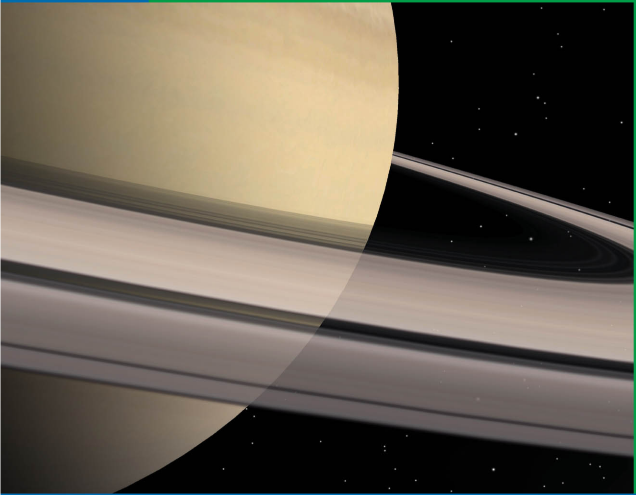
OpenGL® SuperBible

Seventh Edition
Comprehensive Tutorial and Reference



Graham Sellers ■ Richard S. Wright, Jr. ■ Nicholas Haemel

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

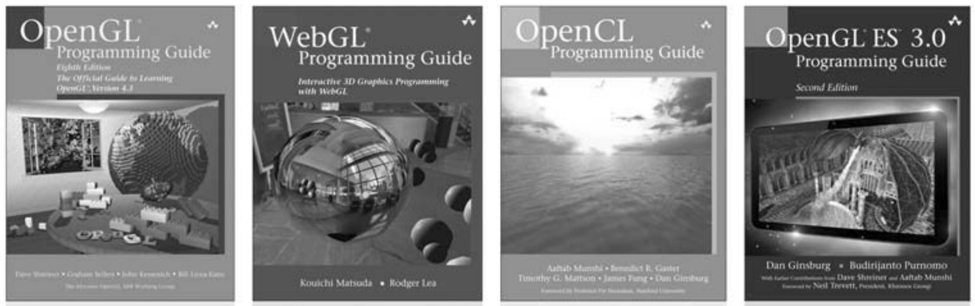
OpenGL®

SuperBible

Seventh Edition

OpenGL Series

from Addison-Wesley


OpenGL Programming Guide
Eighth Edition
The Official Guide to Learning OpenGL, Version 4.3
WebGL Programming Guide
Interactive 3D Graphics Programming with WebGL
Kouichi Matsuda • Rodger Lea
OpenCL Programming Guide
Aaftab Munshi • Benedict R. Gaster
Timothy G. Mattson • James Fung • Dan Ginsburg
OpenGL ES 3.0 Programming Guide
Second Edition
Dan Ginsburg • Budirijanto Purnomo

Addison-Wesley

Visit **informit.com/opengl** for a complete list of available products.

The OpenGL graphics system is a software interface to graphics hardware. ("GL" stands for "Graphics Library".) It allows you to create interactive programs that produce color images of moving, three-dimensional objects. With OpenGL, you can control computer-graphics technology to produce realistic pictures, or ones that depart from reality in imaginative ways.

The **OpenGL Series** from Addison-Wesley Professional comprises tutorial and reference books that help programmers gain a practical understanding of OpenGL standards, along with the insight needed to unlock OpenGL's full potential.

Make sure to connect with us!
informit.com/socialconnect

informIT.com
the trusted technology learning source
| Addison-Wesley | Safari

ALWAYS LEARNING

PEARSON

OpenGL® SuperBible

Seventh Edition

Comprehensive Tutorial and Reference

Graham Sellers
Richard S. Wright, Jr.
Nicholas Haemel

Addison-Wesley

New York • Boston • Indianapolis • San Francisco
Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

OpenGL® is a registered trademark of Silicon Graphics Inc. and is used by permission of Khronos.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data
Wright, Richard S., Jr., 1965- author.
OpenGL superBible : comprehensive tutorial and reference.—Seventh edition / Graham Sellers, Richard S. Wright, Jr., Nicholas Haemel.
pages cm
Includes bibliographical references and index.
ISBN 978-0-672-33747-5 (pbk. : alk. paper)—ISBN 0-672-33747-9 (pbk. : alk. paper)
1. Computer graphics. 2. OpenGL. I. Sellers, Graham, author. II. Haemel, Nicholas, author. III. Title.
T385.W728 2016
006.6′8—dc23
2015014278

Copyright © 2016 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, 200 Old Tappan Road, Old Tappan, New Jersey 07675, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-672-33747-5
ISBN-10: 0-672-33747-9
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, July 2015

Editor-in-Chief
Mark L. Taub

Executive Editor
Laura Lewin

Managing Editor
John Fuller

Full-Service Production Manager
Julie B. Nahil

Copy Editor
Jill Hobbs

Indexer
Larry D. Sweazy

Proofreader
Anna Popick

Technical Reviewer
Matías Goldberg

Editorial Assistant
Olivia Basegio

Compositor
diacriTech

For you, the reader.
—Graham Sellers

This page intentionally left blank

Contents

Figures **xv**
Tables **xxi**
Listings **xxiii**
Foreword **xxxi**
Preface **xxxiii**
The Architecture of the Book xxxiv
What's New in This Edition xxxvi
How to Build the Examples xxxvii
Errata xxxviii
Note from the Publisher xxxviii
Acknowledgments **xxxix**
About the Author **xli**

I Foundations **1**

1 Introduction **3**
OpenGL and the Graphics Pipeline 4
The Origins and Evolution of OpenGL 6
Core Profile OpenGL 8
Primitives, Pipelines, and Pixels 10
Summary 11

2 Our First OpenGL Program **13**
Creating a Simple Application 14
Using Shaders 17

Drawing Our First Triangle 24
Summary 26

3 Following the Pipeline 27
Passing Data to the Vertex Shader 28
Vertex Attributes 28
Passing Data from Stage to Stage 30
Interface Blocks 31
Tessellation 33
Tessellation Control Shaders 33
The Tessellation Engine 35
Tessellation Evaluation Shaders 35
Geometry Shaders 37
Primitive Assembly, Clipping, and Rasterization 39
Clipping 39
Viewport Transformation 40
Culling 41
Rasterization 43
Fragment Shaders 43
Framebuffer Operations 47
Pixel Operations 47
Compute Shaders 48
Using Extensions in OpenGL 49
Enhancing OpenGL with Extensions 50
Summary 54

4 Math for 3D Graphics 55
Is This the Dreaded Math Chapter 56
A Crash Course in 3D Graphics Math 57
Vectors, or Which Way Is Which? 57
Common Vector Operators 60
Matrices 64
Matrix Construction and Operators 66
Understanding Transformations 69
Coordinate Spaces in OpenGL 70
Coordinate Transformations 72
Concatenating Transformations 80
Quaternions 81
The Model–View Transform 83
Projection Transformations 86
Interpolation, Lines, Curves, and Splines 89

Curves . . . 91
Splines . . . 94
Summary . . . 97

5 Data **99**
Buffers . . . 100
Creating Buffers and Allocating Memory . . . 100
Filling and Copying Data in Buffers . . . 107
Feeding Vertex Shaders from Buffers . . . 109
Uniforms . . . 117
Default Block Uniforms . . . 117
Uniform Blocks . . . 121
Using Uniforms to Transform Geometry . . . 135
Shader Storage Blocks . . . 140
Synchronizing Access to Memory . . . 145
Atomic Counters . . . 147
Synchronizing Access to Atomic Counters . . . 151
Textures . . . 152
Creating and Initialzing Textures . . . 152
Texture Targets and Types . . . 154
Reading from Textures in Shaders . . . 156
Loading Textures from Files . . . 160
Controlling How Texture Data Is Read . . . 164
Array Textures . . . 177
Writing to Textures in Shaders . . . 182
Synchronizing Access to Images . . . 194
Texture Compression . . . 195
Texture Views . . . 199
Summary . . . 203

6 Shaders and Programs **205**
Language Overview . . . 206
Data Types . . . 206
Built-In Functions . . . 213
Compiling, Linking, and Examining Programs . . . 219
Getting Information from the Compiler . . . 219
Getting Information from the Linker . . . 223
Separate Programs . . . 225
Shader Subroutines . . . 231
Program Binaries . . . 235
Summary . . . 238

II In Depth 239

7 Vertex Processing and Drawing Commands 241
Vertex Processing . . . 242
Vertex Shader Inputs . . . 242
Vertex Shader Outputs . . . 247
Drawing Commands . . . 249
Indexed Drawing Commands . . . 249
Instancing . . . 255
Indirect Draws . . . 269
Storing Transformed Vertices . . . 278
Using Transform Feedback . . . 279
Starting, Pausing, and Stopping Transform Feedback . . . 284
Ending the Pipeline with Transform Feedback . . . 286
Transform Feedback Example: Physical Simulation . . . 286
Clipping . . . 296
User-Defined Clipping . . . 300
Summary . . . 303

8 Primitive Processing 305
Tessellation . . . 306
Tessellation Primitive Modes . . . 307
Tessellation Subdivision Modes . . . 316
Passing Data between Tessellation Shaders . . . 319
Communication between Shader Invocations . . . 322
Tessellation Example: Terrain Rendering . . . 323
Tessellation Example: Cubic Bézier Patches . . . 329
Geometry Shaders . . . 333
The Pass-Through Geometry Shader . . . 334
Using Geometry Shaders in an Application . . . 336
Discarding Geometry in the Geometry Shader . . . 341
Modifying Geometry in the Geometry Shader . . . 344
Generating Geometry in the Geometry Shader . . . 346
Changing the Primitive Type in the Geometry Shader . . . 349
Multiple Streams of Storage . . . 352
New Primitive Types Introduced by the Geometry Shader . . . 353
Multiple Viewport Transformations . . . 361
Summary . . . 364

9 Fragment Processing and the Framebuffer 365
Fragment Shaders . . . 366
Interpolation and Storage Qualifiers . . . 366

Per-Fragment Tests 369
Scissor Testing 369
Stencil Testing 372
Depth Testing 376
Early Testing 380
Color Output 382
Blending 382
Logical Operations 387
Color Masking 388
Off-Screen Rendering 390
Multiple Framebuffer Attachments 395
Layered Rendering 397
Framebuffer Completeness 403
Rendering in Stereo 407
Antialiasing 412
Antialiasing by Filtering 413
Multi-Sample Antialiasing 415
Multi-Sample Textures 417
Sample Rate Shading 421
Centroid Sampling 424
Advanced Framebuffer Formats 428
Rendering with No Attachments 428
Floating-Point Framebuffers 430
Integer Framebuffers 444
The sRGB Color Space 446
Point Sprites 448
Texturing Points 449
Rendering a Star Field 450
Point Parameters 453
Shaped Points 454
Rotating Points 456
Getting at Your Image 458
Reading from a Framebuffer 458
Copying Data between Framebuffers 461
Reading Back Texture Data 464
Summary 466

10 Compute Shaders 467
Using Compute Shaders 468
Executing Compute Shaders 469
Compute Shader Communication 474
Examples 479

Compute Shader Parallel Prefix Sum 479
Compute Shader Flocking 492
Summary 502

11 Advanced Data Management 503
Eliminating Binding 504
Sparsely Populated Textures 509
Texture Compression 516
The RGTC Compression Scheme 516
Generating Compressed Data 519
Packed Data Formats 525
High-Quality Texture Filtering 527
Summary 531

12 Controlling and Monitoring the Pipeline 533
Queries 534
Occlusion Queries 535
Timer Queries 545
Transform Feedback Queries 549
Pipeline State Queries 555
Synchronization in OpenGL 556
Draining the Pipeline 557
Synchronization and Fences 557
Summary 562

III In Practice 565

13 Rendering Techniques 567
Lighting Models 568
The Phong Lighting Model 568
Blinn-Phong Lighting 577
Rim Lighting 579
Normal Mapping 582
Environment Mapping 587
Material Properties 597
Casting Shadows 599
Atmospheric Effects 605
Non-Photo-Realistic Rendering 610
Cell Shading: Texels as Light 610

Alternative Rendering Methods 613
Deferred Shading 613
Screen-Space Techniques 624
Rendering without Triangles 631
Two-Dimensional Graphics 647
Distance Field Textures 647
Bitmap Fonts 656
Summary 659

14 High-Performance OpenGL 661
Optimizing CPU Performance 661
Multi-Threading in OpenGL 662
Packet Buffers 668
Low-Overhead OpenGL 677
Indirect Rendering 678
GPU Work Generation 683
Zero Copy 691
Performance Analysis Tools 699
Windows Performance Toolkit and GPUView 699
GPU PerfStudio 703
Tuning Your Application for Speed 706
Summary 726

15 Debugging and Stability 729
Debugging Your Applications 730
Debug Contexts 730
Security and Robustness 737
Graphics Reset 737
Range-Checked Reads 740
Summary 742

A Further Reading 743

B The SBM File Format 749

C The SuperBible Tools 759

Glossary 797
Index 805

This page intentionally left blank

Figures

Figure 1.1 Simplified graphics pipeline 6
Figure 1.2 Future Crew's 1992 demo—Unreal 9

Figure 2.1 The output of our first OpenGL application 16
Figure 2.2 Rendering our first point 23
Figure 2.3 Making our first point bigger 24
Figure 2.4 Our very first OpenGL triangle 26

Figure 3.1 Our first tessellated triangle 37
Figure 3.2 Tessellated triangle after adding a geometry shader . . 39
Figure 3.3 Clockwise (left) and counterclockwise (right) winding order 42
Figure 3.4 Result of Listing 3.10 45
Figure 3.5 Result of Listing 3.12 46
Figure 3.6 Realtech VR's OpenGL Extensions Viewer 51

Figure 4.1 A point in space is both a vertex and a vector 58
Figure 4.2 The dot product: cosine of the angle between two vectors 61
Figure 4.3 A cross product returns a vector perpendicular to its parameters 62
Figure 4.4 Reflection and refraction 64
Figure 4.5 A 4×4 matrix representing rotation and translation 68
Figure 4.6 Modeling transformations: (a) rotation, then translation and (b) translation, then rotation 69

