  $[0, 1]$ , where values close to zero correspond to near-perfect specular reflection, rather than by specifying  $\alpha$  values directly. The `RoughnessToAlpha()` method, not included here, performs a mapping from such roughness values to  $\alpha$  values.

*(TrowbridgeReitzDistribution Public Methods)* ≡

540

```
static inline Float RoughnessToAlpha(Float roughness);
```

*(TrowbridgeReitzDistribution Private Data)* ≡

540

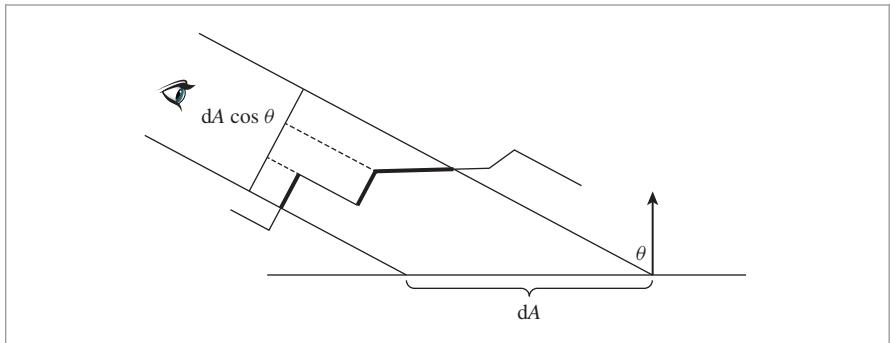
```
const Float alphax, alphay;
```

The `D()` method is a fairly direct transcription of Equation (8.12).

*(MicrofacetDistribution Method Definitions)* +≡

```
Float TrowbridgeReitzDistribution::D(const Vector3f &wh) const {
    Float tan2Theta = Tan2Theta(wh);
    if (std::isinf(tan2Theta)) return 0.0;
    const Float cos4Theta = Cos2Theta(wh) * Cos2Theta(wh);
    Float e = (Cos2Phi(wh) / (alphax * alphax) +
               Sin2Phi(wh) / (alphay * alphay)) * tan2Theta;
    return 1.0 / (Pi * alphax * alphay * cos4Theta * (1.0 + e) * (1.0 + e));
}
```

Cos2Phi() 511  
Cos2Theta() 510  
Float 1062  
MicrofacetDistribution 537  
Pi 1063  
Sin2Phi() 511  
Tan2Theta() 510  
Vector3f 60



**Figure 8.17:** As seen from a viewer or a light source, a differential area on the surface has area  $dA \cos \theta$ , where  $\cos \theta$  is the angle of the incident direction with the surface normal. The projected surface area of visible microfacets (thick lines) must be equal to  $dA \cos \theta$  as well; the masking-shadowing function  $G_1$  gives the fraction of the total microfacet area over  $dA$  that is visible in the given direction.

### 8.4.3 MASKING AND SHADOWING

The distribution of microfacet normals alone isn't enough to fully characterize the microsurface for rendering. It's also important to account for the fact that some microfacets will be invisible from a given viewing or illumination direction because they are back-facing (and thus, other microfacets are in front of them) and also for the fact that some of the forward-facing microfacet area will be hidden since it's shadowed by back-facing microfacets. These effects are accounted for by Smith's *masking-shadowing function*  $G_1(\omega, \omega_h)$ , which gives the fraction of microfacets with normal  $\omega_h$  that are visible from direction  $\omega$ . (Note that  $0 \leq G_1(\omega, \omega_h) \leq 1$ .) In the usual case where the probability a microfacet is visible is independent of its orientation  $\omega_h$ , we can write this function as  $G_1(\omega)$ .

As shown in Figure 8.17, a differential area  $dA$  on the surface has area  $dA \cos \theta$  when viewed from a direction  $\omega$  that makes an angle  $\theta$  with the surface normal. The area of visible microfacets seen from this direction must also be equal to  $dA \cos \theta$ , which leads to a normalization constraint for  $G_1$ :

$$\cos \theta = \int_{\mathcal{H}^2(\mathbf{n})} G_1(\omega, \omega_h) \max(0, \omega \cdot \omega_h) D(\omega_h) d\omega_h. \quad [8.13]$$

In other words, the projected area of visible microfacets for a given direction  $\omega$  must be equal to  $(\omega \cdot \mathbf{n}) = \cos \theta$  times the differential area of the macrosurface  $dA$ .

Because the microfacets form a heightfield, every backfacing microfacet shadows a forward-facing microfacet of equal projected area in the direction  $\omega$ . If  $A^+(\omega)$  is the projected area of forward-facing microfacets as seen from the direction  $\omega$  and  $A^-(\omega)$  is the projected area of backward-facing microfacets from Equation (8.13), then  $\cos \theta = A^+(\omega) - A^-(\omega)$ . We can thus alternatively write the masking-shadowing function as the ratio of visible microfacet area to total forward-facing microfacet area:

$$G_1(\omega) = \frac{A^+(\omega) - A^-(\omega)}{A^+(\omega)}.$$

Shadowing-masking functions are traditionally expressed in terms of an auxiliary function  $\Lambda(\omega)$ , which measures invisible masked microfacet area per visible microfacet area.

$$\Lambda(\omega) = \frac{A^-(\omega)}{A^+(\omega) - A^-(\omega)} = \frac{A^-(\omega)}{\cos \theta}. \quad [8.14]$$

The `Lambda()` method computes this function. Its implementation is specific to each microfacet distribution.

(*MicrofacetDistribution Public Methods*) +≡ 537  
`virtual float Lambda(const Vector3f &w) const = 0;`

Some algebra lets us express  $G_1(\omega)$  in terms of  $\Lambda(\omega)$ :

$$G_1(\omega) = \frac{1}{1 + \Lambda(\omega)},$$

and therefore we can provide a `G1()` method in terms of `Lambda()`.

(*MicrofacetDistribution Public Methods*) +≡ 537  
`float G1(const Vector3f &w) const {  
 return 1 / (1 + Lambda(w));  
}`

The microfacet distribution alone doesn't impose enough conditions to imply a specific  $\Lambda(\omega)$  function; many functions can fulfill the constraint in Equation (8.13). If we assume that there is no correlation between the heights of nearby points on the microsurface, for example, then it's possible to find a unique  $\Lambda(\omega)$  given  $D(\omega_h)$ . (For many microfacet models, a closed-form expression can be found.) Although the underlying assumption isn't true in reality—for actual microsurfaces, the height at a point is generally close to the heights of nearby points—the resulting  $\Lambda(\omega)$  functions turn out to be fairly accurate when compared to measured reflection from actual surfaces.

Under the assumption of no correlation of the heights of nearby points,  $\Lambda(\omega)$  for the isotropic Beckmann–Spizzichino distribution is

$$\Lambda(\omega) = \frac{1}{2} \left( \operatorname{erf}(a) - 1 + \frac{e^{-a^2}}{a\sqrt{\pi}} \right), \quad [8.15]$$

where  $a = 1/(\alpha \tan \theta)$  and  $\operatorname{erf}$  is the error function,  $\operatorname{erf}(x) = 2/\sqrt{\pi} \int_0^x e^{-x'^2} dx'$ .

`prbt`'s computation of the Beckmann–Spizzichino  $\Lambda(\omega)$  function is based on a rational polynomial approximation of Equation (8.15) that is much more efficient to evaluate because it avoids calling `std::erf()` and `std::exp()`, both of which are fairly expensive to evaluate.