Figure 4.7 Two perspectives of view coordinates 72
Figure 4.8 The modeling transformations 73
Figure 4.9 A cube translated ten units in the positive *y* direction 75
Figure 4.10 A cube rotated about an arbitrary axis 78
Figure 4.11 A non-uniform scaling of a cube 80
Figure 4.12 A side-by-side example of an orthographic versus perspective projection 88
Figure 4.13 Finding a point on a line 90
Figure 4.14 A simple Bézier curve 91
Figure 4.15 A cubic Bézier curve 93
Figure 4.16 A cubic Bézier spline 95

Figure 5.1 Binding buffers and uniform blocks to binding points 133
Figure 5.2 A few frames from the spinning cube application . . . 139
Figure 5.3 Many cubes! 140
Figure 5.4 A simple textured triangle 157
Figure 5.5 A full-screen texture loaded from a `.KTX` file 162
Figure 5.6 An object wrapped in simple textures 164
Figure 5.7 Texture filtering—nearest neighbor (left) and linear (right) 169
Figure 5.8 A series of mipmapped images 171
Figure 5.9 A tunnel rendered with three textures and mipmapping 174
Figure 5.10 Example of texture coordinate wrapping modes 176
Figure 5.11 `GL_MIRROR_CLAMP_TO_EDGE` in action 177
Figure 5.12 Output of the alien rain sample 182
Figure 5.13 Resolved per-fragment linked lists 194

Figure 6.1 Shape of a Hermite curve 217

Figure 7.1 Indices used in an indexed draw 250
Figure 7.2 Base vertex used in an indexed draw 253
Figure 7.3 Triangle strips (a) without primitive restart and (b) with primitive restart 255
Figure 7.4 First attempt at an instanced field of grass 260
Figure 7.5 Slightly perturbed blades of grass 260
Figure 7.6 Control over the length and orientation of our grass 262
Figure 7.7 The final field of grass 263
Figure 7.8 Result of instanced rendering 268
Figure 7.9 Result of asteroid rendering program 277
Figure 7.10 Relationship of transform feedback binding points . . 282
Figure 7.11 Connections of vertices in the spring mass system . . 287
Figure 7.12 Simulation of points connected by springs 295

Figure 7.13 Visualizing springs in the spring mass system 296
Figure 7.14 Clipping lines 297
Figure 7.15 Clipping triangles 298
Figure 7.16 Clipping triangles using a guard band 299
Figure 7.17 Rendering with user clip distances 303

Figure 8.1 Schematic of OpenGL tessellation 307
Figure 8.2 Tessellation factors for quad tessellation 308
Figure 8.3 Quad tessellation example 309
Figure 8.4 Tessellation factors for triangle tessellation 310
Figure 8.5 Triangle tessellation example 311
Figure 8.6 Tessellation factors for isoline tessellation 313
Figure 8.7 Isoline tessellation example 314
Figure 8.8 Tessellated isoline spirals example 315
Figure 8.9 Triangle tessellated using point mode 316
Figure 8.10 Tessellation using different subdivision modes 318
Figure 8.11 Displacement map used in terrain example 323
Figure 8.12 Terrain rendered using tessellation 328
Figure 8.13 Tessellated terrain in wireframe 328
Figure 8.14 Final rendering of a cubic Bézier patch 332
Figure 8.15 A Bézier patch and its control cage 333
Figure 8.16 Geometry culled from different viewpoints 344
Figure 8.17 Exploding a model using the geometry shader 345
Figure 8.18 Basic tessellation using the geometry shader 348
Figure 8.19 Displaying the normals of a model using a geometry shader 352
Figure 8.20 Lines produced using lines with adjacency primitives 354
Figure 8.21 Triangles produced using `GL_TRIANGLES_ADJACENCY` 355
Figure 8.22 Triangles produced using `GL_TRIANGLE_STRIP_ADJACENCY` 355
Figure 8.23 Ordering of vertices for `GL_TRIANGLE_STRIP_ADJACENCY` 356
Figure 8.24 Rendering a quad using a pair of triangles 357
Figure 8.25 Parameterization of a quad 358
Figure 8.26 Quad rendered using a geometry shader 359
Figure 8.27 Result of rendering to multiple viewports 363

Figure 9.1 Contrasting perspective-correct and linear interpolation 369
Figure 9.2 Rendering with four different scissor rectangles 372
Figure 9.3 Effect of depth clamping at the near plane 379
Figure 9.4 A clipped object with and without depth clamping . . 380

Figure 9.5 All possible combinations of blending functions . . . 386
Figure 9.6 Result of rendering into a texture 395
Figure 9.7 Result of the layered rendering example 401
Figure 9.8 Result of stereo rendering to a stereo display 412
Figure 9.9 Antialiasing using line smoothing 413
Figure 9.10 Antialiasing using polygon smoothing 414
Figure 9.11 Antialiasing sample positions 415
Figure 9.12 No antialising (left) and eight-sample antialiasing (center and right) 416
Figure 9.13 Antialiasing of high-frequency shader output 422
Figure 9.14 Partially covered multi-sampled pixels 425
Figure 9.15 Different views of an HDR image 433
Figure 9.16 Histogram of levels for treelights.ktx 434
Figure 9.17 Naïve tone mapping by clamping 435
Figure 9.18 Transfer curve for adaptive tone mapping 436
Figure 9.19 Result of adaptive tone-mapping program 438
Figure 9.20 The effect of light bloom on an image 439
Figure 9.21 Original and thresholded output for bloom example 441
Figure 9.22 Blurred thresholded bloom colors 443
Figure 9.23 Result of the bloom program 444
Figure 9.24 Gamma curves for sRGB and simple powers 447
Figure 9.25 A particle effect in the flurry screen saver 449
Figure 9.26 The star texture map 451
Figure 9.27 Flying through space with point sprites 453
Figure 9.28 Two potential orientations of textures on a point sprite 454
Figure 9.29 Analytically generated point sprite shapes 455

Figure 10.1 Global and local compute work group dimensions . . 473
Figure 10.2 Effect of race conditions in a compute shader 478
Figure 10.3 Effect of `barrier()` on race conditions 478
Figure 10.4 Sample input and output of a prefix sum operation . . 480
Figure 10.5 Breaking a prefix sum into smaller chunks 482
Figure 10.6 A 2D prefix sum 484
Figure 10.7 Computing the sum of a rectangle in a summed area table 487
Figure 10.8 Variable filtering applied to an image 487
Figure 10.9 Depth of field in a photograph 488
Figure 10.10 Applying depth of field to an image 491
Figure 10.11 Effects achievable with depth of field 491
Figure 10.12 Stages in the iterative flocking algorithm 493
Figure 10.13 Output of compute shader flocking program 501

Figure 11.1 Output of the `bindlesstex` example application . . . 508
Figure 11.2 Output of the `sparsetexture` example application . . 514
Figure 11.3 Representation of image data as endpoints of a line 517
Figure 11.4 Result of using RGTC texture compression on a distance field 525
Figure 11.5 Linear interpolation under high magnification 528
Figure 11.6 Graph showing linear interpolation 529
Figure 11.7 Graph showing smooth interpolation 529
Figure 11.8 Result of smooth interpolation 531

Figure 13.1 Vectors used in Phong lighting 570
Figure 13.2 Per-vertex lighting (Gouraud shading) 573
Figure 13.3 Per-fragment lighting (Phong shading) 574
Figure 13.4 Varying specular parameters of a material 577
Figure 13.5 Phong lighting (left) versus Blinn-Phong lighting (right) 579
Figure 13.6 Rim lighting vectors 580
Figure 13.7 Result of rim lighting example 581
Figure 13.8 Example normal map 583
Figure 13.9 Result of normal mapping example 586
Figure 13.10 A selection of spherical environment maps 588
Figure 13.11 Result of rendering with spherical environment mapping 589
Figure 13.12 Example equirectangular environment map 591
Figure 13.13 Rendering result of equirectangular environment map 591
Figure 13.14 The layout of six cube faces in the cubemap sample program 593
Figure 13.15 Cubemap environment rendering with a skybox . . . 597
Figure 13.16 Pre-filtered environment maps and gloss map 598
Figure 13.17 Result of per-pixel gloss example 599
Figure 13.18 Depth as seen from a light 602
Figure 13.19 Results of rendering with shadow maps 605
Figure 13.20 Graphs of exponential decay 608
Figure 13.21 Applying fog to a tessellated landscape 609
Figure 13.22 A one-dimensional color lookup table 610
Figure 13.23 A toon-shaded torus 613
Figure 13.24 Visualizing components of a G-buffer 618
Figure 13.25 Final rendering using deferred shading 620
Figure 13.26 Deferred shading with (left) and without (right) normal maps 622
Figure 13.27 Bumpy surface occluding points 625
Figure 13.28 Selection of random vector in an oriented hemisphere 626

Figure 13.29 Effect of increasing direction count on ambient occlusion 627
Figure 13.30 Effect of introducing noise in ambient occlusion . . . 628
Figure 13.31 Ambient occlusion applied to a rendered scene 629
Figure 13.32 A few frames from the Julia set animation 635
Figure 13.33 Simplified 2D illustration of ray tracing 636
Figure 13.34 Our first ray-traced sphere 640
Figure 13.35 Our first lit ray-traced sphere 641
Figure 13.36 Implementing a stack using framebuffer objects 642
Figure 13.37 Ray-traced spheres with increasing ray bounces 643
Figure 13.38 Adding a ray-traced plane 645
Figure 13.39 Ray-traced spheres in a box 646
Figure 13.40 Low-resolution texture used for a logo 648
Figure 13.41 High-resolution texture used for a logo 649
Figure 13.42 Distance field of the OpenGL logo 650
Figure 13.43 Distance fields for a line 650
Figure 13.44 Output of distance field rendering application 651
Figure 13.45 Distance field for English characters 652
Figure 13.46 Distance field of a Chinese character 653
Figure 13.47 Chinese text rendered using distance fields 653
Figure 13.48 Two textures to be mixed using a distance field 654
Figure 13.49 Landscape map texture and distance field 655
Figure 13.50 Result of landscape texturing with distance fields . . . 655
Figure 13.51 Output of font rendering demo 659

Figure 14.1 Output of the OpenMP particle simulator 667
Figure 14.2 CPU utilization of the `ompparticles` application . . . 667
Figure 14.3 Indirect material parameters 683
Figure 14.4 Output of the `cullindirect` application 690
Figure 14.5 Synchronizing access to a mapped buffer 695
Figure 14.6 Persistent mapped Julia fractal 698
Figure 14.7 GPUView in action 700
Figure 14.8 Vsync seen in GPUView 701
Figure 14.9 A packet dialog in GPUView 703
Figure 14.10 GPU PerfStudio running an example application . . . 704
Figure 14.11 GPU PerfStudio frame debugger 705
Figure 14.12 GPU PerfStudio HUD control window 705
Figure 14.13 GPU PerfStudio overlaying information 706
Figure 14.14 GPU PerfStudio showing AMD performance counters 707
Figure 14.15 GPUView showing the effect of **`glReadPixels()`** into system memory 708
Figure 14.16 GPUView showing the effect of **`glReadPixels()`** into a buffer 709

Figure B.1 Dump of example SBM file 757

Tables

Table 1.1 OpenGL Versions and Publication Dates 7

Table 4.1 Common Coordinate Spaces Used in 3D Graphics . . . 70

Table 5.1 Buffer Storage Flags 102
Table 5.2 Buffer-Mapping Flags 106
Table 5.3 Basic OpenGL Type Tokens and Their Corresponding C Types 108
Table 5.4 Uniform Parameter Queries via **`glGetActiveUniformsiv()`** 127
Table 5.5 Atomic Operations on Shader Storage Blocks 143
Table 5.6 Texture Targets and Description 155
Table 5.7 Basic Texture Targets and Sampler Types 157
Table 5.8 Texture Filters, Including Mipmapped Filters 172
Table 5.9 Image Types 183
Table 5.10 Image Data Format Classes 185
Table 5.11 Image Data Format Classes 186
Table 5.12 Atomic Operations on Images 189
Table 5.13 Native OpenGL Texture Compression Formats 195
Table 5.14 Texture View Target Compatibility 200
Table 5.15 Texture View Format Compatibility 201

Table 6.1 Scalar Types in GLSL 206
Table 6.2 Vector and Matrix Types in GLSL 208

Table 7.1 Vertex Attribute Types 244
Table 7.2 Draw Type Matrix 250
Table 7.3 Values for `primitiveMode` 285

Table 8.1 Allowed Draw Modes for Geometry Shader Input Modes 337
Table 8.2 Sizes of Input Arrays to Geometry Shaders 339

Table 9.1 Stencil Functions 373
Table 9.2 Stencil Operations 375
Table 9.3 Depth Comparison Functions 377
Table 9.4 Blend Functions 384
Table 9.5 Blend Equations 387
Table 9.6 Logic Operations 388
Table 9.7 Framebuffer Completeness Return Values 405
Table 9.8 Floating-Point Texture Formats 432

Table 11.1 First RGTC Encoding for RED Images 518
Table 11.2 Second RGTC Encoding for RED Images 519
Table 11.3 Packed Data Formats Supported in OpenGL 527

Table 12.1 Possible Return Values for **`glClientWaitSync()`** 560

Table 14.1 Map Buffer Access Types 710

Table C.1 OpenGL Functions 760
Table C.2 OpenGL Extensions (Core) 777

Listings

Listing 2.1 Our first OpenGL application 14
Listing 2.2 Animating color over time 16
Listing 2.3 Our first vertex shader 18
Listing 2.4 Our first fragment shader 18
Listing 2.5 Compiling a simple shader 18
Listing 2.6 Creating the program member variable 21
Listing 2.7 Rendering a single point 22
Listing 2.8 Producing multiple vertices in a vertex shader 25
Listing 2.9 Rendering a single triangle 25

Listing 3.1 Declaration of a vertex attribute 28
Listing 3.2 Updating a vertex attribute 29
Listing 3.3 Vertex shader with an output 30
Listing 3.4 Fragment shader with an input 31
Listing 3.5 Vertex shader with an output interface block 31
Listing 3.6 Fragment shader with an input interface block 32
Listing 3.7 Our first tessellation control shader 34
Listing 3.8 Our first tessellation evaluation shader 36
Listing 3.9 Our first geometry shader 38
Listing 3.10 Deriving a fragment's color from its position 44
Listing 3.11 Vertex shader with an output 45
Listing 3.12 Deriving a fragment's color from its position 46
Listing 3.13 Simple do-nothing compute shader 49

Listing 5.1 Creating and initializing a buffer 103
Listing 5.2 Updating the content of a buffer with **`glBufferSubData()`** 104

Listing 5.3 Mapping a buffer's data store with **glMapNamedBuffer()** 104
Listing 5.4 Setting up a vertex attribute 112
Listing 5.5 Using an attribute in a vertex shader 113
Listing 5.6 Declaring two inputs to a vertex shader 113
Listing 5.7 Multiple separate vertex attributes 114
Listing 5.8 Multiple interleaved vertex attributes 115
Listing 5.9 Example uniform block declaration 121
Listing 5.10 Declaring a uniform block with the **std140** layout . . 123
Listing 5.11 Example uniform block with offsets 124
Listing 5.12 Uniform block with user-specified offsets 125
Listing 5.13 Uniform block with user-specified alignments 125
Listing 5.14 Retrieving the indices of uniform block members . . 126
Listing 5.15 Retrieving the information about uniform block members 127
Listing 5.16 Setting a single float in a uniform block 129
Listing 5.17 Retrieving the indices of uniform block members . . 129
Listing 5.18 Specifying the data for an array in a uniform block 129
Listing 5.19 Setting up a matrix in a uniform block 130
Listing 5.20 Specifying bindings for uniform blocks 133
Listing 5.21 Uniform blocks binding layout qualifiers 134
Listing 5.22 Setting up cube geometry 136
Listing 5.23 Building the model–view matrix for a spinning cube 137
Listing 5.24 Updating the projection matrix for the spinning cube 137
Listing 5.25 Rendering loop for the spinning cube 137
Listing 5.26 Spinning cube vertex shader 138
Listing 5.27 Spinning cube fragment shader 138
Listing 5.28 Rendering loop for the spinning cube 139
Listing 5.29 Example shader storage block declaration 141
Listing 5.30 Using a shader storage block in place of vertex attributes 142
Listing 5.31 Setting up an atomic counter buffer 148
Listing 5.32 Setting up an atomic counter buffer 148
Listing 5.33 Counting area using an atomic counter 150
Listing 5.34 Using the result of an atomic counter in a uniform block 151
Listing 5.35 Generating, initializing, and binding a texture 152
Listing 5.36 Updating texture data with **glTexSubImage2D()** 153
Listing 5.37 Reading from a texture in GLSL 156
Listing 5.38 The header of a .KTX file 160
Listing 5.39 Loading a .KTX file 161
Listing 5.40 Vertex shader with a single texture coordinate 162
Listing 5.41 Fragment shader with a single texture coordinate . . 163

Listing 5.42 Initializing an array texture 178
Listing 5.43 Vertex shader for the alien rain sample 179
Listing 5.44 Fragment shader for the alien rain sample 180
Listing 5.45 Rendering loop for the alien rain sample 181
Listing 5.46 Fragment shader performing image loads and stores 188
Listing 5.47 Filling a linked list in a fragment shader 191
Listing 5.48 Traversing a linked list in a fragment shader 193

Listing 6.1 Retrieving the compiler log from a shader 221
Listing 6.2 Fragment shader with external function declaration 224
Listing 6.3 Configuring a separable program pipeline 226
Listing 6.4 Printing interface information 230
Listing 6.5 Example subroutine uniform declaration 231
Listing 6.6 Setting values of subroutine uniforms 234
Listing 6.7 Retrieving a program binary 236

Listing 7.1 Declaration of multiple vertex attributes 243
Listing 7.2 Setting up indexed cube geometry 251
Listing 7.3 Drawing indexed cube geometry 252
Listing 7.4 Drawing the same geometry many times 256
Listing 7.5 Pseudocode for **glDrawArraysInstanced()** 258
Listing 7.6 Pseudocode for **glDrawElementsInstanced()** 258
Listing 7.7 Simple vertex shader with per-vertex color 264
Listing 7.8 Simple instanced vertex shader 265
Listing 7.9 Getting ready for instanced rendering 266
Listing 7.10 Example use of an indirect draw command 272
Listing 7.11 Setting up the indirect draw buffer for asteroids . . . 273
Listing 7.12 Vertex shader inputs for asteroids 274
Listing 7.13 Per-indirect draw attribute setup 274
Listing 7.14 Asteroid field vertex shader 275
Listing 7.15 Drawing asteroids 276
Listing 7.16 Spring mass system vertex setup 288
Listing 7.17 Spring mass system vertex shader 291
Listing 7.18 Spring mass system iteration loop 293
Listing 7.19 Spring mass system rendering loop 294
Listing 7.20 Clipping an object against a plane and a sphere . . . 302

Listing 8.1 Simple quad tessellation control shader example . . . 308
Listing 8.2 Simple quad tessellation evaluation shader example 310
Listing 8.3 Simple triangle tessellation control shader example 311
Listing 8.4 Simple triangle tessellation evaluation shader example 312

Listing 8.5 Simple isoline tessellation control shader example . . 313
Listing 8.6 Simple isoline tessellation evaluation shader example 313
Listing 8.7 Isoline spirals tessellation evaluation shader 315
Listing 8.8 Vertex shader for terrain rendering 324
Listing 8.9 Tessellation control shader for terrain rendering . . . 325
Listing 8.10 Tessellation evaluation shader for terrain rendering 326
Listing 8.11 Fragment shader for terrain rendering 327
Listing 8.12 Cubic Bézier patch vertex shader 329
Listing 8.13 Cubic Bézier patch tessellation control shader 330
Listing 8.14 Cubic Bézier patch tessellation evaluation shader . . 331
Listing 8.15 Cubic Bézier patch fragment shader 332
Listing 8.16 Source code for a simple geometry shader 334
Listing 8.17 Geometry shader layout qualifiers 334
Listing 8.18 Iterating over the elements of `gl_in[]` 335
Listing 8.19 Definition of `gl_in[]` 338
Listing 8.20 Configuring the custom culling geometry shader . . . 342
Listing 8.21 Finding a face normal in a geometry shader 343
Listing 8.22 Conditionally emitting geometry in
a geometry shader 343
Listing 8.23 Setting up the “explode” geometry shader 345
Listing 8.24 Pushing a face out along its normal 345
Listing 8.25 Pass-through vertex shader 346
Listing 8.26 Setting up the “tessellator” geometry shader 347
Listing 8.27 Generating new vertices in a geometry shader 347
Listing 8.28 Emitting a single triangle from a geometry shader . . 347
Listing 8.29 Using a function to produce faces in
a geometry shader 348
Listing 8.30 A pass-through vertex shader that includes normals 349
Listing 8.31 Setting up the “normal visualizer”
geometry shader 350
Listing 8.32 Producing lines from normals in the geometry shader 350
Listing 8.33 Drawing a face normal in the geometry shader 351
Listing 8.34 Geometry shader for rendering quads 359
Listing 8.35 Fragment shader for rendering quads 360
Listing 8.36 Rendering to multiple viewports in
a geometry shader 362

Listing 9.1 Setting up scissor rectangle arrays 371
Listing 9.2 Example stencil buffer usage and stencil
border decorations 375
Listing 9.3 Rendering with all blending functions 384
Listing 9.4 Setting up a simple framebuffer object 393
Listing 9.5 Rendering to a texture 394

Listing 9.6 Setting up an FBO with multiple attachments 396
Listing 9.7 Declaring multiple outputs in a fragment shader . . . 396
Listing 9.8 Setting up a layered framebuffer 397
Listing 9.9 Layered rendering using a geometry shader 398
Listing 9.10 Displaying an array texture—vertex shader 399
Listing 9.11 Displaying an array texture—fragment shader 400
Listing 9.12 Attaching texture layers to a framebuffer 402
Listing 9.13 Checking completeness of a framebuffer object 406
Listing 9.14 Creating a stereo window 408
Listing 9.15 Drawing into a stereo window 408
Listing 9.16 Rendering to two layers with a geometry shader . . . 410
Listing 9.17 Copying from an array texture to a stereo back buffer 411
Listing 9.18 Turning on line smoothing 414
Listing 9.19 Choosing eight-sample antialiasing 416
Listing 9.20 Setting up a multi-sample framebuffer attachment . . 418
Listing 9.21 Simple multi-sample “maximum” resolve 419
Listing 9.22 Fragment shader producing high-frequency output 422
Listing 9.23 A 100-megapixel virtual framebuffer 430
Listing 9.24 Applying a simple exposure coefficient to an HDR image 435
Listing 9.25 Adaptive HDR to LDR conversion fragment shader . . 437
Listing 9.26 Bloom fragment shader—output bright data to a separate buffer 440
Listing 9.27 Blur fragment shader 442
Listing 9.28 Adding a bloom effect to the scene 443
Listing 9.29 Creating integer framebuffer attachments 445
Listing 9.30 Texturing a point sprite in the fragment shader 450
Listing 9.31 Vertex shader for the star field effect 452
Listing 9.32 Fragment shader for the star field effect 453
Listing 9.33 Fragment shader for generating shaped points 455
Listing 9.34 Naïve rotated point sprite fragment shader 456
Listing 9.35 Rotated point sprite vertex shader 457
Listing 9.36 Rotated point sprite fragment shader 457
Listing 9.37 Taking a screenshot with **glReadPixels()** 460

Listing 10.1 Creating and compiling a compute shader 468
Listing 10.2 Compute shader image inversion 473
Listing 10.3 Dispatching the image copy compute shader 474
Listing 10.4 Compute shader with race conditions 476
Listing 10.5 Simple prefix sum implementation in C++ 480
Listing 10.6 Prefix sum implementation using a compute shader . 483
Listing 10.7 Compute shader to generate a 2D prefix sum 485
Listing 10.8 Depth of field using summed area tables 489

Listing 10.9 Initializing shader storage buffers for flocking 494
Listing 10.10 The rendering loop for the flocking example 495
Listing 10.11 Compute shader for updates in flocking example . . . 496
Listing 10.12 The first rule of flocking 497
Listing 10.13 The second rule of flocking 497
Listing 10.14 Main body of the flocking update compute shader . . 498
Listing 10.15 Inputs to the flock rendering vertex shader 500
Listing 10.16 Flocking vertex shader body 500

Listing 11.1 Declaring samplers inside uniform blocks 505
Listing 11.2 Declaring samplers inside uniform blocks 506
Listing 11.3 Making textures resident 507
Listing 11.4 Creating a sparse texture 510
Listing 11.5 Determining supported sparse texture page sizes . . . 511
Listing 11.6 Simple texture commitment management 513
Listing 11.7 Sampling a sparse texture with clamped level-of-detail (LoD) 515
Listing 11.8 Fetching a block of texels using `textureGatherOffset` 520
Listing 11.9 Generating a palette for RGTC encoding 521
Listing 11.10 Palettizing an RGTC block 521
Listing 11.11 Packing an RGTC block 522
Listing 11.12 Generating a palette for RGTC encoding 523
Listing 11.13 Main function for RGTC compression 524
Listing 11.14 High-quality texture filtering function 530

Listing 12.1 Getting the result from a query object 538
Listing 12.2 Figuring out if occlusion query results are ready . . . 538
Listing 12.3 Simple, application-side conditional rendering 539
Listing 12.4 Rendering when query results aren't available 540
Listing 12.5 Basic conditional rendering example 541
Listing 12.6 A more complete conditional rendering example . . . 543
Listing 12.7 Simple control flow using queries 544
Listing 12.8 Timing operations using timer queries 546
Listing 12.9 Timing operations using **`glQueryCounter()`** 547
Listing 12.10 Drawing data written to a transform feedback buffer 552
Listing 12.11 Working while waiting for a sync object 558

Listing 13.1 The Gouraud shading vertex shader 571
Listing 13.2 The Gouraud shading fragment shader 572
Listing 13.3 The Phong shading vertex shader 574
Listing 13.4 The Phong shading fragment shader 575

Listing 13.5 Blinn-Phong fragment shader 578
Listing 13.6 Rim lighting shader function 580
Listing 13.7 Vertex shader for normal mapping 584
Listing 13.8 Fragment shader for normal mapping 585
Listing 13.9 Spherical environment mapping vertex shader 588
Listing 13.10 Spherical environment mapping fragment shader . . 588
Listing 13.11 Equirectangular environment mapping fragment shader 590
Listing 13.12 Loading a cubemap texture 592
Listing 13.13 Vertex shader for skybox rendering 594
Listing 13.14 Fragment shader for skybox rendering 595
Listing 13.15 Vertex shader for cubemap environment rendering 595
Listing 13.16 Fragment shader for cubemap environment rendering 596
Listing 13.17 Fragment shader for per-fragment shininess 598
Listing 13.18 Getting ready for shadow mapping 600
Listing 13.19 Setting up matrices for shadow mapping 601
Listing 13.20 Setting up a shadow matrix 602
Listing 13.21 Simplified vertex shader for shadow mapping 603
Listing 13.22 Simplified fragment shader for shadow mapping . . . 604
Listing 13.23 Displacement map tessellation evaluation shader . . 606
Listing 13.24 Application of fog in a fragment shader 608
Listing 13.25 The toon vertex shader 611
Listing 13.26 The toon fragment shader 612
Listing 13.27 Initializing a G-buffer 615
Listing 13.28 Writing to a G-buffer 616
Listing 13.29 Unpacking data from a G-buffer 617
Listing 13.30 Lighting a fragment using data from a G-buffer 618
Listing 13.31 Deferred shading with normal mapping (fragment shader) 621
Listing 13.32 Ambient occlusion fragment shader 630
Listing 13.33 Setting up the Julia set renderer 633
Listing 13.34 Inner loop of the Julia renderer 633
Listing 13.35 Using a gradient texture to color the Julia set 634
Listing 13.36 Ray–sphere interesection test 638
Listing 13.37 Determining closest intersection point 639
Listing 13.38 Ray–plane interesection test 644
Listing 13.39 Fragment shader for distance field rendering 651
Listing 13.40 Getting ready for bitmap fonts 656
Listing 13.41 Bitmap font rendering shader 657
Listing 13.42 Bitmap font rendering shader 658

Listing 14.1 OpenMP particle updater 663
Listing 14.2 Setting up a persistent mapped buffer 665

Listing 14.3 OpenMP particle rendering loop 665
Listing 14.4 Example packet data structure 669
Listing 14.5 Union of all packets 670
Listing 14.6 Appending a packet to a packet buffer 670
Listing 14.7 Executing a packet buffer 671
Listing 14.8 Disassembly of `packet_stream::execute` 672
Listing 14.9 Implementation of `packet_stream::EnableDisable` 673
Listing 14.10 Optimized packet insertion 676
Listing 14.11 Optimized packet execution 677
Listing 14.12 Uniform block declarations for indirect materials . . 680
Listing 14.13 Passing material index through a vertex shader 681
Listing 14.14 Declaration of material properties 681
Listing 14.15 Passing material index through a vertex shader 682
Listing 14.16 Candidate draws used for culling 685
Listing 14.17 Matrix data used for compute shader culling 685
Listing 14.18 Object culling in a compute shader 686
Listing 14.19 Driving compute culling shaders 687
Listing 14.20 Julia fractals on the CPU 696
Listing 14.21 Persistent mapped fractal rendering 697

Listing 15.1 Creating a debug context with the `sb7` framework . . 730
Listing 15.2 Setting the debug callback function 731
Listing 15.3 Shader with an infinite loop 738

Foreword

When OpenGL was young, the highest-end SGI systems like the Reality Engine 2 cost $80,000 and could render 200,000 textured triangles per second, or 3,333 triangles per frame at 60 Hz. The CPUs of that era were slower than today, to be sure, but at around 100 MHz, that's still 500 CPU cycles for each triangle. It was pretty easy to be graphics limited back then, and the API reflected that—the only way to specify geometry was immediate mode! Well, there were also display lists for static geometry, which made being graphics-limited even easier.