Float 1062  
MicrofacetDistribution::  
Lambda()  
542  
Vector3f 60

```

⟨MicrofacetDistribution Method Definitions⟩ +≡
    Float BeckmannDistribution::Lambda(const Vector3f &w) const {
        Float absTanTheta = std::abs(TanTheta(w));
        if (std::isinf(absTanTheta)) return 0.;
        ⟨Compute alpha for direction w 543⟩
        Float a = 1 / (alpha * absTanTheta);
        if (a >= 1.6f)
            return 0;
        return (1 - 1.259f * a + 0.396f * a * a) /
            (3.535f * a + 2.181f * a * a);
    }
}

```

Masking-shadowing functions for anisotropic distributions are most easily computed by taking their corresponding isotropic function and stretching the underlying microsurface according to the  $\alpha_x$  and  $\alpha_y$  values. Equivalently, one can compute an interpolated  $\alpha$  value for the direction of interest and use that with the isotropic function; see the “Further Reading” section at the end of this chapter for more details.

```

⟨Compute alpha for direction w⟩ ≡
    Float alpha = std::sqrt(Cos2Phi(w) * alphax * alphax +
                           Sin2Phi(w) * alphay * alphay);

```

543

Under the uncorrelated height assumption, the form of  $\Lambda(\omega)$  for the Trowbridge–Reitz distribution is quite simple:

$$\Lambda(\omega) = \frac{-1 + \sqrt{1 + \alpha^2 \tan^2 \theta}}{2}.$$

```

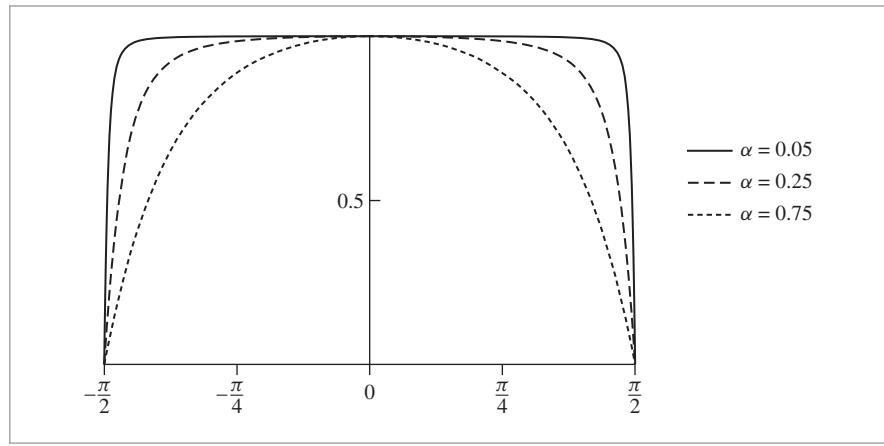
⟨MicrofacetDistribution Method Definitions⟩ +≡
    Float TrowbridgeReitzDistribution::Lambda(const Vector3f &w) const {
        Float absTanTheta = std::abs(TanTheta(w));
        if (std::isinf(absTanTheta)) return 0.;
        ⟨Compute alpha for direction w 543⟩
        Float alpha2Tan2Theta = (alpha * absTanTheta) * (alpha * absTanTheta);
        return (-1 + std::sqrt(1.f + alpha2Tan2Theta)) / 2;
    }
}

```

Figure 8.18 shows a plot of the Trowbridge–Reitz  $G_1(\omega)$  function for a few values of  $\alpha$ . Note that the function is close to one over much of the domain but falls to zero at grazing angles. Note also that increasing surface roughness (i.e., higher values of  $\alpha$ ) causes the function to fall off more quickly.

BeckmannDistribution::alphax  
539  
BeckmannDistribution::alphay  
539  
Cos2Phi() 511  
Float 1062  
Sin2Phi() 511  
TanTheta() 510  
Vector3f 60

One last useful function related to the geometric properties of a microfacet distribution is  $G(\omega_o, \omega_i)$ , which gives the fraction of microfacets in a differential area that are visible from both directions  $\omega_o$  and  $\omega_i$ . Defining  $G$  requires some additional assumptions. For starters, we know that  $G_1(\omega_o)$  gives the fraction of microfacets that are visible from the direction  $\omega_o$  and  $G_1(\omega_i)$  gives the fraction for  $\omega_i$ . If we assume that the probability of a microfacet being visible from both directions is the probability that it is visible from each



**Figure 8.18: The Masking-Shadowing Function  $G_1(\omega)$  for the Trowbridge-Reitz Distribution.**  
Increasing surface roughness (higher  $\alpha$  values) cause the function to fall off to zero more quickly.

direction independently, then we have

$$G(\omega_o, \omega_i) = G_1(\omega_o) G_1(\omega_i).$$

In practice, however, these probabilities aren't independent, and this formulation underestimates  $G$ . To see why, consider the case where  $\omega_o = \omega_i$ ; in this case any microfacet that is visible from  $\omega_o$  is also visible from  $\omega_i$ , and so  $G(\omega_o, \omega_i) = G_1(\omega_o) = G_1(\omega_i)$ . Because  $G_1(\omega) \leq 1$ , their product in this case will cause  $G(\omega_o, \omega_i)$  to be too small (unless  $G_1(\omega) = 1$ , which is usually only true if  $\omega = (0, 0, 1)$ ). More generally, the closer together the two directions are, the more correlation there is between  $G_1(\omega_o)$  and  $G_1(\omega_i)$ .

A more accurate model can be derived assuming that microfacet visibility is more likely the higher up a given point on a microfacet is. This assumption leads to the model

$$G(\omega_o, \omega_i) = \frac{1}{1 + \Lambda(\omega_o) + \Lambda(\omega_i)}.$$

This approximation is fairly accurate in practice and is the one we'll use in pbrt. See the “Further Reading” section at the end of this chapter for pointers to information about this function's derivation as well as more sophisticated approaches to defining  $G(\omega_o, \omega_i)$  functions.

*(MicrofacetDistribution Public Methods)*  $\equiv$

537

```
Float G(const Vector3f &wo, const Vector3f &wi) const {
    return 1 / (1 + Lambda(wo) + Lambda(wi));
}
```

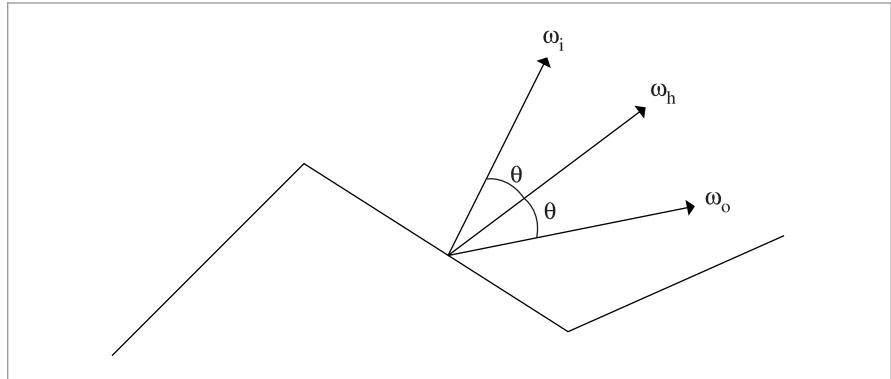
Float 1062

MicrofacetDistribution::  
Lambda()  
542

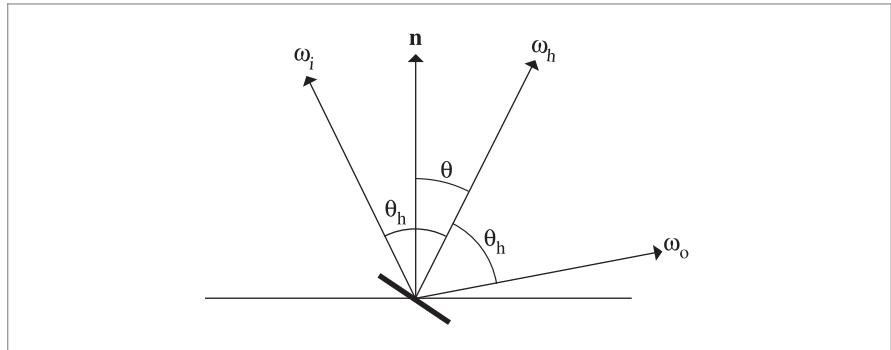
Vector3f 60

#### 8.4.4 THE TORRANCE-SPARROW MODEL

An early microfacet model was developed by Torrance and Sparrow (1967) to model metallic surfaces. They modeled surfaces as collections of perfectly smooth mirrored microfacets. Because the microfacets are perfectly specular, only those with a normal



**Figure 8.19:** For perfectly specular microfacets and a given pair of directions  $\omega_i$  and  $\omega_o$ , only those microfacets with normal  $\omega_h = \widehat{\omega_i + \omega_o}$  reflect light from  $\omega_i$  to  $\omega_o$ .



**Figure 8.20:** Setting for the Derivation of the Torrance–Sparrow Model. For directions  $\omega_i$  and  $\omega_o$ , only microfacets with normal  $\omega_h$  reflect light. The angle between  $\omega_h$  and  $n$  is denoted by  $\theta$ , and the angle between  $\omega_h$  and  $\omega_o$  is denoted by  $\theta_h$ . (The angle between  $\omega_h$  and  $\omega_i$  is also necessarily  $\theta_h$ .)

equal to the *half-angle vector*,

$$\omega_h = \widehat{\omega_i + \omega_o},$$

cause perfect specular reflection from  $\omega_i$  to  $\omega_o$  (Figure 8.19).