OpenGL is not young anymore, the highest-end GPUs that it can run on cost around $1000, and they don't even list triangles per second in their basic product description anymore, but the number is north of 6 billion. Today these GPUs are in the middle of the single digit teraflops and several hundred gigabytes per second of bandwidth. CPUs have gotten faster, too: With 4 cores and around 3 GHz, they are shy of 200 gigaflops and have around 20 gigabytes per second of memory bandwidth. So where we had 500 CPU cycles for a triangle in the early days, we now have 0.5 cycles. Even if we could perfectly exploit all 4 cores, that would give us a paltry 2 CPU cycles for each triangle!

All that is to say that the growth in hardware graphics performance has outstripped conventional CPU performance growth by several orders of magnitude, and the consequences are pretty obvious today. Not only is the CPU frequently the limiting factor in graphics performance, we have an API that was designed against a different set of assumptions.

The good news with OpenGL is that it has evolved too. First it added vertex arrays so that a single draw command with fairly low CPU overhead gets amplified into a lot of GPU work. This helped for a while, but it wasn't enough. We added instancing to further increase the amount of work, but this was a somewhat limited form of work amplification, as we don't always want many instances of the same object in an organic, believable rendering.

Recognizing that these emerging limitations in the API had to be circumvented somehow, OpenGL designers began extending the interface to remove as much CPU-side overhead from the interface as possible. The "bindless" family of extensions allows the GPU to reference buffers and textures directly rather than going through expensive binding calls in the driver. Persistent maps allow the application to scribble on memory at the same time the GPU is referencing it. This sounds dangerous—and it can be!—but allowing the application to manage memory hazards relieves a tremendous burden from the driver and allows for far simpler, less general mechanisms to be employed. Sparse texture arrays allow applications to manage texture memory as well with similar, very low-overhead benefits. And finally multi-draw and multi-draw indirect added means the GPU can generate the very buffers that it sources for drawing, leaving the CPU a lot more available for other work.

All of these advances in OpenGL have been loosely lumped under the *AZDO* (Approaching Zero Driver Overhead) umbrella, and most of them have been incorporated into the core API. There are still significant areas for improvement as we try to get to an API that allows developers to render as much as they want, the way they want, without worrying that the CPU or driver overhead will get in the way. These features require a bit more work to make use of, but the results can be truly amazing! This edition of the *OpenGL® SuperBible* includes many new examples that make use of AZDO features and provide good guidance on how to get the CPU out of the way. In particular, you'll learn good ways to make use of zero copy, proper fencing, and bindless.

Cass Everitt
Oculus

Preface

This book is designed both for people who are learning computer graphics through OpenGL and for people who may already know about graphics but want to learn about OpenGL. The intended audience is students of computer science, computer graphics, or game design; professional software engineers; or simply just hobbyists and people who are interested in learning something new. We begin by assuming that the reader knows nothing about either computer graphics or OpenGL. The reader should be familiar with computer programming in C++, however.

One of our goals with this book was to ensure that there were as few forward references as possible and to require little or no assumed knowledge. The book is accessible and readable, and if you start from the beginning and read all the way through, you should come away with a good comprehension of how OpenGL works and how to use it effectively in your applications. After reading and understanding the content of this book, you will be well positioned to read and learn from more advanced computer graphics research articles and confident that you can take the principles that they cover and implement them in OpenGL.

It is *not* a goal of this book to cover every last feature of OpenGL—that is, to mention every function in the specification or every value that can be passed to a command. Rather, we intend to provide a solid understanding of OpenGL, introduce the fundamentals, and explore some of its more advanced features. After reading this book, readers should be comfortable looking up finer details in the OpenGL specification, experimenting with

OpenGL on their own machines, and using extensions (bonus features that add capabilities to OpenGL not required by the main specification).

The Architecture of the Book

This book is subdivided into three parts. In Part I, “Foundations,” we explain what OpenGL is and how it connects to the graphics pipeline, and we give minimal working examples that are sufficient to demonstrate each section of it without requiring much, if any, knowledge of any other part of the whole system. We lay a foundation for the math behind three-dimensional computer graphics, and describe how OpenGL manages the large amounts of data that are required to provide a compelling experience to the users of such applications. We also describe the programming model for *shaders*, which form a core part of any OpenGL application.

In Part II, “In Depth,” we introduce features of OpenGL that require some knowledge of multiple parts of the graphics pipeline and may refer to concepts mentioned in Part I. This allows us to cover more complex topics without glossing over details or telling you to skip forward in the book to find out how something really works. By taking a second pass over the OpenGL system, we are able to delve into where data goes as it leaves each part of OpenGL, as you’ll already have been (at least briefly) introduced to its destination.

Finally, in Part III, “In Practice,” we dive deeper into the graphics pipeline, cover some more advanced topics, and give a number of examples that use multiple features of OpenGL. We provide a number of worked examples that implement various rendering techniques, give a series of suggestions and advice on OpenGL best practices and performance considerations, and end up with a practical overview of OpenGL on several popular platforms, including mobile devices.

In Part I, we start gently and then blast through OpenGL to give you a taste of what’s to come. Then, we lay the groundwork of knowledge that will be essential to you as you progress through the rest of the book. In this part, you will find the following chapters:

- Chapter 1, “Introduction,” provides a brief introduction to OpenGL, including its origins, history, and current state.
- Chapter 2, “Our First OpenGL Program,” jumps right into OpenGL and shows you how to create a simple OpenGL application using the source code provided with this book.

- Chapter 3, “Following the Pipeline,” takes a more careful look at OpenGL and its various components, introducing each in a little more detail and adding to the simple example presented in the previous chapter.
- Chapter 4, “Math for 3D Graphics,” introduces the foundations of math that is essential for effective use of OpenGL and the creation of interesting 3D graphics applications.
- Chapter 5, “Data,” provides you with the tools necessary to manage data that will be consumed and produced by OpenGL.
- Chapter 6, “Shaders and Programs,” takes a deeper look at *shaders*, which are fundamental to the operation of modern graphics applications.

In Part II, we take a more detailed look at several of the topics introduced in the first chapters. We dig deeper into each of the major parts of OpenGL and our example applications start to become a little more complex and interesting. In this part, you will find these six chapters:

- Chapter 7, “Vertex Processing and Drawing Commands,” covers the inputs to OpenGL and the mechanisms by which semantics are applied to the raw data you provide.
- Chapter 8, “Primitive Processing,” covers some higher-level concepts in OpenGL, including connectivity information, higher-order surfaces, and tessellation.
- Chapter 9, “Fragment Processing and the Framebuffer,” looks at how high-level 3D graphics information is transformed by OpenGL into 2D images, and how your applications can determine the appearance of objects on the screen.
- Chapter 10, “Compute Shaders,” illustrates how your applications can harness OpenGL for more than just graphics and make use of the incredible computing power locked up in a modern graphics card.
- Chapter 11, “Advanced Data Management,” discusses topics related to managing large data sets, loading data efficiently, and arbitrating access to that data once loaded.
- Chapter 12, “Controlling and Monitoring the Pipeline,” shows you how to get a glimpse into how OpenGL executes the commands you give it—including how long they take to execute, and how much data they produce.

In Part III, we build on the knowledge that you will have gained in reading the first two parts of the book and use it to construct example applications that touch on multiple aspects of OpenGL. We also get into the practicalities of building larger OpenGL applications and deploying them across multiple platforms. In this part, you will find three chapters:

- Chapter 13, "Rendering Techniques," covers several applications of OpenGL for graphics rendering, including simulation of light, artistic methods and even some nontraditional techniques.
- Chapter 14, "High-Performance OpenGL," digs into some topics related to getting the highest possible performance from OpenGL.
- Chapter 15, "Debugging and Stability," provides advice and tips on how to get your applications running without errors and how to debug problems with your programs.

Finally, several appendices are provided that describe the tools and file formats used in this book, discuss which versions of OpenGL support which features and list which extensions introduced those features, and give pointers to more useful OpenGL resources.

What's New in This Edition

In this book, we have expanded on the sixth edition to cover new features and topics introduced in OpenGL in versions 4.4 and 4.5 of the API. In the previous edition, we did not cover extensions—features that are entirely optional and not a mandatory part of the OpenGL core—and so left out a number of interesting topics. Since the release of the sixth edition of this book, some of these extensions have become fairly ubiquitous; in turn, we have decided to cover the ARB and KHR extensions. Thus extensions that have been ratified by Khronos (the OpenGL governing body) are part of this book.

We have built on the previous edition by expanding the book's application framework and adding new chapters and appendices that provide further insight and cover new topics. One important set of features enabled by the extensions that are now part of the book are the AZDO (Approaching Zero Driver Overhead) features, which are a way of using OpenGL that produces very low software overhead and correspondingly high performance. These features include *persistent maps* and *bindless textures*.

To make room for the new content, we decided to remove the chapter on platform specifics, which covered per-platform window system bindings. Also gone is official support for the Apple Mac platform. Almost all of the new content in this edition requires features introduced with OpenGL 4.4 or 4.5, or recent OpenGL extensions—none of which were supported by OS X at the time of writing. There is no expectation that Apple will further invest in its OpenGL implementation, so we encourage our readers to move away from the platform. To support multiple platforms, we recommend the use of cross-platform toolkits such as the excellent `SDL` (`https://www.libsdl.org/`) or `glfw` (`http://www.glfw.org/`) libraries. In fact, this book's framework is built on `glfw`, and it works well for us.

This book includes several new example applications, including demonstrations of new features, a texture compressor, text drawing, font rendering using distance fields, high-quality texture filtering, and multi-threaded programs using OpenMP. We also tried to address all of the errata and feedback we've received from our readers since the publication of the previous edition. We believe this to be the best update yet to the *OpenGL® SuperBible* yet.

We hope you enjoy it.

How to Build the Examples

Retrieve the sample code from the book's companion Web site, `http://www.openglsuperbible.com`, unpack the archive to a directory on your computer, and follow the instructions in the included `HOWTOBUILD.TXT` file for your platform of choice. The book's source code has been built and tested on Microsoft Windows (Windows 7 or later is required) and Linux (several major distributions). It is recommended that you install any available operating system updates and obtain the most recent graphics drivers from your graphics card manufacturer.

You may notice some minor discrepancies between the source code printed in this book and that in the source files. There are a number of reasons for this:

- This book is about OpenGL 4.5—the most recent version at the time of writing. The examples printed in the book are written assuming that OpenGL 4.5 is available on the target platform. However, we understand that in practice, operating systems, graphics drivers, and platforms may not have the *latest and greatest* available.

Consequently, where possible, we've made minor modifications to the example applications to allow them to run on earlier versions of OpenGL.

- Several months passed between when this book's text was finalized for printing and when the sample applications were packaged and posted to the Web. In that time, we discovered opportunities for improvement, whether that was uncovering new bugs, platform dependencies, or optimizations. The latest version of the source code on the Web will have those fixes and tweaks applied and will therefore deviate from the necessarily static copy printed in the book.
- There is not necessarily a one-to-one mapping of listings in the book's text and example applications in the Web package. Some example applications demonstrate more than one concept, some aren't mentioned in the book at all, and some listings in the book don't have an equivalent example application. Where possible, we've mentioned which of the example applications correspond to the listings in the book. We recommend that the reader take a close look at the example application package, as it includes some nuggets that may not be mentioned in the book.

Errata

We made a bunch of mistakes—we're certain of it. It's incredibly frustrating as an author to spot an error that you made and know that it has been printed, in books that your readers paid for, thousands and thousands of times. We have to accept that this will happen, though, and do our best to correct issues as we are able. If you think you see something that doesn't quite gel, check the book's Web site for errata:

`http://www.openglsuperbible.com`

Note from the Publisher

Some of the figures in the print edition of the book are dark due to the nature of the images themselves. To assist readers, color PDFs of figures are freely available at http://www.openglsuperbible.com and http://informit.com/title/9780672337475. In addition, PowerPoint slides of the figures for professors' classroom use are available at www.pearsonhighered.com/educator/product/OpenGL-Superbible-Comprehensive-Tutorial-and-Reference/9780672337475.page.

Acknowledgments

First, thanks to you—the reader. The best part of what I do is knowing that someone I've never met might benefit from all this. It's the biggest thrill, and the reason why people like me do this. I appreciate that you're reading this now and hope you get as much out of this book as I put into it.

I'd like to thank my wonderful wife, Chris, who's put up with me disappearing into my office for three editions of this book now. She's worked around my deadlines and cheered me on as I made (sometimes slow and painful) progress. I couldn't have done this without her. Thanks, too, to my kids, Jeremy and Emily. The answer to "What are you doing, dad?" is almost always "Working"—and you've always taken it in stride.

Thanks to my coauthors, Richard and Nick. You've let me run alone on this edition, but your names are on the cover because of your contributions—your fingerprints are etched into this book. Many thanks to Matías Goldberg, who performed a thorough technical review of the book on short notice.

Thanks again to Laura Lewin and Olivia Basegio and the Pearson team for letting me be me and just dropping random files and documents off whenever I felt like it. I don't work well with a plan, but seem to relish pressure and am really excellent at procrastination. I'm glad you guys put up with me.

Graham Sellers

This page intentionally left blank

About the Author

Graham Sellers is a classic geek. His family got their first computer (a BBC Model B) right before his sixth birthday. After his mum and dad stayed up all night programming it to play "Happy Birthday," he was hooked and determined to figure out how it worked. Next came basic programming and then assembly language. His first real exposure to graphics was via "demos" in the early 1990s, and then through Glide, and finally OpenGL in the late 1990s. Graham holds a master's in engineering from the University of Southampton, England.

Currently, Graham is a software architect at AMD. He represents AMD at the OpenGL ARB and has contributed to many extensions and to the core OpenGL Specification. Prior to that, he was a team lead at Epson, implementing OpenGL-ES and OpenVG drivers for embedded products. Graham holds several patents in the fields of computer graphics and image processing. When he's not working on OpenGL, he likes to disassemble and reverse-engineer old video game consoles (just to see how they work and what he can make them do). Originally from England, Graham now lives in Orlando, Florida, with his wife and two children.

This page intentionally left blank

Chapter 3

Following the Pipeline

WHAT YOU'LL LEARN IN THIS CHAPTER

- What each of the stages in the OpenGL pipeline does.
- How to connect your shaders to the fixed-function pipeline stages.
- How to create a program that uses every stage of the graphics pipeline simultaneously.

In this chapter, we will walk all the way along the OpenGL pipeline from start to finish, providing insight into each of the stages, which include fixed-function blocks and programmable shader blocks. You have already read a whirlwind introduction to the vertex and fragment shader stages. However, the application that you constructed simply drew a single triangle at a fixed position. If we want to render anything interesting with OpenGL, we're going to have to learn a lot more about the pipeline and all of the things you can do with it. This chapter introduces every part of the pipeline, hooks them up to one another, and provides an example shader for each stage.

Passing Data to the Vertex Shader

The vertex shader is the first *programmable* stage in the OpenGL pipeline and has the distinction of being the only mandatory stage in the graphics pipeline. However, before the vertex shader runs, a fixed-function stage known as *vertex fetching*, or sometimes *vertex pulling*, is run. This automatically provides inputs to the vertex shader.

Vertex Attributes

In GLSL, the mechanism for getting data in and out of shaders is to declare global variables with the `in` and `out` storage qualifiers. You were briefly introduced to the `out` qualifier in Chapter 2, "Our First OpenGL Program," when Listing 2.4 used it to output a color from the fragment shader. At the start of the OpenGL pipeline, we use the `in` keyword to bring inputs into the vertex shader. Between stages, `in` and `out` can be used to form conduits from shader to shader and pass data between them. We'll get to that shortly. For now, consider the input to the vertex shader and what happens if you declare a variable with an `in` storage qualifier. This marks the variable as an input to the vertex shader, which means that it is essentially an input to the OpenGL graphics pipeline. It is automatically filled in by the fixed-function vertex fetch stage. The variable becomes known as a *vertex attribute*.

Vertex attributes are how vertex data is introduced into the OpenGL pipeline. To declare a vertex attribute, you declare a variable in the vertex shader using the `in` storage qualifier. An example of this is shown in Listing 3.1, where we declare the variable `offset` as an input attribute.

```
#version 450 core

// 'offset' is an input vertex attribute
layout (location = 0) in vec4 offset;

void main(void)
{
    const vec4 vertices[3] = vec4[3](vec4( 0.25, -0.25, 0.5, 1.0),
                                     vec4(-0.25, -0.25, 0.5, 1.0),
                                     vec4( 0.25,  0.25, 0.5, 1.0));

    // Add 'offset' to our hard-coded vertex position
    gl_Position = vertices[gl_VertexID] + offset;
}
```

Listing 3.1: Declaration of a vertex attribute

In Listing 3.1, we have added the variable `offset` as an input to the vertex shader. As it is an input to the first shader in the pipeline, it will be filled automatically by the vertex fetch stage. We can tell this stage what to fill the variable with by using one of the many variants of the vertex attribute functions, **`glVertexAttrib*()`**. The prototype for **`glVertexAttrib4fv()`**, which we use in this example, is

```
void glVertexAttrib4fv(GLuint index,
                       const GLfloat * v);
```

Here, the parameter `index` is used to reference the attribute and `v` is a pointer to the new data to put into the attribute. You may have noticed the `layout (location = 0)` code in the declaration of the `offset` attribute. This is a *layout qualifier*, which we have used to set the *location* of the vertex attribute to zero. This location is the value we'll pass in `index` to refer to the attribute.

Each time we call one of the **`glVertexAttrib*()`** functions (of which there are many), it will update the value of the vertex attribute that is passed to the vertex shader. We can use this approach to animate our one triangle. Listing 3.2 shows an updated version of our rendering function that updates the value of `offset` in each frame.

```
// Our rendering function
virtual void render(double currentTime)
{
    const GLfloat color[] = { (float)sin(currentTime) * 0.5f + 0.5f,
                              (float)cos(currentTime) * 0.5f + 0.5f,
                              0.0f, 1.0f };
    glClearBufferfv(GL_COLOR, 0, color);

    // Use the program object we created earlier for rendering
    glUseProgram(rendering_program);

    GLfloat attrib[] = { (float)sin(currentTime) * 0.5f,
                         (float)cos(currentTime) * 0.6f,
                         0.0f, 0.0f };

    // Update the value of input attribute 0
    glVertexAttrib4fv(0, attrib);

    // Draw one triangle
    glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

Listing 3.2: Updating a vertex attribute

When we run the program with the rendering function of Listing 3.2, the triangle will move in a smooth oval shape around the window.

Passing Data from Stage to Stage

So far, you have seen how to pass data into a vertex shader by creating a vertex attribute using the `in` keyword, how to communicate with fixed-function blocks by reading and writing built-in variables such as `gl_VertexID` and `gl_Position`, and how to output data from the fragment shader using the `out` keyword. However, it's also possible to send your own data from shader stage to shader stage using the same `in` and `out` keywords. Just as you used the `out` keyword in the fragment shader to create the output variable to which it writes its color values, so you can also create an output variable in the vertex shader by using the `out` keyword. Anything you write to an output variable in one shader is sent to a similarly named variable declared with the `in` keyword in the subsequent stage. For example, if your vertex shader declares a variable called `vs_color` using the `out` keyword, it would match up with a variable named `vs_color` declared with the `in` keyword in the fragment shader stage (assuming no other stages were active in between).

If we modify our simple vertex shader as shown in Listing 3.3 to include `vs_color` as an output variable, and correspondingly modify our simple fragment shader to include `vs_color` as an input variable as shown in Listing 3.4, we can pass a value from the vertex shader to the fragment shader. Then, rather than outputting a hard-coded value, the fragment can simply output the color passed to it from the vertex shader.

```
#version 450 core

// 'offset' and 'color' are input vertex attributes
layout (location = 0) in vec4 offset;
layout (location = 1) in vec4 color;

// 'vs_color' is an output that will be sent to the next shader stage
out vec4 vs_color;

void main(void)
{
    const vec4 vertices[3] = vec4[3](vec4( 0.25, -0.25, 0.5, 1.0),
                                     vec4(-0.25, -0.25, 0.5, 1.0),
                                     vec4( 0.25,  0.25, 0.5, 1.0));

    // Add 'offset' to our hard-coded vertex position
    gl_Position = vertices[gl_VertexID] + offset;

    // Output a fixed value for vs_color
    vs_color = color;
}
```

Listing 3.3: Vertex shader with an output

As you can see in Listing 3.3, we declare a second input to our vertex shader, `color` (this time at location 1), and write its value to the `vs_output` output. This is picked up by the fragment shader of Listing 3.4 and written to the framebuffer. This allows us to pass a color all the way from a vertex attribute that we can set with **`glVertexAttrib*()`** through the vertex shader, into the fragment shader, and out to the framebuffer. As a consequence, we can draw different-colored triangles!

```
#version 450 core

// Input from the vertex shader
in vec4 vs_color;

// Output to the framebuffer
out vec4 color;

void main(void)
{
   // Simply assign the color we were given by the vertex shader to our output
   color = vs_color;
}
```

Listing 3.4: Fragment shader with an input

Interface Blocks

Declaring interface variables one at a time is possibly the simplest way to communicate data between shader stages. However, in most nontrivial applications, you will likely want to communicate a number of different pieces of data between stages; these may include arrays, structures, and other complex arrangements of variables. To achieve this, we can group together a number of variables into an *interface block*. The declaration of an interface block looks a lot like a structure declaration, except that it is declared using the `in` or `out` keyword depending on whether it is an input to or output from the shader. An example interface block definition is shown in Listing 3.5.

```
#version 450 core

// 'offset' is an input vertex attribute
layout (location = 0) in vec4 offset;
layout (location = 1) in vec4 color;

// Declare VS_OUT as an output interface block
out VS_OUT
{
    vec4 color;     // Send color to the next stage
} vs_out;
```

```
void main(void)
{
    const vec4 vertices[3] = vec4[3](vec4( 0.25, -0.25, 0.5, 1.0),
                                     vec4(-0.25, -0.25, 0.5, 1.0),
                                     vec4( 0.25,  0.25, 0.5, 1.0));

    // Add 'offset' to our hard-coded vertex position
    gl_Position = vertices[gl_VertexID] + offset;

    // Output a fixed value for vs_color
    vs_out.color = color;
}
```

Listing 3.5: Vertex shader with an output interface block

Note that the interface block in Listing 3.5 has both a block name (`VS_OUT`, uppercase) and an instance name (`vs_out`, lowercase). Interface blocks are matched between stages using the block name (`VS_OUT` in this case), but are referenced in shaders using the instance name. Thus, modifying our fragment shader to use an interface block gives the code shown in Listing 3.6.

```
#version 450 core

// Declare VS_OUT as an input interface block
in VS_OUT
{
    vec4 color;     // Send color to the next stage
} fs_in;

// Output to the framebuffer
out vec4 color;

void main(void)
{
    // Simply assign the color we were given by the vertex shader to our output
    color = fs_in.color;
}
```

Listing 3.6: Fragment shader with an input interface block

Matching interface blocks by block name but allowing block instances to have different names in each shader stage serves two important purposes. First, it allows the name by which you refer to the block to be different in each stage, thereby avoiding confusing things such as having to use `vs_out` in a fragment shader. Second, it allows interfaces to go from being single items to arrays when crossing between certain shader stages, such as the vertex and tessellation or geometry shader stages, as we will see in a short while. Note that interface blocks are only for moving data from shader stage to shader stage—you can't use them to group together inputs to the vertex shader or outputs from the fragment shader.

Tessellation

Tessellation is the process of breaking a high-order primitive (which is known as a *patch* in OpenGL) into many smaller, simpler primitives such as triangles for rendering. OpenGL includes a fixed-function, configurable tessellation engine that is able to break up quadrilaterals, triangles, and lines into a potentially large number of smaller points, lines, or triangles that can be directly consumed by the normal rasterization hardware further down the pipeline. Logically, the tessellation phase sits directly after the vertex shading stage in the OpenGL pipeline and is made up of three parts: the tessellation control shader, the fixed-function tessellation engine, and the tessellation evaluation shader.

Tessellation Control Shaders

The first of the three tessellation phases is the tessellation control shader (TCS; sometimes known as simply the control shader). This shader takes its input from the vertex shader and is primarily responsible for two things: the determination of the level of tessellation that will be sent to the tessellation engine, and the generation of data that will be sent to the tessellation evaluation shader that is run after tessellation has occurred.

Tessellation in OpenGL works by breaking down high-order surfaces known as *patches* into points, lines, or triangles. Each patch is formed from a number of *control points*. The number of control points per patch is configurable and set by calling **`glPatchParameteri()`** with `pname` set to `GL_PATCH_VERTICES` and `value` set to the number of control points that will be used to construct each patch. The prototype of **`glPatchParameteri()`** is

```
void glPatchParameteri(GLenum pname,
                       GLint value);
```

By default, the number of control points per patch is three. Thus, if this is what you want (as in our example application), you don't need to call it at all. The maximum number of control points that can be used to form a single patch is implementation defined, but is guaranteed to be at least 32.

When tessellation is active, the vertex shader runs once per control point, while the tessellation control shader runs in batches on groups of control points where the size of each batch is the same as the number of vertices per patch. That is, vertices are used as control points and the result of the vertex shader is passed in batches to the tessellation control shader as its input. The number of control points per patch can be changed such that the number of control points that is output by the tessellation control shader can differ from the number of control points that it consumes. The number of control points produced by the control shader is set using an output layout qualifier in the control shader's source code. Such a layout qualifier looks like this:

```
layout (vertices = N) out;
```

Here, `N` is the number of control points per patch. The control shader is responsible for calculating the values of the output control points and for setting the tessellation factors for the resulting patch that will be sent to the fixed-function tessellation engine. The output tessellation factors are written to the `gl_TessLevelInner` and `gl_TessLevelOuter` built-in output variables, whereas any other data that is passed down the pipeline is written to user-defined output variables (those declared using the **`out`** keyword, or the special built-in `gl_out` array) as normal.

Listing 3.7 shows a simple tessellation control shader. It sets the number of output control points to three (the same as the default number of input control points) using the `layout (vertices = 3) out;` layout qualifier, copies its input to its output (using the built-in variables `gl_in` and `gl_out`), and sets the inner and outer tessellation level to 5. Higher numbers would produce a more densely tessellated output, and lower numbers would yield a more coarsely tessellated output. Setting the tessellation factor to 0 will cause the whole patch to be thrown away.

The built-in input variable `gl_InvocationID` is used as an index into the `gl_in` and `gl_out` arrays. This variable contains the zero-based index of the control point within the patch being processed by the current invocation of the tessellation control shader.