The derivation of the Torrance–Sparrow model has a number of interesting steps; we'll go through it in detail here. First, consider the differential flux  $d\Phi_h$  incident on the microfacets oriented with half-angle  $\omega_h$  for directions  $\omega_i$  and  $\omega_o$ . From the definition of radiance, Equation (5.2), it is

$$d\Phi_h = L_i(\omega_i) d\omega dA^\perp(\omega_h) = L_i(\omega_i) d\omega \cos \theta_h dA(\omega_h),$$

where we have written  $dA(\omega_h)$  for the area measure of the microfacets with orientation  $\omega_h$  and  $\cos \theta_h$  for the cosine of the angle between  $\omega_i$  and  $\omega_h$  (Figure 8.20).

The differential area of microfacets with orientation  $\omega_h$  is

$$dA(\omega_h) = D(\omega_h) d\omega_h dA.$$

The first two terms of this product describe the differential area of facets per unit area that have the proper orientation, and the  $dA$  term converts this to differential area.

Therefore,

$$d\Phi_h = L_i(\omega_i) d\omega \cos \theta_h D(\omega_h) d\omega_h dA. \quad [8.16]$$

If we assume that the microfacets individually reflect light according to Fresnel's law, the outgoing flux is

$$d\Phi_o = F_r(\omega_o) d\Phi_h. \quad [8.17]$$

Again using the definition of radiance, the reflected outgoing radiance is

$$L(\omega_o) = \frac{d\Phi_o}{d\omega_o \cos \theta_o dA}.$$

If we substitute Equation (8.17) into this and then Equation (8.16) into the result, we have

$$L(\omega_o) = \frac{F_r(\omega_o) L_i(\omega_i) d\omega_i D(\omega_h) d\omega_h dA \cos \theta_h}{d\omega_o dA \cos \theta_o}.$$

In Section 14.1.1, we will derive an important relationship between  $d\omega_h$  and  $d\omega_o$  under specular reflection:

$$d\omega_h = \frac{d\omega_o}{4 \cos \theta_h}. \quad [8.18]$$

We can substitute this relationship into the previous equation and simplify, giving

$$L(\omega_o) = \frac{F_r(\omega_o) L_i(\omega_i) D(\omega_h) d\omega_i}{4 \cos \theta_o}.$$

We can now apply the definition of the BRDF, Equation (5.8) and add the geometric attenuation term  $G(\omega_o, \omega_i)$ , which gives us the Torrance–Sparrow BRDF:

$$f_r(\omega_o, \omega_i) = \frac{D(\omega_h) G(\omega_o, \omega_i) F_r(\omega_o)}{4 \cos \theta_o \cos \theta_i}. \quad [8.19]$$

One of the nice things about the Torrance–Sparrow model is that the derivation doesn't depend on the particular microfacet distribution being used. Furthermore, it doesn't depend on a particular Fresnel function, so it can be used for both conductors and dielectrics. However, the relationship between  $d\omega_h$  and  $d\omega_o$  used in the derivation does depend on the assumption of specular reflection from microfacets.

`MicrofacetReflection` uses the Torrance–Sparrow model to implement a general microfacet-based BRDF.

```
<BxDF Declarations> +≡
    class MicrofacetReflection : public BxDF {
public:
    <MicrofacetReflection Public Methods 547>
private:
    <MicrofacetReflection Private Data 547>
};
```

Its constructor takes the reflectance, a pointer to a MicrofacetDistribution implementation, and a Fresnel function.

```
<MicrofacetReflection Public Methods> ≡ 547
    MicrofacetReflection(const Spectrum &R,
                        MicrofacetDistribution *distribution, Fresnel *fresnel)
    : BxDF(BxDFType(BSDF_REFLECTION | BSDF_GLOSSY)), R(R),
      distribution(distribution), fresnel(fresnel) { }
```

```
<MicrofacetReflection Private Data> ≡ 547
    const Spectrum R;
    const MicrofacetDistribution *distribution;
    const Fresnel *fresnel;
```

Evaluating the terms of the Torrance–Sparrow BRDF is straightforward. For the Fresnel term, recall that given specular reflection, the angle  $\theta_h$  is the same between  $\omega_h$  and both  $\omega_i$  and  $\omega_o$ , so it doesn't matter which vector we use to compute the cosine of  $\theta_h$ . We arbitrarily choose  $\omega_i$ .

```
AbsCosTheta() 510
BSDF_GLOSSY 513
BSDF_REFLECTION 513
BxDF 513
BxDFType 513
Dot() 63
Float 1062
Fresnel 521
Fresnel::Evaluate() 522
MicrofacetDistribution 537
MicrofacetDistribution::D() 538
MicrofacetDistribution::G() 544
MicrofacetReflection 547
MicrofacetReflection::
    distribution 547
    MicrofacetReflection::fresnel 547
MicrofacetReflection::R 547
Spectrum 315
Vector3f::Normalize() 66
Vector3f 60

<Handle degenerate cases for microfacet reflection> ≡ 547
    if (cosThetaI == 0 || cosTheta0 == 0) return Spectrum(0.);
    if (wh.x == 0 && wh.y == 0 && wh.z == 0) return Spectrum(0.);
```

Two edge cases that come up with incident and outgoing directions at glancing angles need to be handled explicitly to avoid NaN values being generated from the evaluation of the BRDF.

*<Handle degenerate cases for microfacet reflection> ≡*

```
    if (cosThetaI == 0 || cosTheta0 == 0) return Spectrum(0.);
    if (wh.x == 0 && wh.y == 0 && wh.z == 0) return Spectrum(0.);
```

It's also possible to define a BTDF for transmission through microfacets that exhibit perfect specular transmission. In that setting, with transmission from a medium with

index of refraction  $\eta_i$  to a medium with index of refraction  $\eta_t$ , then  $d\omega_h$  and  $d\omega_o$  are related by:

$$d\omega_h = \frac{\eta_o^2 |\omega_o \cdot \omega_h| d\omega_o}{(\eta_i(\omega_i \cdot \omega_h) + \eta_o(\omega_o \cdot \omega_h))^2}$$

This relationship can be used in place of Equation (8.18) in the derivation of the Torrance–Sparrow BRDF. The result is

$$f_r(\omega_o, \omega_i) = \frac{\eta^2 D(\omega_h) G(\omega_o, \omega_i) (1 - F_r(\omega_o))}{((\omega_o \cdot \omega_h) + \eta(\omega_i \cdot \omega_h))^2} \frac{|\omega_i \cdot \omega_h| |\omega_o \cdot \omega_h|}{\cos \theta_o \cos \theta_i}, \quad [8.20]$$

where  $\eta = \eta_i/\eta_o$ . For specular transmission, the half-angle vector is

$$\omega_h = \omega_o + \eta \omega_i.$$

(You may want to verify that this normal vector causes  $\omega_o$  to be refracted in the direction  $\omega_i$ , via Equation (8.9).)

The `MicrofacetTransmission` class implements this BTDF.

```
(BxDF Declarations) +≡
class MicrofacetTransmission : public BxDF {
public:
    (MicrofacetTransmission Public Methods 548)
private:
    (MicrofacetTransmission Private Data 548)
};

(MicrofacetTransmission Public Methods) ≡ 548
MicrofacetTransmission(const Spectrum &T,
    MicrofacetDistribution *distribution, Float etaA, Float etaB,
    TransportMode mode)
: BxDF(BxDFType(BSDF_TRANSMISSION | BSDF_GLOSSY)),
    T(T), distribution(distribution), etaA(etaA), etaB(etaB),
    fresnel(etaA, etaB), mode(mode) { }

(MicrofacetTransmission Private Data) ≡ 548
const Spectrum T;
const MicrofacetDistribution *distribution;
const Float etaA, etaB;
const FresnelDielectric fresnel;
const TransportMode mode;

Its f() method is a direct transcription of Equation (8.20). Its implementation is therefore not included here.

(MicrofacetTransmission Public Methods) +≡ 548
Spectrum f(const Vector3f &wo, const Vector3f &wi) const;
```

Figure 8.21 shows the dragon rendered with the Torrance–Sparrow model and both reflection and transmission and Figure 8.22 compares the appearance of two spheres with an isotropic and anisotropic microfacet model lit by a light source simulating a distant environment.