```
#version 450 core

layout (vertices = 3) out;

void main(void)
{
    // Only if I am invocation 0 ...
    if (gl_InvocationID == 0)
```

```
        {
            gl_TessLevelInner[0] = 5.0;
            gl_TessLevelOuter[0] = 5.0;
            gl_TessLevelOuter[1] = 5.0;
            gl_TessLevelOuter[2] = 5.0;
        }
        // Everybody copies their input to their output
        gl_out[gl_InvocationID].gl_Position =
            gl_in[gl_InvocationID].gl_Position;
    }
```

Listing 3.7: Our first tessellation control shader

The Tessellation Engine

The tessellation engine is a fixed-function part of the OpenGL pipeline that takes high-order surfaces represented as patches and breaks them down into simpler primitives such as points, lines, or triangles. Before the tessellation engine receives a patch, the tessellation control shader processes the incoming control points and sets tessellation factors that are used to break down the patch. After the tessellation engine produces the output primitives, the vertices representing them are picked up by the tessellation evaluation shader. The tessellation engine is responsible for producing the parameters that are fed to the invocations of the tessellation evaluation shader, which it then uses to transform the resulting primitives and get them ready for rasterization.

Tessellation Evaluation Shaders

Once the fixed-function tessellation engine has run, it produces a number of output vertices representing the primitives it has generated. These are passed to the tessellation evaluation shader. The tessellation evaluation shader (TES; also called simply the evaluation shader) runs an invocation for each vertex produced by the tessellator. When the tessellation levels are high, the tessellation evaluation shader could run an extremely large number of times. For this reason, you should be careful with complex evaluation shaders and high tessellation levels.

Listing 3.8 shows a tessellation evaluation shader that accepts input vertices produced by the tessellator as a result of running the control shader shown in Listing 3.7. At the beginning of the shader is a layout qualifier that sets the tessellation mode. In this case, we selected the mode to be triangles. Other qualifiers, equal_spacing and cw, indicate that new vertices should be generated equally spaced along the tessellated polygon edges and that a clockwise vertex winding order should be used for the generated triangles. We will cover the other possible choices in the "Tessellation" section in Chapter 8.

The remainder of the shader assigns a value to gl_Position just like a vertex shader does. It calculates this using the contents of two more built-in variables. The first is gl_TessCoord, which is the *barycentric coordinate* of the vertex generated by the tessellator. The second is the gl_Position member of the gl_in[] array of structures. This matches the gl_out structure written to in the tessellation control shader given in Listing 3.7. This shader essentially implements pass-through tessellation. That is, the tessellated output patch is exactly the same shape as the original, incoming triangular patch.

```
#version 450 core

layout (triangles, equal_spacing, cw) in;

void main(void)
{
    gl_Position = (gl_TessCoord.x * gl_in[0].gl_Position +
                   gl_TessCoord.y * gl_in[1].gl_Position +
                   gl_TessCoord.z * gl_in[2].gl_Position);
}
```

Listing 3.8: Our first tessellation evaluation shader

To see the results of the tessellator, we need to tell OpenGL to draw only the outlines of the resulting triangles. To do this, we call **glPolygonMode()**, whose prototype is

```
void glPolygonMode(GLenum face,
                   GLenum mode);
```

The face parameter specifies which type of polygons we want to affect. Because we want to affect everything, we set it to GL_FRONT_AND_BACK. The other modes will be explained shortly. mode says how we want our polygons to be rendered. As we want to render in wireframe mode (i.e., lines), we set this to GL_LINE. The result of rendering our one triangle

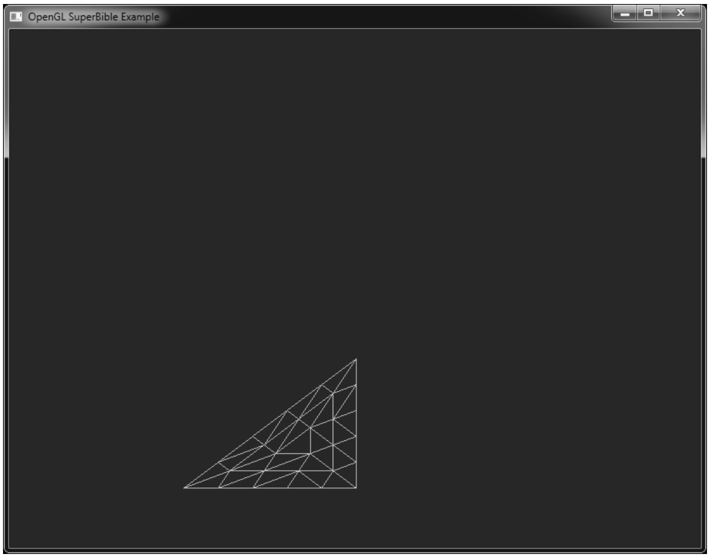


Figure 3.1: Our first tessellated triangle

example with tessellation enabled and the two shaders of Listing 3.7 and Listing 3.8 is shown in Figure 3.1.

Geometry Shaders

The geometry shader is logically the last shader stage in the front end, sitting after the vertex and tessellation stages and before the rasterizer. The geometry shader runs once per primitive and has access to all of the input vertex data for all of the vertices that make up the primitive being processed. The geometry shader is also unique among the shader stages in that it is able to increase or reduce the amount of data flowing through the pipeline in a programmatic way. Tessellation shaders can also increase or decrease the amount of work in the pipeline, but only implicitly by setting the tessellation level for the patch. Geometry shaders, in contrast, include two functions—`EmitVertex()` and `EndPrimitive()`—that explicitly produce vertices that are sent to primitive assembly and rasterization.

Another unique feature of geometry shaders is that they can change the primitive mode mid-pipeline. For example, they can take triangles as input and produce a bunch of points or lines as output, or even create triangles from independent points. An example geometry shader is shown in Listing 3.9.

```
#version 450 core

layout (triangles) in;
layout (points, max_vertices = 3) out;

void main(void)
{
    int i;

    for (i = 0; i < gl_in.length(); i++)
    {
        gl_Position = gl_in[i].gl_Position;
        EmitVertex();
    }
}
```

Listing 3.9: Our first geometry shader

The shader shown in Listing 3.9 acts as another simple pass-through shader that converts triangles into points so that we can see their vertices. The first layout qualifier indicates that the geometry shader is expecting to see triangles as its input. The second layout qualifier tells OpenGL that the geometry shader will produce points and that the maximum number of points that each shader will produce will be three. In the `main` function, a loop runs through all of the members of the `gl_in` array, which is determined by calling its `.length()` function.

We actually know that the length of the array will be three because we are processing triangles and every triangle has three vertices. The outputs of the geometry shader are again similar to those of a vertex shader. In particular, we write to `gl_Position` to set the position of the resulting vertex. Next, we call `EmitVertex()`, which produces a vertex at the output of the geometry shader. Geometry shaders automatically call `EndPrimitive()` at the end of your shader, so calling this function explicitly is not necessary in this example. As a result of running this shader, three vertices will be produced and rendered as points.

By inserting this geometry shader into our simple one tessellated triangle example, we obtain the output shown in Figure 3.2. To create this image, we set the point size to 5.0 by calling **`glPointSize()`**. This makes the points large and highly visible.

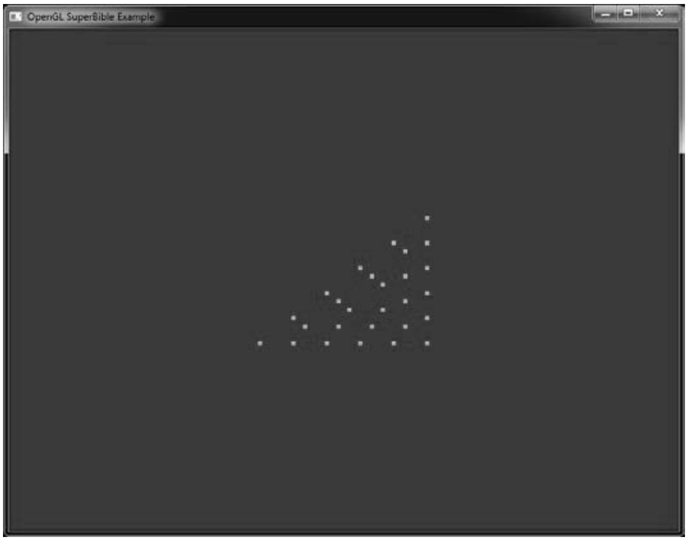


Figure 3.2: Tessellated triangle after adding a geometry shader

Primitive Assembly, Clipping, and Rasterization

After the front end of the pipeline has run (which includes vertex shading, tessellation, and geometry shading), a fixed-function part of the pipeline performs a series of tasks that take the vertex representation of our scene and convert it into a series of pixels, which in turn need to be colored and written to the screen. The first step in this process is primitive assembly, which is the grouping of vertices into lines and triangles. Primitive assembly still occurs for points, but it is trivial in that case.

Once primitives have been constructed from their individual vertices, they are *clipped* against the displayable region, which usually means the window or screen, but can also be a smaller area known as the *viewport*. Finally, the parts of the primitive that are determined to be potentially visible are sent to a fixed-function subsystem called the rasterizer. This block determines which pixels are covered by the primitive (point, line, or triangle) and sends the list of pixels on to the next stage—that is, fragment shading.

Clipping

As vertices exit the front end of the pipeline, their position is said to be in *clip space*. This is one of the many coordinate systems that can be used to represent positions. You may have noticed that the `gl_Position` variable that we have written to in our vertex, tessellation, and geometry shaders has a `vec4` type, and that the positions we have produced by writing to it are all four-component vectors. This is what is known as a *homogeneous* coordinate. The homogeneous coordinate system is used in projective geometry because much of the math ends up being simpler in homogeneous coordinate space than it does in regular Cartesian space. Homogeneous coordinates have one more component than their equivalent Cartesian coordinate, which is why our three-dimensional position vector is represented as a four-component variable.

Although the output of the front end is a four-component homogeneous coordinate, clipping occurs in Cartesian space. Thus, to convert from homogeneous coordinates to Cartesian coordinates, OpenGL performs a *perspective division*, which involves dividing all four components of the position by the last, w component. This has the effect of projecting the vertex from the homogeneous space to the Cartesian space, leaving w as 1.0. In all of the examples so far, we have set the w component of `gl_Position` as 1.0, so this division has not had any effect. When we explore projective geometry in a short while, we will discuss the effect of setting w to values other than 1.0.

After the projective division, the resulting position is in *normalized device space*. In OpenGL, the visible region of normalized device space is the volume that extends from -1.0 to 1.0 in the x and y dimensions and from 0.0 to 1.0 in the z dimension. Any geometry that is contained in this region may become visible to the user and anything outside of it should be discarded. The six sides of this volume are formed by planes in three-dimensional space. As a plane divides a coordinate space in two, the volumes on each side of the plane are called *half-spaces*.

Before passing primitives on to the next stage, OpenGL performs clipping by determining which side of each of these planes the vertices of each primitive lie on. Each plane effectively has an "outside" and an "inside." If a primitive's vertices all lie on the "outside" of any one plane, then the whole thing is thrown away. If all of primitive's vertices are on the "inside" of all the planes (and therefore inside the view volume), then it is passed through unaltered. Primitives that are partially visible (which means that they cross one of the planes) must be handled specially. More details about how this works is given in the "Clipping" section in Chapter 7.

Viewport Transformation

After clipping, all of the vertices of the geometry have coordinates that lie between -1.0 and 1.0 in the x and y dimensions. Along with a z coordinate that lies between 0.0 and 1.0, these are known as normalized device coordinates. However, the window that you're drawing to has coordinates that usually[1] start from $(0, 0)$ at the bottom left and range to $(w - 1, h - 1)$, where w and h are the width and height of the window in pixels, respectively. To place your geometry into the window, OpenGL applies the *viewport transform*, which applies a scale and offset to the vertices' normalized device coordinates to move them into *window coordinates*. The scale and bias to apply are determined by the viewport bounds, which you can set by calling **glViewport()** and **glDepthRange()**. Their prototypes are

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

and