BSDF\_GLOSSY 513  
 BSDF\_TRANSMISSION 513  
 BxDF 513  
 BxDFType 513  
 Float 1062  
 FresnelDielectric 522  
 MicrofacetDistribution 537  
 MicrofacetTransmission 548  
 MicrofacetTransmission::distribution 548  
 MicrofacetTransmission::etaA 548  
 MicrofacetTransmission::etaB 548  
 MicrofacetTransmission::fresnel 548  
 MicrofacetTransmission::mode 548  
 MicrofacetTransmission::T 548  
 Spectrum 315  
 TransportMode 960  
 Vector3f 60



(a)

(b)

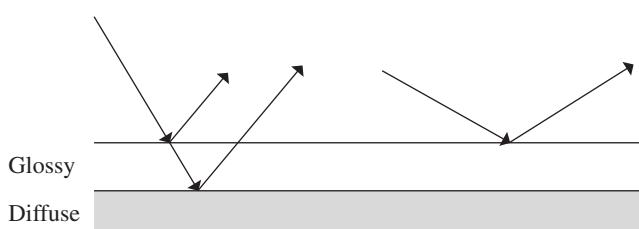
**Figure 8.21:** Dragon models rendered with the Torrance–Sparrow microfacet model featuring both reflection (a) and transmission (b). (*Model courtesy of Christian Schüller.*)



**Figure 8.22:** Spheres rendered with an isotropic microfacet distribution (left) and an anisotropic distribution (right). Note the different specular highlight shapes from the anisotropic model. We have used spheres here instead of the dragon, since anisotropic models like these depend on a globally consistent set of tangent vectors over the surface to orient the direction of anisotropy in a reasonable way.

## 8.5 FRESNEL INCIDENCE EFFECTS

Many BRDF models in graphics do not account for the fact that Fresnel reflection reduces the amount of light that reaches the bottom level of layered objects. Consider a polished wood table or a wall with glossy paint: if you look at their surfaces head-on, you primarily



**Figure 8.23:** The FresnelBlend BRDF models the effect of a surface with a glossy layer on top of a diffuse substrate. As the angle of incidence of the vectors  $\omega_i$  and  $\omega_o$  heads toward glancing (right), the amount of light that reaches the diffuse substrate is reduced by Fresnel effects, and the diffuse layer becomes less visibly apparent.

see the wood or the paint pigment color. As you move your viewpoint toward a glancing angle, you see less of the underlying color as it is overwhelmed by increasing glossy reflection due to Fresnel effects.

In this section, we will implement a BRDF model developed by Ashikhmin and Shirley (2000, 2002) that models a diffuse underlying surface with a glossy specular surface above it. The effect of reflection from the diffuse surface is modulated by the amount of energy left after Fresnel effects have been considered. Figure 8.23 shows this idea: when the incident direction is close to the normal, most light is transmitted to the diffuse layer and the diffuse term dominates. When the incident direction is close to glancing, glossy reflection is the primary mode of reflection. The car model in Figures 12.19 and 12.20 uses this BRDF for its paint.

```
(BxDF Declarations) +≡
class FresnelBlend : public BxDF {
public:
    (FresnelBlend Public Methods 551)
private:
    (FresnelBlend Private Data 550)
};
```

The model takes two spectra, representing diffuse and specular reflectance, and a microfacet distribution for the glossy layer.

```
(BxDF Method Definitions) +≡
FresnelBlend::FresnelBlend(const Spectrum &Rd, const Spectrum &Rs,
                           MicrofacetDistribution *distribution)
    : BxDF(BxDFType(BSDF_REFLECTION | BSDF_GLOSSY)),
      Rd(Rd), Rs(Rs), distribution(distribution) { }
```

```
(FresnelBlend Private Data) ≡
const Spectrum Rd, Rs;
MicrofacetDistribution *distribution;
```

<b>BSDF_GLOSSY</b> 513 <b>BSDF_REFLECTION</b> 513 <b>BxDF</b> 513 <b>BxDFType</b> 513 <b>FresnelBlend</b> 550 <b>FresnelBlend::distribution</b> 550 <b>FresnelBlend::Rd</b> 550 <b>FresnelBlend::Rs</b> 550 <b>MicrofacetDistribution</b> 537 <b>Spectrum</b> 315	<b>550</b>
--	------------

This model is based on the weighted sum of a glossy specular term and a diffuse term. Accounting for reciprocity and energy conservation, the glossy specular term is derived as

$$f_r(p, \omega_o, \omega_i) = \frac{D(\omega_h)F(\omega_o)}{4(\omega_h \cdot \omega_i)(\max((\mathbf{n} \cdot \omega_o), (\mathbf{n} \cdot \omega_i)))},$$

where  $D(\omega_h)$  is a microfacet distribution term and  $F(\omega_o)$  represents Fresnel reflectance. Note that this is quite similar to the Torrance–Sparrow model.

The key to Ashikhmin and Shirley's model is the derivation of a diffuse term such that the model still obeys reciprocity and conserves energy. The derivation is dependent on an approximation to the Fresnel reflection equations due to Schlick (1993), who approximated Fresnel reflection as

$$F_r(\cos \theta) = R + (1 - R)(1 - \cos \theta)^5,$$

where  $R$  is the reflectance of the surface at normal incidence.

Given this Fresnel term, the diffuse term in the following equation successfully models Fresnel-based reduced diffuse reflection in a physically plausible manner:

$$f_r(p, \omega_i, \omega_o) = \frac{28R_d}{23\pi}(1 - R_s) \left(1 - \left(1 - \frac{(\mathbf{n} \cdot \omega_i)}{2}\right)^5\right) \left(1 - \left(1 - \frac{(\mathbf{n} \cdot \omega_o)}{2}\right)^5\right).$$

We will not include the derivation of this result here.

```
(FresnelBlend Public Methods) ≡ 550
    Spectrum SchlickFresnel(Float cosTheta) const {
        auto pow5 = [] (Float v) { return (v * v) * (v * v) * v; };
        return Rs + pow5(1 - cosTheta) * (Spectrum(1.) - Rs);
    }

(BxDF Method Definitions) +≡
    Spectrum FresnelBlend::f(const Vector3f &wo, const Vector3f &wi) const {
        auto pow5 = [] (Float v) { return (v * v) * (v * v) * v; };
        Spectrum diffuse = (28.f/(23.f*Pi)) * Rd * 
            (Spectrum(1.f) - Rs) *
            (1 - pow5(1 - .5f * AbsCosTheta(wi))) *
            (1 - pow5(1 - .5f * AbsCosTheta(wo)));
        Vector3f wh = wi + wo;
        if (wh.x == 0 && wh.y == 0 && wh.z == 0) return Spectrum(0);
        wh = Normalize(wh);
        Spectrum specular = distribution->D(wh) /
            (4 * AbsDot(wi, wh) *
            std::max(AbsCosTheta(wi), AbsCosTheta(wo))) *
            SchlickFresnel(Dot(wi, wh));
        return diffuse + specular;
    }
```



**Figure 8.24:** Dragon models rendered using the FourierBSDF model. The surface of the dragon on the left has a BSDF that models the appearance of rough gold; the one on the right is coated copper. (Model courtesy of Christian Schüller.)

## 8.6 FOURIER BASIS BSDFs

While reflection models like Torrance–Sparrow and Oren–Nayar can accurately represent many materials, some materials have BRDF shapes that don’t match these models well. (Examples include layered materials like metals with smooth or rough coatings or fabrics, which are often partially retro-reflective.) One option for materials like these is to store their BSDF values in a large 3D or 4D lookup table, though this approach can require an unacceptable amount of storage—for example, if  $\omega_i$  and  $\omega_o$  are sampled in spherical angles with 1-degree spacing, then over one billion sample points are needed to represent the corresponding anisotropic BSDF in the form of a 4D lookup table.

Therefore, having a more compact representation that still represents the BSDF accurately is highly desirable. This section introduces the FourierBSDF, which represents BSDFs with sums of scaled cosine terms using the Fourier basis. This representation is accurate, space-efficient, and works well with Monte Carlo integration (see Section 14.1.4.) Figure 8.24 shows two instances of the dragon model rendered using this representation.

Here, we won’t discuss how BSDFs are transformed into this representation, but we will focus on its use in rendering. See the “Further Reading” section at the end of this chapter for pointers to more details on those issues and the `scenes/brdfs` directory in the `pbrt` distribution for a variety of BSDFs represented in this format.

The FourierBSDF represents isotropic BSDFs by parameterizing the BSDF by a pair of spherical coordinates for the incident and outgoing directions, where  $\mu_i$  and  $\mu_o$  denote the cosines of the incident and outgoing zenith angles, respectively, and  $\phi_i$  and  $\phi_o$  are the azimuth angles:

$$f(\omega_i, \omega_o) = f(\mu_i, \phi_i, \mu_o, \phi_o).$$

The assumption of isotropy means that the function can be rewritten with a simpler dependence on the zenith angle cosines and the azimuth difference angle  $\phi = \phi_i - \phi_o$ :

$$f(\omega_i, \omega_o) = f(\mu_i, \mu_o, \phi_i - \phi_o) = f(\mu_i, \mu_o, \phi).$$

Isotropic BSDFs are generally also *even* functions of the azimuth difference, i.e.:

$$f(\mu_i, \mu_o, \phi) = f(\mu_i, \mu_o, -\phi). \quad [8.21]$$

Given these properties, the product of the BSDF with the cosine falloff is then represented as a Fourier series in the azimuth angle difference.

$$f(\mu_i, \mu_o, \phi_i - \phi_o) |\mu_i| = \sum_{k=0}^{m-1} a_k(\mu_i, \mu_o) \cos(k (\phi_i - \phi_o)) \quad [8.22]$$

Note how only cosine terms and no sine terms are needed due to Equation (8.21). The function evaluations  $a_0(\mu_i, \mu_o), \dots, a_{m-1}(\mu_i, \mu_o)$  denote the Fourier coefficients for a specific pair of zenith angle cosines.

Next, the functions  $a_k$  are discretized over their input arguments. We choose a set of zenith angle cosines  $\bar{\mu} = \{\mu_0, \dots, \mu_{n-1}\}$  and store the values of  $a_k(\mu_i, \mu_j)$  for every pair  $0 \leq i, j < n$ . Thus, we can think of each  $a_k$  as a  $n \times n$  matrix, and the entire BRDF representation then consists of a set of  $m$  such matrices. Each describes a different azimuthal oscillation frequency in the material's response to incident illumination.

The maximum order  $m$  needed to evaluate Equation (8.22) to satisfactory accuracy varies: it depends on the particular zenith angles, so it's worth adapting the number of coefficients  $a_k$  to the complexity of the BSDF for a given pair of directions. Doing this is very important for the compactness of this representation.

To see the value of being able to vary the number of coefficients, consider nearly perfect specular reflection: when  $\mu_i \approx \mu_o$ , many coefficients are necessary to accurately represent the specular lobe, which is zero for almost all azimuth angle differences  $\phi = \phi_i - \phi_o$  and then very large for a small set of directions around  $\phi = \pi$ , where the incident and outgoing directions are nearly opposite. However, when  $\mu_i$  and  $\mu_o$  aren't aligned, only a single term is needed to represent that the BSDF is zero (or has negligible value).<sup>9</sup>

For smoother BSDFs, most or all pairs of  $\mu_i$  and  $\mu_o$  angles require multiple coefficients  $a_k$  to represent their  $\phi$  distribution accurately, but their smoothness means that not too many coefficients are generally needed for each  $a_k$ . The FourierBSDF representation exploits this property and only stores the sparse set of coefficients that is needed to achieve a desired accuracy. Thus, for most types of realistic BSDF data, the representation of Equation (8.22) is fairly compact; a few megabytes is typical.

`FourierBSDFTable` is a helper structure that holds all of the data for a BSDF represented in this manner. It's mostly a simple struct that collects data that's directly accessible to calling code, though it does provide a few utility methods.

---

<sup>9</sup> Jakob et al. (2014a) showed that this adaptivity makes it possible to represent a shiny mirror with Beckmann roughness  $\alpha = 0.01$  with 1% relative  $L^2$  error using 51 MiB, while using the maximum order  $m$  needed for any pair of directions for all pairs of directions would require 28 GiB to achieve the same error.

```
{BSDF Declarations} +≡
struct FourierBSDFTable {
    (FourierBSDFTable Public Data 554)
    (FourierBSDFTable Public Methods 554)
};
```

The `Read()` method initializes the structure for the BSDF stored in the given file. It returns `true` on success or `false` if an error was encountered while reading the file.

```
(FourierBSDFTable Public Methods) ≡ 554
static bool Read(const std::string &filename, FourierBSDFTable *table);
```

If the BSDF represents scattering at the boundary between two different media, then the `FourierBSDFTable::eta` member variable gives the relative index of refraction over the surface boundary (Section 8.2.3) `mMax` gives the maximum order  $m$  for any pair of  $\mu_i$ ,  $\mu_o$  directions; this upper limit is useful when allocating temporary buffers to store  $a_k$  coefficients, for example.

```
(FourierBSDFTable Public Data) ≡ 554
Float eta;
int mMax;
```

`nChannels` gives the number of spectral channels available; in this implementation, it is either 1, representing a monochromatic BSDF, or 3, representing a BSDF with RGB colors. Here, the three-channel variant actually stores luminance, red, and blue, rather than red, green, and blue—representing luminance directly turns out to be useful for the Monte Carlo sampling routines defined in Section 14.1.4, since it provides aggregate information about the function over all color channels. The corresponding green color is easily computed from luminance, red, and blue, as we'll see shortly.

```
(FourierBSDFTable Public Data) +≡ 554
int nChannels;
```

The zenith angles are discretized into `nMu` directions, which are stored in the `mu` array. `mu` is sorted from low to high, so that binary search can be used to find the entry that's closest to a given  $\mu_i$  or  $\mu_o$  angle.

```
(FourierBSDFTable Public Data) +≡ 554
int nMu;
Float *mu;
```

To evaluate Equation (8.22), we need to know the target Fourier order  $m$  and all coefficients  $a_0, \dots, a_{m-1}$  corresponding to the directions  $\omega_i$  and  $\omega_o$ . For simplicity now, we'll present the basic ideas as if only the coefficients for the closest `mu` directions less than or equal to  $\mu_i$  and  $\mu_o$  are used, though the implementation to follow interpolates between coefficients from multiple `mu` values around the directions.

The order  $m$  of the Fourier representation is always bounded by `mMax` but varies with respect to the incident and outgoing zenith angle cosine  $\mu_i$  and  $\mu_o$ : how many orders are needed can be determined by querying an `nMu × nMu` integer matrix `m`.

Float 1062  
FourierBSDFTable 554

```
<FourierBSDFTable Public Data> +≡
    int *m;
```

To find  $m$  for a particular set of angles, we first perform two binary searches in the discretized directions  $\mu$  to give the offsets  $oi$  and  $oo$  such that

$$\begin{aligned} \mu[oi] &\leq \mu_i < \mu[oi + 1] \\ \mu[oo] &\leq \mu_o < \mu[oo + 1] \end{aligned}$$

Using these indices, the requisite order can be fetched from  $m[oo * nMu + oi]$ .