```
void glDepthRange(GLdouble nearVal, GLdouble farVal);
```

This transform takes the following form:

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{p_x}{2} x_d + o_x \\ \frac{p_y}{2} y_d + o_y \\ \frac{f-n}{2} z_d + \frac{n+f}{2} \end{pmatrix}$$

Here, x_w, y_w, and z_w are the resulting coordinates of the vertex in window space, and x_d, y_d, and z_d are the incoming coordinates of the vertex in normalized device space. p_x and p_y are the width and height of the viewport in pixels, and n and f are the near and far plane distances in the z coordinate, respectively. Finally, o_x, o_y, and o_z are the origins of the viewport.

Culling

Before a triangle is processed further, it may be optionally passed through a stage called *culling*, which determines whether the triangle faces toward or away from the viewer and can decide whether to actually go ahead and draw it based on the result of this computation. If the triangle faces toward the viewer, then it is considered to be *front-facing*; otherwise, it is said to be *back-facing*. It is very common to discard triangles that are back-facing because when an object is closed, any back-facing triangle will be hidden by another front-facing triangle.

1. It's possible to change the coordinate convention such that the $(0, 0)$ origin is at the upper-left corner of the window, which matches the convention used in some other graphics systems.

To determine whether a triangle is front- or back-facing, OpenGL will determine its *signed* area in window space. One way to determine the area of a triangle is to take the cross product of two of its edges. The equation for this is

$$a = \frac{1}{2}\sum_{i=0}^{n-1} x_w^i y_w^{i\oplus 1} - x_w^{i\oplus 1} y_w^i$$

Here, x_w^i and y_w^i are the coordinates of the ith vertex of the triangle in window space and $i \oplus 1$ is $(i+1)$ mod 3. If the area is positive, then the triangle is considered to be front-facing; if it is negative, then it is considered to be back-facing. The sense of this computation can be reversed by calling **`glFrontFace()`** with `dir` set to either `GL_CW` or `GL_CCW` (where `CW` and `CCW` stand for clockwise and counterclockwise, respectively). This is known as the *winding order* of the triangle, and the clockwise or counterclockwise terms refer to the order in which the vertices appear in window space. By default, this state is set to `GL_CCW`, indicating that triangles whose vertices are in counterclockwise order are considered to be front-facing and those whose vertices are in clockwise order are considered to be back-facing. If the state is `GL_CW`, then a is simply negated before being used in the culling process. Figure 3.3 shows this pictorially for the purpose of illustration.

Once the direction that the triangle is facing has been determined, OpenGL is capable of discarding either front-facing, back-facing, or even both types of triangles. By default, OpenGL will render all triangles, regardless of which way they face. To turn on culling, call **`glEnable()`** with `cap` set to `GL_CULL_FACE`. When you enable culling, OpenGL will cull back-facing triangles by default. To change which types of triangles are

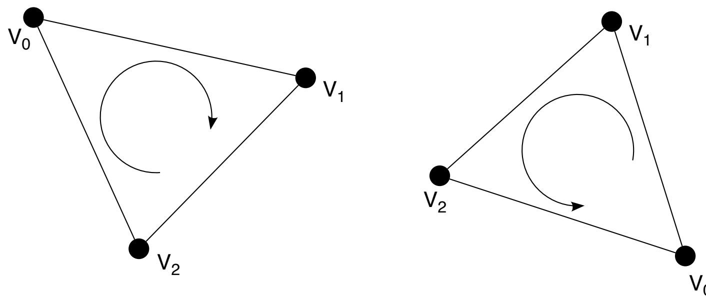


Figure 3.3: Clockwise (left) and counterclockwise (right) winding order

culled, call **`glCullFace()`** with `face` set to `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`.

As points and lines don't have any geometric area,[2] this facing calculation doesn't apply to them and they can't be culled at this stage.

Rasterization

Rasterization is the process of determining which fragments might be covered by a primitive such as a line or a triangle. There are myriad algorithms for doing this, but most OpenGL systems will settle on a half-space–based method for triangles, as it lends itself well to parallel implementation. Essentially, OpenGL will determine a bounding box for the triangle in window coordinates and test every fragment inside it to determine whether it is inside or outside the triangle. To do this, it treats each of the triangle's three edges as a half-space that divides the window in two.

Fragments that lie on the interior of all three edges are considered to be inside the triangle and fragments that lie on the exterior of any of the three edges are considered to be outside the triangle. Because the algorithm to determine which side of a line a point lies on is relatively simple and is independent of anything besides the position of the line's endpoints and of the point being tested, many tests can be performed concurrently, providing the opportunity for massive parallelism.

Fragment Shaders

The fragment[3] shader is the last programmable stage in OpenGL's graphics pipeline. This stage is responsible for determining the color of each fragment before it is sent to the framebuffer for possible composition into the window. After the rasterizer processes a primitive, it produces a list of fragments that need to be colored and passes this list to the fragment

2. Obviously, once they are rendered to the screen, points and lines have area; otherwise, we wouldn't be able to see them. However, this area is artificial and can't be calculated directly from their vertices.

3. The term *fragment* is used to describe an element that may ultimately contribute to the final color of a pixel. The pixel may not end up being the color produced by any particular invocation of the fragment shader due to a number of other effects such as depth or stencil tests, blending, and multi-sampling, all of which will be covered later in the book.

shader. Here, an explosion in the amount of work in the pipeline occurs, as each triangle could produce hundreds, thousands, or even millions of fragments.

Listing 2.4 in Chapter 2 contains the source code of our first fragment shader. It's an extremely simple shader that declares a single output and then assigns a fixed value to it. In a real-world application, the fragment shader would normally be substantially more complex and be responsible for performing calculations related to lighting, applying materials, and even determining the depth of the fragment. Available as input to the fragment shader are several built-in variables such as `gl_FragCoord`, which contains the position of the fragment within the window. It is possible to use these variables to produce a unique color for each fragment.

Listing 3.10 provides a shader that derives its output color from `gl_FragCoord`. Figure 3.4 shows the output of running our original single-triangle program with this shader installed.

```
#version 450 core

out vec4 color;

void main(void)
{
    color = vec4(sin(gl_FragCoord.x * 0.25) * 0.5 + 0.5,
                 cos(gl_FragCoord.y * 0.25) * 0.5 + 0.5,
                 sin(gl_FragCoord.x * 0.15) * cos(gl_FragCoord.y * 0.15),
                 1.0);
}
```

Listing 3.10: Deriving a fragment's color from its position

As you can see, the color of each pixel in Figure 3.4 is now a function of its position and a simple screen-aligned pattern has been produced. The shader of Listing 3.10 created the checkered patterns in the output.

The `gl_FragCoord` variable is one of the built-in variables available to the fragment shader. However, just as with other shader stages, we can define our own inputs to the fragment shader, which will be filled in based on the outputs of whichever stage is last before rasterization. For example, if we have a simple program with only a vertex shader and fragment shader in it, we can pass data from the fragment shader to the vertex shader.

The inputs to the fragment shader are somewhat unlike inputs to other shader stages, in that OpenGL *interpolates* their values across the primitive

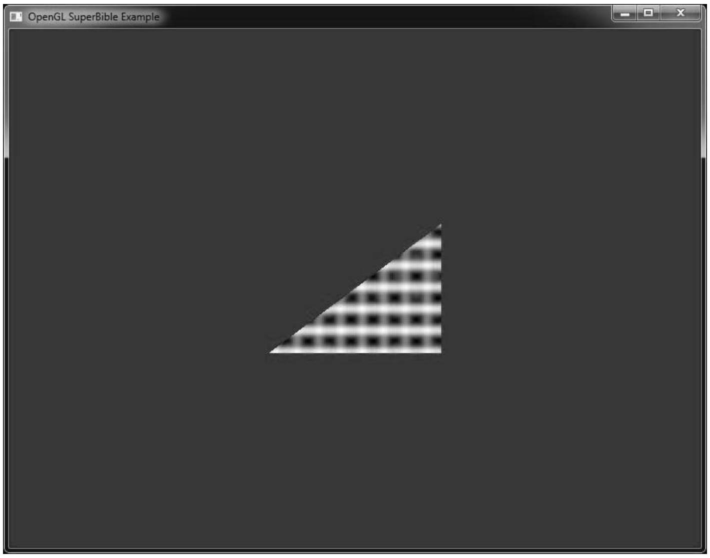


Figure 3.4: Result of Listing 3.10

that's being rendered. To demonstrate, we take the vertex shader of Listing 3.3 and modify it to assign a different, fixed color for each vertex, as shown in Listing 3.11.

```
#version 450 core

// 'vs_color' is an output that will be sent to the next shader stage
out vec4 vs_color;

void main(void)
{
    const vec4 vertices[3] = vec4[3](vec4( 0.25, -0.25, 0.5, 1.0),
                                     vec4(-0.25, -0.25, 0.5, 1.0),
                                     vec4( 0.25,  0.25, 0.5, 1.0));
    const vec4 colors[] = vec4[3](vec4( 1.0, 0.0, 0.0, 1.0),
                                  vec4( 0.0, 1.0, 0.0, 1.0),
                                  vec4( 0.0, 0.0, 1.0, 1.0));

    // Add 'offset' to our hard-coded vertex position
    gl_Position = vertices[gl_VertexID] + offset;

    // Output a fixed value for vs_color
    vs_color = color[gl_VertexID];
}
```

Listing 3.11: Vertex shader with an output

As you can see, in Listing 3.11 we added a second constant array that contains colors and index into it using `gl_VertexID`, writing its content to the `vs_color` output. In Listing 3.12 we modify our simple fragment shader to include the corresponding input and write its value to the output.

```
#version 450 core

// 'vs_color' is the color produced by the vertex shader
in vec4 vs_color;

out vec4 color;

void main(void)
{
    color = vs_color;
}
```

Listing 3.12: Deriving a fragment's color from its position

The result of using this new pair of shaders is shown in Figure 3.5. As you can see, the color changes smoothly across the triangle.

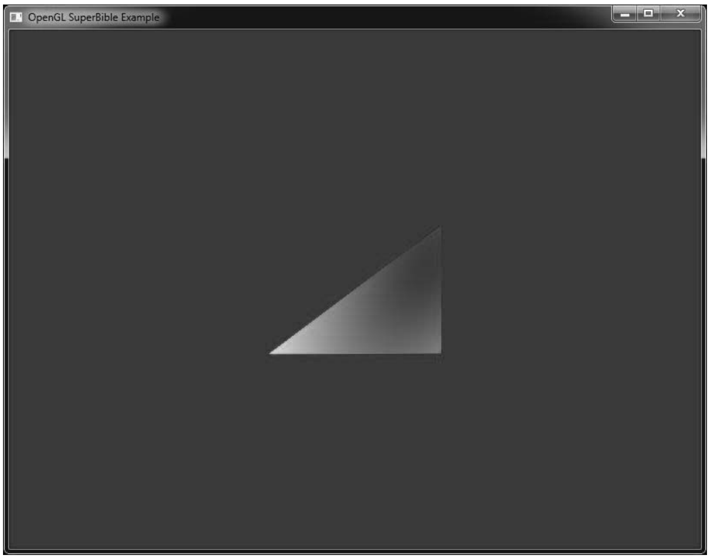


Figure 3.5: Result of Listing 3.12

Framebuffer Operations

The framebuffer is the last stage of the OpenGL graphics pipeline. It can represent the visible content of the screen and a number of additional regions of memory that are used to store per-pixel values other than color. On most platforms, this means the window you see on your desktop (or possibly the whole screen if your application covers it), which is owned by the operating system (or windowing system to be more precise). The framebuffer provided by the windowing system is known as the default framebuffer, but it is possible to provide your own if you wish to do things like render into off-screen areas. The state held by the framebuffer includes information such as where the data produced by your fragment shader should be written, what the format of that data should be, and so on. This state is stored in a *framebuffer object*. Also considered part of the framebuffer, but not stored per framebuffer object, is the pixel operation state.

Pixel Operations

After the fragment shader has produced an output, several things may happen to the fragment before it is written to the window, such as a determination of whether it even belongs in the window. Each of these things may be turned on or off by your application. The first thing that could happen is the *scissor test*, which tests your fragment against a rectangle that you can define. If it's inside the rectangle, then it will be processed further; if it's outside, it will be thrown away.

Next comes the *stencil test*. This compares a reference value provided by your application with the contents of the stencil buffer, which stores a single[4] value per pixel. The content of the stencil buffer has no particular semantic meaning and can be used for any purpose.

After the stencil test has been performed, the *depth test* is performed. The depth test is an operation that compares the fragment's z coordinate against the contents of the *depth buffer*. The depth buffer is a region of memory that, like the stencil buffer, is part of the framebuffer with enough space for a single value for each pixel; it contains the depth (which is related to distance from the viewer) of each pixel.

4. It's possible for a framebuffer to store multiple depth, stencil, or color values per pixel when a technique called *multi-sampling* is employed. We'll dig into this later in the book.

Normally, the values in the depth buffer range from 0 to 1, with 0 being the closest possible point in the depth buffer and 1 being the furthest possible point in the depth buffer. To determine whether a fragment is closer than other fragments that have already been rendered in the same place, OpenGL can compare the z component of the fragment's window-space coordinate against the value already in the depth buffer. If this value is less than what's already there, then the fragment is visible. The sense of this test can also be changed. For example, you can ask OpenGL to let fragments through that have a z coordinate that is greater than, equal to, or not equal to the content of the depth buffer. The result of the depth test also affects what OpenGL does to the stencil buffer.

Next, the fragment's color is sent to either the blending or logical operation stage, depending on whether the framebuffer is considered to store floating-point, normalized, or integer values. If the content of the framebuffer is either floating-point or normalized integer values, then blending is applied. Blending is a highly configurable stage in OpenGL and will be covered in detail in its own section.

In short, OpenGL is capable of using a wide range of functions that take components of the output of your fragment shader and of the current content of the framebuffer and calculate new values that are written back to the framebuffer. If the framebuffer contains unnormalized integer values, then logical operations such as logical AND, OR, and XOR can be applied to the output of your shader and the value currently in the framebuffer to produce a new value that will be written back into the framebuffer.

Compute Shaders

The first sections of this chapter describe the *graphics pipeline* in OpenGL. However, OpenGL also includes the *compute shader* stage, which can almost be thought of as a separate pipeline that runs indepdendently of the other graphics-oriented stages.

Compute shaders are a way of getting at the computational power possessed by the graphics processor in the system. Unlike the graphics-centric vertex, tessellation, geometry, and fragment shaders, compute shaders could be considered as a special, single-stage pipeline all on their own. Each compute shader operates on a single unit of work known as a *work item*; these items are, in turn, collected together into small groups called *local workgroups*. Collections of these workgroups can be sent into OpenGL's compute pipeline to be processed. The compute shader doesn't have any fixed inputs or outputs besides a handful of built-in variables to tell the shader which item it is working on. All processing performed by a compute shader is explicitly written to memory by the shader itself, rather than being consumed by a subsequent pipeline stage. A very basic compute shader is shown in Listing 3.13.