All of the  $a_k$  coefficients for all of the pairs of discretized directions  $\mu$  are packed into the  $a$  array. Because the maximum order (and thus, number of coefficients) varies and can even be zero depending on the characteristics of the BSDF for a given pair of directions, finding the offset into the  $a$  array is a two-step process:

1. First, the offsets  $oi$  and  $oo$  are used to index into the  $aOffset$  array to get an offset into  $a$ :  $offset = aOffset[oo * nMu + oi]$ . (The  $aOffset$  array thus has a total of  $nMu * nMu$  entries.)
2. Next, the  $m$  coefficients starting at  $a[offset]$  give the  $a_k$  values for the corresponding pair of directions. For the three color channel case, the first  $m$  coefficients after  $a[offset]$  encode coefficients for luminance, the next  $m$  correspond to the red channel, and then blue follows.

```
<FourierBSDFTable Public Data> +≡
    int *aOffset;
    float *a;
```

`GetAk()` is a small convenience method that, given offsets into the  $\mu$  array for the incident and outgoing direction cosines, returns the order  $m$  of coefficients for the directions and a pointer to their coefficients.

```
<FourierBSDFTable Public Methods> +≡
    const float *GetAk(int offsetI, int offset0, int *mptr) const {
        *mptr = m[offset0 * nMu + offsetI];
        return a + aOffset[offset0 * nMu + offsetI];
    }
```

BxDF 513  
 float 1062  
 FourierBSDFTable 554  
 FourierBSDFTable::a 555  
 FourierBSDFTable::aOffset 555  
 FourierBSDFTable::m 555  
 FourierBSDFTable::nMu 554  
 FourierMaterial 583

The `FourierBSDF` class provides a bridge between the `FourierBSDFTable` representation and the `BxDF` interface. Instances of this class are created by the `FourierMaterial` class.

```
<BxDF Declarations> +≡
    class FourierBSDF : public BxDF {
    public:
        <FourierBSDF Public Methods 556>
    private:
        <FourierBSDF Private Data 556>
    };
```

*(FourierBSDF Public Methods)  $\equiv$*

555

```
FourierBSDF(const FourierBSDFTable &bsdfTable, TransportMode mode)
: BxDF(BxDFType(BSDF_REFLECTION | BSDF_TRANSMISSION | BSDF_GLOSSY)),
  bsdfTable(bsdfTable), mode(mode) { }
```

The FourierBSDF class stores only a const reference to the table; the table is large enough that we definitely don't want to make a separate copy of it for each FourierBSDF instance. Only read-access is needed here, so this approach doesn't cause any problems. (FourierMaterial is responsible for the FourierBSDFTable storage.)

*(FourierBSDF Private Data)  $\equiv$*

555

```
const FourierBSDFTable &bsdfTable;
const TransportMode mode;
```

Evaluating the BSDF is a matter of computing the cosines  $\mu_i$  and  $\mu_o$ , finding the corresponding coefficients  $a_k$  and maximum order, and then evaluating Equation (8.22).

*(BxDF Method Definitions)  $+ \equiv$*

```
Spectrum FourierBSDF::f(const Vector3f &wo, const Vector3f &wi) const {
    (Find the zenith angle cosines and azimuth difference angle 556)
    (Compute Fourier coefficients  $a_k$  for  $(\mu_i, \mu_o)$  557)
    (Evaluate Fourier expansion for angle  $\phi$  558)
}
```

There is an important difference of convention in how directions are represented within the FourierBSDF: the incident direction  $\omega_i$  is negated compared to the usual approach in pbrt. This difference is helpful when performing other computations such as computing BSDFs for layered materials using this representation.<sup>10</sup>

*(Find the zenith angle cosines and azimuth difference angle)  $\equiv$*

556, 821

```
Float muI = CosTheta(-wi), mu0 = CosTheta(wo);
Float cosPhi = CosDPhi(-wi, wo);
```

So that the reconstructed BSDF is fairly smooth, the implementation here interpolates  $a_k$  coefficients over the product of the four quantized  $\mu$  directions that surround  $\mu_i$  and the four that surround  $\mu_o$ . The interpolation is performed with a *tensor-product spline*, where weights for the sampled function values are computed separately for each parameter and then multiplied together. Each final Fourier coefficient  $a_k$  is thus computed by

$$a_k = \sum_{a=0}^3 \sum_{b=0}^3 a_k(o_i + a, o_o + b) w_i(a) w_o(b), \quad [8.23]$$

where  $a_k(i, j)$  gives the  $k$ th Fourier coefficient for the quantized directions  $\mu_i$ ,  $\mu_j$  and  $w_i$  and  $w_o$  are the spline weights. This interpolation ensures sufficient smoothness even

<sup>10</sup> For example, an implication of this convention is that for light passing unchanged through a medium, if we consider  $a_k(\mu_i, \mu_o)$  as a matrix, then we have a diagonal matrix where the non-zero entries are the Fourier coefficients corresponding to a delta distribution that is zero for all  $\phi \neq 0$ . This property in turn makes the notation for these sorts of computations easier to work with.

BSDF\_GLOSSY 513  
 BSDF\_REFLECTION 513  
 BSDF\_TRANSMISSION 513  
 BxDF 513  
 BxDFType 513  
 CosDPhi() 512  
 CosTheta() 510  
 Float 1062  
 FourierBSDF 555  
 FourierBSDF::bsdfTable 556  
 FourierBSDF::mode 556  
 FourierBSDFTable 554  
 FourierMaterial 583  
 Spectrum 315  
 TransportMode 960  
 Vector3f 60

when the discretization of directions  $\mu_i$  is relatively coarse; the details of how these weights are computed will be explained in a few pages.

```
(Compute Fourier coefficients  $a_k$  for  $(\mu_i, \mu_o)$ ) ≡ 556, 819
  (Determine offsets and weights for  $\mu_i$  and  $\mu_o$  557)
  (Allocate storage to accumulate  $a_k$  coefficients 557)
  (Accumulate weighted sums of nearby  $a_k$  coefficients 557)
```

For each direction  $\mu_i$  and  $\mu_o$ , `GetWeightsAndOffset()` returns the offset of the first of the four `mu` values to be interpolated over and an array of four floating-point weights.

```
(Determine offsets and weights for  $\mu_i$  and  $\mu_o$ ) ≡ 557
int offsetI, offset0;
Float weightsI[4], weights0[4];
if (!bsdfTable.GetWeightsAndOffset(muI, &offsetI, weightsI) ||
    !bsdfTable.GetWeightsAndOffset(mu0, &offset0, weights0))
    return Spectrum(0.f);
```

The various  $a_k$  vectors in the  $4 \times 4$  extent of the directions being interpolated over may have different orders  $m$ . Therefore, the implementation here allocates storage for the  $a_k$  values using the maximum possible order  $m$  times the number of channels for the size. For the multiple-channel case, the first `bsdfTable.mMax` entries of the `ak` array allocated here will be used for the first channel, the next `mMax` are for the second channel, and so forth. (Thus there is generally some unused space in the `ak` array for the usual case that the maximum order over the sixteen directions is less than `mMax`.) All of this storage is initialized to zero, so that subsequent code can add terms of Equation (8.23) to the corresponding entry in `ak` directly.

```
(Allocate storage to accumulate  $a_k$  coefficients) ≡ 557
Float *ak = ALLOCA(Float, bsdfTable.mMax * bsdfTable.nChannels);
memset(ak, 0, bsdfTable.mMax * bsdfTable.nChannels * sizeof(Float));
```

Given weights, offsets, and storage for the result, the interpolation can now be performed.

```
(Accumulate weighted sums of nearby  $a_k$  coefficients) ≡ 557
int mMax = 0;
for (int b = 0; b < 4; ++b) {
    for (int a = 0; a < 4; ++a) {
        (Add contribution of (a, b) to  $a_k$  values 558)
    }
}
```

ALLOCA() 1071  
 Float 1062  
`FourierBSDFTable::`  
`GetWeightsAndOffset()` 563  
`FourierBSDFTable::mMax` 554  
`FourierBSDFTable::nChannels` 554  
`Spectrum` 315

Given the weights and the starting offsets, adding each term of the sum in Equation (8.23) is a matter of getting the corresponding coefficients from the table for the offset and adding them to the running sum in `ak`.

```
(Add contribution of (a, b) to  $a_k$  values) ≡ 557
    Float weight = weightsI[a] * weights0[b];
    if (weight != 0) {
        int m;
        const Float *ap = bsdfTable.GetAk(offsetI + a, offset0 + b, &m);
        mMax = std::max(mMax, m);
        for (int c = 0; c < bsdfTable.nChannels; ++c)
            for (int k = 0; k < m; ++k)
                ak[c * bsdfTable.mMax + k] += weight * ap[c * m + k];
    }
}
```

Given the final weighted coefficients in  $ak$ , a call to `Fourier()` computes the BSDF value for the first color channel. Error in Fourier reconstruction can manifest itself as negative values, so the returned value must be clamped to zero.

Recall from Equation (8.22) that the  $a_k$  coefficients represent the cosine-weighted BSDF. This cosine factor must be removed from the value returned from the `f()` method; the `scale` term encodes this factor.

```
(Evaluate Fourier expansion for angle  $\phi$ ) ≡ 556
    Float Y = std::max((Float)0, Fourier(ak, mMax, cosPhi));
    Float scale = muI != 0 ? (1 / std::abs(muI)) : (Float)0;
    (Update scale to account for adjoint light transport 961)
    if (bsdfTable.nChannels == 1)
        return Spectrum(Y * scale);
    else {
        (Compute and return RGB colors for tabulated BSDF 559)
    }
}
```

As with specular transmission, radiance is scaled as it passes from a medium with one index of refraction to another, but this scaling isn't applied to rays starting from the camera. A definition and discussion of the fragment `(Update scale to account for adjoint light transport)`, which handles this adjustment, is provided in Section 16.1.

As mentioned earlier, when there are three color channels, the first channel encodes luminance and the next two are red and blue, respectively. To see how to compute a green channel value, consider the implementation of the function `RGBToXYZ()`, which uses the following equation to compute  $y_\lambda$  from red, green, and blue color components assuming the color primaries from the sRGB standard:

$$y_\lambda = 0.212671 r + 0.715160 g + 0.072169 b.$$

In this case, we know  $y_\lambda$ ,  $r$ , and  $b$ . Solving for  $g$ , we can find:

$$g = 1.39829 y_\lambda - 0.100913 b - 0.297375 r.$$

As before, any color coefficients with negative values due to error in Fourier reconstruction must be clamped to zero.

```
Float 1062
Fourier() 559
FourierBSDFTable::GetAk() 555
FourierBSDFTable::mMax 554
FourierBSDFTable::nChannels 554
RGBToXYZ() 328
Spectrum 315
```

```
<Compute and return RGB colors for tabulated BSDF> ≡ 558
    Float R = Fourier(ak + 1 * bsdfTable.mMax, mMax, cosPhi);
    Float B = Fourier(ak + 2 * bsdfTable.mMax, mMax, cosPhi);
    Float G = 1.39829f * Y - 0.100913f * B - 0.297375f * R;
    Float rgb[3] = { R * scale, G * scale, B * scale };
    return Spectrum::FromRGB(rgb).Clamp();
```

We'll now define the `Fourier()` function, which takes an array of coefficients  $a_k$ , the maximum order  $m$ , and the cosine of the angle  $\phi$ . It evaluates the weighted sum of cosines in Equation (8.22), which can be written in the simpler form with  $a_k$  now known:

$$f(\phi) = \sum_{k=0}^{m-1} a_k \cos(k \phi). \quad [8.24]$$

The implementation of this function uses double precision for the sum of terms in order to minimize the impact of floating-point round-off error in computing the sum.

```
<Fourier Interpolation Definitions> ≡
    Float Fourier(const Float *a, int m, double cosPhi) {
        double value = 0.0;
        <Initialize cosine iterates 559>
        for (int k = 0; k < m; ++k) {
            <Add the current summand and update the cosine iterates 560>
        }
        return value;
    }
```

As the number of coefficients increases, a naïve evaluation of Equation (8.24) involves a correspondingly large number of trigonometric function calls. This can lead to severe performance issues: current CPU architectures require a few hundred processor cycles for a single invocation of `std::cos()`. Therefore, it pays to use the *multiple angle formula* for cosines:

$$\cos(k \phi) = (2 \cos \phi) \cos((k - 1)\phi) - \cos((k - 2)\phi) \quad [8.25]$$

This expression expresses cosine of summand  $k$  in Equation (8.24) in terms of those used for the summands  $k - 1$  and  $k - 2$ .

The implementation starts with the declaration of two variables for the current and preceding cosine variables, corresponding to values for the indices  $k = -1$  and  $k = 0$ . Here, it's important to also use double precision to compute the  $\cos(k \phi)$  values; once  $m$  has values in the thousands, accumulated floating-point rounding error with 32-bit floats can become noticeable when using the multiple angle formula.

```
Float 1062
Fourier() 559
FourierBSDFTable::mMax 554
Spectrum::Clamp() 317
Spectrum::FromRGB() 330
```

```
<Initialize cosine iterates> ≡ 559
    double cosKMinusOnePhi = cosPhi;
    double cosKPhi = 1;
```

The body of the loop then adds the product of the current coefficient and cosine value to a running sum and computes the cosine values for the next iteration.

*(Add the current summand and update the cosine iterates) ≡* 559

```
value += a[k] * cosKPhi;
double cosKPlusOnePhi = 2 * cosPhi * cosKPhi - cosKMinusOnePhi;
cosKMinusOnePhi = cosKPhi;
cosKPhi = cosKPlusOnePhi;
```

### 8.6.1 SPLINE INTERPOLATION

The last detail to explain is how the spline-based interpolation used to reconstruct the  $a_k$  coefficients works. The implementation here uses the Catmull–Rom spline, which can be expressed in 1D as a weighted sum over four control points, where the weight and the particular control points used depend on the parametric location along the curve’s path where its value is being computed. The spline passes through the given control points and follows a fairly smooth curve along the way.

To understand how these weights are computed, first suppose we are given a set of values of a function  $f$  and its derivative  $f'$  at positions  $x_0, x_1, \dots, x_k$ . For each interval  $[x_i, x_{i+1}]$ , we would like to approximate the function using a cubic polynomial

$$p_i(x) = ax^3 + bx^2 + cx + d, \quad (8.26)$$

which is chosen so that it matches the function and its derivative at the sample locations, i.e.,  $p_i(x_i) = f(x_i)$ ,  $p_i(x_{i+1}) = f(x_{i+1})$ ,  $p'_i(x_i) = f'(x_i)$ , and  $p'_i(x_{i+1}) = f'(x_{i+1})$ . For simplicity, let us just focus on the first interval and furthermore suppose that  $[x_0, x_1] = [0, 1]$ . Solving for the coefficients  $a, b, c$ , and  $d$  yields

$$\begin{aligned} a &= f'(x_0) + f'(x_1) + 2f(x_0) - 2f(x_1), \\ b &= 3f(x_1) - 3f(x_0) - 2f'(x_0) - f'(x_1), \\ c &= f'(x_0), \\ d &= f(x_0). \end{aligned}$$

Note how all of the coefficients are linear in the function and derivative values, which lets us rewrite Equation (8.26) as

$$\begin{aligned} p(x) &= (2x^3 - 3x^2 + 1)f(x_0) \\ &\quad + (-2x^3 + 3x^2)f(x_1) \\ &\quad + (x^3 - 2x^2 + x)f'(x_0) \\ &\quad + (x^3 - x^2)f'(x_1). \end{aligned} \quad (8.27)$$

This kind of interpolant is convenient but unfortunately still too restrictive, since we cannot generally expect derivative information to be available: analytic derivatives of reflectance models are often cumbersome, and measured data does not provide them at all. We therefore estimate the derivatives at each  $f(x_i)$  using central differences based on the two adjacent function values  $f(x_{i-1})$  and  $f(x_{i+1})$ . The estimated derivative is then

$$f'(x_0) \approx \frac{f(x_1) - f(x_{-1})}{x_1 - x_{-1}} = \frac{f(x_1) - f(x_{-1})}{1 - x_{-1}}.$$

Similarly, the derivative at  $f(x_1)$  can be estimated using the two adjacent function values:

$$f'(x_1) \approx \frac{f(x_2) - f(x_0)}{x_2 - x_0} = \frac{f(x_2) - f(x_0)}{x_2}.$$

If we substitute these two expressions into Equation (8.27) and again collect  $f$  terms, we have:

$$\begin{aligned} p(x) &= \frac{x^3 - 2x^2 + x}{x_{-1} - 1} f(x_{-1}) \\ &\quad + \left( 2x^3 - 3x^2 + 1 - \frac{x^3 - x^2}{x_2} \right) f(x_0) \\ &\quad + \left( -2x^3 + 3x^2 + \frac{x^3 - 2x^2 + x}{1 - x_{-1}} \right) f(x_1) \\ &\quad + \frac{x^3 - x^2}{x_2} f(x_2), \end{aligned}$$

Note that the weights are independent of the function values: we can therefore also express this interpolation as

$$p(x) = w_0 f(x_{-1}) + w_1 f(x_0) + w_2 f(x_1) + w_3 f(x_2), \quad [8.28]$$

with

$$\begin{aligned} w_0 &= \frac{x^3 - 2x^2 + x}{x_{-1} - 1} \\ w_1 &= 2x^3 - 3x^2 + 1 - \frac{x^3 - x^2}{x_2} = \left( 2x^3 - 3x^2 + 1 \right) - w_3 \\ w_2 &= -2x^3 + 3x^2 + \frac{x^3 - 2x^2 + x}{1 - x_{-1}} = \left( -2x^3 + 3x^2 \right) + w_0 \\ w_3 &= \frac{x^3 - x^2}{x_2}. \end{aligned} \quad [8.29]$$

The `CatmullRomWeights()` function takes the variable  $x$  and the number of interpolation nodes and their positions as arguments. It does not use the function values in any way but instead computes the index offset and an array with four weights corresponding to the expressions in Equation (8.29).

`CatmullRomWeights()` [562](#)

The code that computes these weights is useful beyond the task of BSDF reconstruction and is thus defined in the files `core/interpolation.h` and `core/interpolation.cpp`.

```
(Spline Interpolation Definitions) ≡
bool CatmullRomWeights(int size, const Float *nodes, Float x,
                        int *offset, Float *weights) {
    {Return false if x is out of bounds 562}
    {Search for the interval idx containing x 562}
    {Compute the t parameter and powers 562}
    {Compute initial node weights w1 and w2 563}
    {Compute first node weight w0 563}
    {Compute last node weight w3}
    return true;
}
```

The first statement returns a failure when  $x$  is outside the domain of the function. Note the somewhat peculiar way of writing the conditional logic in negated form: this way, we can also catch NaN arguments, which by convention cause comparisons to evaluate to false.

```
(Return false if x is out of bounds) ≡ 562
if (!(x >= nodes[0] && x <= nodes[size-1]))
    return false;
```

The `FindInterval()` helper function efficiently locates the index of the interval containing  $x$  via binary search. With its result, we can now set the `*offset` return value to the index of the node  $x_{i-1}$  and set variables  $x_0$  and  $x_1$  that delimit the extent of the domain of the corresponding spline segment.

Note that it's possible that this offset would cause an out-of-bounds array access when Equation (8.28) is evaluated. (Specifically, in the case where the offset is one element before the start of the `nodes` array, when `idx == 0`, or if `idx` equals the size of the array minus one.) In these cases, the corresponding interpolation weights will always be set to zero for any out-of-bounds entries. Code that uses these weights in pbrt is therefore carefully written to never access the function values array for any indices where the weight is zero.

```
(Search for the interval idx containing x) ≡ 562
int idx = FindInterval(size, [&](int i) { return nodes[i] <= x; });
*offset = idx - 1;
Float x0 = nodes[idx], x1 = nodes[idx+1];
```

Because our derivation of the spline assumed the unit interval, we'll define a scaled variable  $t$  in  $[0, 1]$  in the code here. It's also useful to precompute some integer powers of  $t$ .

```
(Compute the t parameter and powers) ≡ 562
Float t = (x - x0) / (x1 - x0), t2 = t * t, t3 = t2 * t;
```

`FindInterval()` 1065

`Float` 1062

The implementation starts by initializing the second and third weights  $w_1$  and  $w_2$  using the results from Equation (8.29). For starters, only the terms in parenthesis are included.

*(Compute initial node weights  $w_1$  and  $w_2$ )* ≡

```
weights[1] = 2 * t3 - 3 * t2 + 1;
weights[2] = -2 * t3 + 3 * t2;
```

562

There are two important details involved in computing the weights  $w_0$  and  $w_3$  from Equation (8.29). First, we need to introduce a scale factor of  $x_1 - x_0$ , which corrects for the fact that the  $t$  values used in the code here incorporate a rescaling to the unit interval, while we actually want derivatives with respect to the original parameterization of the function.

Second, it's necessary to handle an edge condition: the usual case is that  $idx > 0$  and a previous neighbor exists; in this case, `weights[0]` can be initialized directly and the  $w_0$  term can be added to `weights[2]`, completing its initialization. If there is no previous neighbor, then the derivative  $f'(x_0)$  is instead approximated with the forward difference  $f(x_1) - f(x_0)$ . In this case, a similar algebraic process can be followed as was used to find the weights in Equation (8.29); the result is used here.

*(Compute first node weight  $w_0$ )* ≡

```
if (idx > 0) {
    Float w0 = (t3 - 2 * t2 + t) * (x1 - x0) / (x1 - nodes[idx - 1]);
    weights[0] = -w0;
    weights[2] += w0;
} else {
    Float w0 = t3 - 2 * t2 + t;
    weights[0] = 0;
    weights[1] -= w0;
    weights[2] += w0;
}
```

562

The computation for the  $w_3$  follows similarly and so the fragment that implements this part of the function, *(Compute last node weight  $w_3$ )*, isn't included here.

Given this machinery, we can now define the implementation of the `FourierBSDFTable::GetWeightsAndOffsets()` method, which just calls into `CatmullRomWeights()`, passing it the sampled `mu` array.

*(BxDF Method Definitions)* +≡

```
bool FourierBSDFTable::GetWeightsAndOffset(Float cosTheta, int *offset,
                                             Float weights[4]) const {
    return CatmullRomWeights(nMu, mu, cosTheta, offset, weights);
}
```

## FURTHER READING

`CatmullRomWeights()` 562

`Float` 1062

`FourierBSDFTable::mu` 554

`FourierBSDFTable::nMu` 554

Phong (1975) developed an early empirical reflection model for glossy surfaces in computer graphics. Although neither reciprocal nor energy conserving, it was a cornerstone of the first synthetic images of non-Lambertian objects. The Torrance–Sparrow microfacet model is described in Torrance and Sparrow (1967); it was first introduced to graphics by Blinn (1977), and a variant of it was used by Cook and Torrance (1981,

1982). The Oren–Nayar Lambertian model is described in their 1994 paper (Oren and Nayar 1994).

Hall's (1989) book collected and described the state of the art in physically based surface reflection models for graphics; it remains a seminal reference. It discusses the physics of surface reflection in detail, with many pointers to the original literature and with many tables of useful measured data about reflection from real surfaces. Burley's (2012) more recent paper includes a thorough annotated bibliography of more recent work on reflection models for computer graphics.

Heitz's paper on microfacet shadowing-masking functions (2014a) provides a very well-written introduction to microfacet BSDF models in general, with many useful figures and explanations about details of the topic. See the papers by Beckmann and Spizzichino (1963) and Trowbridge and Reitz (1975) for the introduction of their respective microfacet distribution functions. Kurt et al. (2010) developed an anisotropic Beckmann–Spizzichino distribution function; see Heitz (2014a) for anisotropic variants of many other microfacet distribution functions. Early anisotropic BRDF models for computer graphics were developed by Kajiya (1985) and Poulin and Fournier (1990).

The microfacet masking-shadowing function in Equation (8.15) was introduced by Smith (1967), who used the assumption of no correlation between the height of the microsurface at nearby points to derive this result. Smith also first derived the normalization constraint in Equation (8.13). (This result was derived independently by Ashikhmin, Premoze, and Shirley (2000).) See Heitz (2014a) for further discussion of derivations of these functions. A more accurate  $G(\omega_i, \omega_o)$  function for Gaussian microfacet surfaces that better accounts for the effects of correlation between the two directions was developed by Heitz et al. (2013), and the rational approximation to the Beckmann–Spizzichino  $\Lambda(\omega)$  function used in this chapter is due to Heitz (2014a), which is derived from an approximation developed by Walter et al. (2007). Our derivation of the  $\Lambda(\omega)$  function, Equation (8.14), is also due to Heitz (2015).

Stam (2001) developed a generalization of the Cook–Torrance model for transmission, and more recently Walter et al. (2007) revisited this problem. Weyrich et al. (2009) have developed methods to infer a microfacet distribution that matches a measured or desired reflection distribution. Remarkably, they show that it's possible to manufacture actual physical surfaces that match a desired reflection distribution fairly accurately. Simonot (2009) has developed a model that spans Oren–Nayar and Torrance–Sparrow: microfacets are modeled as Lambertian reflectors with a layer above them that ranges from perfectly transmissive to a perfect specular reflector. However, this model doesn't account for masking-shadowing effects and can't be evaluated in closed form.

The microfacet reflection models in this chapter are all based on the assumption that so many microfacets are visible in a pixel that they can be accurately described by their aggregate statistical behavior. This assumption isn't true for many real-world surfaces, where a relatively small number of microfacets may be visible in each pixel; examples of such surfaces include car paint and glittery plastics. Both Yan et al. (2014) and Jakob et al. (2014b) have developed techniques that model these cases well.

It can be useful to be able to find BSDFs for layered materials, such as a metal base surface tarnished with patina, or wood with a varnish coating. Hanrahan and Krueger (1993)