```
#version 450 core

layout (local_size_x = 32, local_size_y = 32) in;

void main(void)
{
    // Do nothing
}
```

Listing 3.13: Simple do-nothing compute shader

Compute shaders are otherwise just like any other shader stage in OpenGL. To compile one, you create a shader object with the type `GL_COMPUTE_SHADER`, attach your GLSL source code to it with **`glShaderSource()`**, compile it with **`glCompileShader()`**, and then link it into a program with **`glAttachShader()`** and **`glLinkProgram()`**. The result is a program object with a compiled compute shader in it that can be launched to do work for you.

The shader in Listing 3.13 tells OpenGL that the size of the local workgroup will be 32 by 32 work items, but then proceeds to do nothing. To create a compute shader that actually does something useful, you need to know a bit more about OpenGL—so we'll revisit this topic later in the book.

Using Extensions in OpenGL

All of the examples shown in this book so far have relied on the core functionality of OpenGL. However, one of OpenGL's greatest strengths is that it can be extended and enhanced by hardware manufacturers, operating system vendors, and even publishers of tools and debuggers. Extensions can have many different effects on OpenGL functionality.

An extension is any addition to a core version of OpenGL. Extensions are listed in the OpenGL extension registry[5] on the OpenGL Web site. These

5. Find the OpenGL extension registry at `http://www.opengl.org/registry/`.

extensions are written as a list of differences from a particular version of the OpenGL specification, and note what that version of OpenGL is. That means the text of the extensions describes how the core OpenGL specification must be changed if the extension is supported. However, popular and generally useful extensions are normally "promoted" into the core versions of OpenGL; thus, if you are running the latest and greatest version of OpenGL, there might not be that many extensions that are interesting but not part of the core profile. A complete list of the extensions that were promoted to each version of OpenGL and a brief synopsis of what they do is included in Appendix C, "OpenGL Features and Versions."

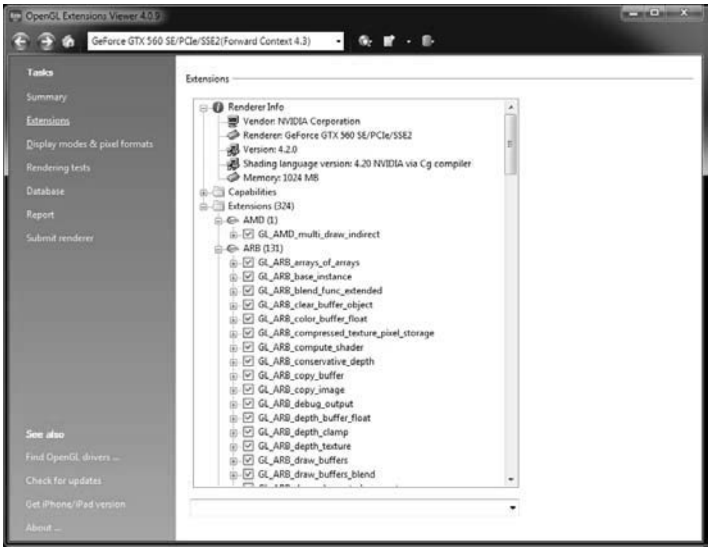
There are three major classifications of extensions: vendor, EXT, and ARB. Vendor extensions are written and implemented on one vendor's hardware. Initials representing the specific vendor are usually part of the extension name—"AMD" for Advanced Micro Devices or "NV" for NVIDIA, for example. It is possible that more than one vendor might support a specific vendor extension, especially if it becomes widely accepted. EXT extensions are written together by two or more vendors. They often start their lives as vendor-specific extensions, but if another vendor is interested in implementing the extension, perhaps with minor changes, it may collaborate with the original authors to produce an EXT version. ARB extensions are an official part of OpenGL because they are approved by the OpenGL governing body, the Architecture Review Board (ARB). These extensions are often supported by most or all major hardware vendors and may also have started out as vendor or EXT extensions.

This extension process may sound confusing at first. Hundreds of extensions currently are available! But new versions of OpenGL are often constructed from extensions programmers have found useful. In this way each extension gets its time in the sun. The ones that shine can be promoted to core; the ones that are less useful are not considered. This "natural selection" process helps to ensure only the most useful and important new features make it into a core version of OpenGL.

A useful tool to determine which extensions are supported in your computer's OpenGL implementation is Realtech VR's OpenGL Extensions Viewer. It is freely available from the Realtech VR Web site (see Figure 3.6).

Enhancing OpenGL with Extensions

Before using any extensions, you *must* make sure that they're supported by the OpenGL implementation that your application is running on. To find



```
Renderer Info
    Vendor: NVIDIA Corporation
    Renderer: GeForce GTX 560 SE/PCIe/SSE2
    Version: 4.2.0
    Shading language version: 4.20 NVIDIA via Cg compiler
    Memory: 1024 MB
Capabilities
Extensions (324)
    AMD (1)
        GL_AMD_multi_draw_indirect
    ARB (131)
        GL_ARB_arrays_of_arrays
        GL_ARB_base_instance
        GL_ARB_blend_func_extended
        GL_ARB_clear_buffer_object
        GL_ARB_color_buffer_float
        GL_ARB_compressed_texture_pixel_storage
        GL_ARB_compute_shader
        GL_ARB_conservative_depth
        GL_ARB_copy_buffer
        GL_ARB_copy_image
        GL_ARB_debug_output
        GL_ARB_depth_buffer_float
        GL_ARB_depth_clamp
        GL_ARB_depth_texture
        GL_ARB_draw_buffers
        GL_ARB_draw_buffers_blend
```

Figure 3.6: Realtech VR's OpenGL Extensions Viewer

out which extensions OpenGL supports, there are two functions that you can use. First, to determine the *number* of supported extensions, you can call **glGetIntegerv()** with the GL_NUM_EXTENSIONS parameter. Next, you can find out the name of each of the supported extensions by calling

```
const GLubyte* glGetStringi(GLenum name,
                            GLuint index);
```

You should pass GL_EXTENSIONS as the name parameter, and a value between 0 and 1 less than the number of supported extensions in index. The function returns the name of the extension as a string. To see if a specific extension is supported, you can simply query the number of extensions, and then loop through each supported extension and compare its name to the one you're looking for. The book's source code comes with a simple function that does this for you. **sb7IsExtensionSupported()** has the prototype

```
int sb7IsExtensionSupported(const char * extname);
```

This function is declared in the <sb7ext.h> header, takes the name of an extension, and returns non-zero if it is supported by the current OpenGL context and zero if it is not. Your application should always check for support for extensions you wish to use before using them.

Extensions generally add to OpenGL in some combination of four different ways:

- They can make things legal that weren't before, by simply removing restrictions from the OpenGL specification.
- They can add tokens or extend the range of values that can be passed as parameters to existing functions.
- They can extend GLSL to add functionality, built-in functions, variables, or data types.
- They can add entirely new functions to OpenGL itself.

In the first case, where things that once were considered errors no longer are, your application doesn't need to do anything besides start using the newly allowed behavior (once you have determined that the extension is supported, of course). Likewise, for the second case, you can just start using the new token values in the relevant functions, presuming that you have their values. The values of the tokens are in the extension specifications, so you can look them up there if they are not included in your system's header files.

To enable use of extensions in GLSL, you must first include a line at the beginning of shaders that use them to tell the compiler that you're going to need their features. For example, to enable the hypothetical `GL_ABC_foobar_feature` extension in GLSL, include the following in the beginning of your shader:

```
#extension GL_ABC_foobar_feature : enable
```

This tells the compiler that you intend to use the extension in your shader. If the compiler knows about the extension, it will let you compile the shader, even if the underlying hardware doesn't support the feature. If this is the case, the compiler should issue a warning if it sees that the extension is actually being used. Typically, extensions to GLSL will add preprocessor tokens to indicate their presence. For example, `GL_ABC_foobar_feature` will implicitly include

```
#define GL_ABC_foobar_feature 1
```

This means that you could write code such as

```
#if GL_ABC_foobar_feature
    // Use functions from the foobar extension
#else
```

```
    // Emulate or otherwise work around the missing functionality
#endif
```

This allows you to conditionally compile or execute functionality that is part of an extension that may or may not be supported by the underlying OpenGL implementation. If your shader absolutely requires support for an extension and will not work at all without it, you can instead include this more assertive code:

```
#extension GL_ABC_foobar_feature : require
```

If the OpenGL implementation does not support the `GL_ABC_foobar_feature` extension, then it will fail to compile the shader and report an error on the line including the `#extension` directive. In effect, GLSL extensions are opt-in features, and applications must[6] tell compilers up front which extensions they intend to use.

Next we come to extensions that introduce new functions to OpenGL. On most platforms, you don't have direct access to the OpenGL driver and extension functions don't just magically appear as available to your applications to call. Rather, you must ask the OpenGL driver for a *function pointer* that represents the function you want to call. Function pointers are generally declared in two parts; the first is the definition of the function pointer type, and the second is the function pointer variable itself. Consider this code as an example:

```
typedef void
(APIENTRYP PFNGLDRAWTRANSFORMFEEDBACKPROC) (GLenum mode,
                                            GLuint id);
PFNGLDRAWTRANSFORMFEEDBACKPROC glDrawTransformFeedback = NULL;
```

This declares the `PFNGLDRAWTRANSFORMFEEDBACKPROC` type as a pointer to a function taking `GLenum` and `GLuint` parameters. Next, it declares the `glDrawTransformFeedback` variable as an instance of this type. In fact, on many platforms, the declaration of the **`glDrawTransformFeedback()`** function is actually just like this. This seems pretty complicated, but fortunately the following header files include declarations of all of the

6. In practice, many implementations enable functionality included in some extensions by default and don't require that your shaders include these directives. However, if you rely on this behavior, your application is likely to not work on other OpenGL drivers. Because of this risk, you should always explicitly enable the extensions that you plan to use.

function prototypes, function pointer types, and token values introduced by all registered OpenGL extensions:

```
#include <glext.h>
#include <glxext.h>
#include <wglext.h>
```

These files can be found at the OpenGL extension registry Web site. The `glext.h` header contains both standard OpenGL extensions and many vendor-specific OpenGL extensions, the `wglext.h` header contains a number of extensions that are Windows specific, and the `glxext.h` header contains definitions that are X specific (X is the windowing system used on Linux and many other UNIX derivatives and implementations).

The method for querying the address of extension functions is actually platform specific. The book's application framework wraps up these intricacies into a handy function that is declared in the `<sb7ext.h>` header file. The function **`sb7GetProcAddress()`** has this prototype:

```
void * sb7GetProcAddress(const char * funcname);
```

Here, `funcname` is the name of the extension function that you wish to use. The return value is the address of the function, if it's supported, and `NULL` otherwise. Even if OpenGL returns a valid function pointer for a function that's part of the extension you want to use, that doesn't mean the extension is present. Sometimes the same function is part of more than one extension, and sometimes vendors ship drivers with partial implementations of extensions present. Always check for support for extensions using the official mechanisms or the **`sb7IsExtensionSupported()`** function.

Summary

In this chapter, you have taken a whirlwind trip down OpenGL's graphics pipeline. You have been (very) briefly introduced to each major stage and have created a program that uses each one of them, if only to do nothing impressive. We've glossed over or even neglected to mention several useful features of OpenGL with the intention of getting you from zero to rendering in as few pages as possible. You've also seen how OpenGL's pipeline and functionality can be enhanced by using extensions, which some of the examples later in the book will rely on. Over the next few chapters, you'll learn more fundamentals of computer graphics and of OpenGL, and then we'll take a second trip down the pipeline, dig deeper into the topics from this chapter, and get into some of the things we skipped in this preview of what OpenGL can do.

Index

{ } (curly braces), 211

2D
  array textures, 178–180
  graphics, 647–659
  prefix sums, 484
  projection, 69
  ray tracing, 636
3D graphics
  coordinate spaces used in, 70
  math for, 55. *See also* Math

Abstraction layers, 4
Acceleration
  graphics, 8
  structures, 646
Access
  map buffer types, 710–711
  memory, 692
  synchronizing
    atomic counters, 151–152
    to buffers, 692–694
    images, 194
    memory, 145–147
  texture arrays, 181–182
  vectors, 208
Adaptive tone mapping, 436
Adding. *See also* Inserting
  detail to images, 586
  directions, 627
  fog effects, 606–609
  planes, 645
Advanced occlusion queries, 543–545
AFR (alternate frame rendering) mode, 719
Algorithms
  flocking, 492–501
  prefix sum operations, 479–492
  shading, 599. *See also* Shadows
Aliasing artifacts, 169
Aligning uniform blocks, 125. *See also* Moving
Allocating
  data stores, 101
  memory, 100–107
  storage, 509, 691
Alpha-to-coverage approach, 420
Alternate frame rendering. *See* AFR mode
Alternative rendering methods, 613–647
  deferred shading, 613–624
  rendering without triangles, 631–647
  screen-space techniques, 624–631
ALU (arithmetic and logic unit) performance, 715
Ambient light, 568
Ambient occlusion, 624–631
AMD hardware, 704. *See also* Hardware
Angles, Euler, 78–79
Antialiasing, 412–428
  centroid sampling, 424–428
  filtering, 413–415
  multi-sample, 156, 415–417
  sample rate shading, 421–423
  textures, 417–421
AoSs (array-of-structures), 115
APIs (application programming interfaces), 3, 6
Appending packets to packet buffers, 670
Application programming interfaces. *See* APIs
Applications
  barriers, inserting, 146–147
  building, 717
  debugging, 717, 729–737
  geometry shaders in, 336–341
  performance, 661. *See also* Performance

Applications (*continued*)
  rendering. *See* Rendering
  simple example of, 14–16
  `sparsetexture`, 512–514
  speed, tuning for, 706–726
  textures, unbinding, 504
Applying
  attributes in vertex shaders, 113
  barriers in shaders, 147
  centroid qualifiers, 424
  compression, 197–198
  compute shaders, 468–469
  extensions, 49–54
  features, 713
  fog, 609
  instanced arrays, 265–271
  mipmaps, 174
  multiple GPUs, 719–721
  multiple vertex shader inputs, 113–116
  multi-threading, 721–723
  query results, 537–540
  roughness, 597
  shaders, 17–24
  transform feedback, 280–298
Approaching Zero Driver Overhead. *See* AZDO
ARB (Architectural Review Board), 7, 8, 50
Architectural Review Board. *See* ARB
Architecture, 5
Arithmetic and logic unit performance. *See* ALU performance
Array-of-structures. *See* AoSs
Arrays, 115, 210–212
  declaring, 211
  floats, 120
  inputs, 339
  instancing, 265–271
  `props`, 229
  scissor testing, configuring, 372
  textures, 177–182, 397, 400, 411
  VAOs (vertex array objects), 295, 493
Artifacts, aliasing, 169
ASCII characters, 652
Asteroids, drawing, 274–279
Atlas textures, 509
Atmospheric effects, 605–609
Atomic counters, 147–152, 151–152
Atomic operations, 140
  images, 188–194
  memory, 142–144
Attachments
  completeness, 404
  framebuffers
    integers, 444–446
    multiple, 395–397
    rendering with no, 428–430
  rendering, 392
  textures, 392
Attributes
  indirect draws, 276
  interleaved, 114, 116
  separate, 114
  vertex shaders, 113
  vertices, 28–29, 110
    configuring, 112
    disabling, 113
    types, 246
AZDO (Approaching Zero Driver Overhead), 677

Back buffers, 390
Back end, 12
Back-facing, 41, 342
Back-to-back waiting, 536
Backward compatibility, 8, 9
Bandwidth, memory, 195, 614
`barrier()` function, 476, 478
Barriers, 476
  applications, inserting, 146–147
  shaders, 147
Barycentric coordinates, 36, 306–307, 311
Base vertex, 254–255
Bézier curves, 91. *See also* Curves
  cubic, 92, 93
  quadratic, 92
  quintic, 94
Bézier splines, 95
BGRA ordering, 247
Big-picture views, 11
Binaries, programs, 235–238, 719
Binding, 100
  buffers, 110, 133
  eliminating, 504–509
  image units, 184
  points
    assigning, 131
    transform feedback, 284
`bindingindex` parameter, 110
Bindless textures, 504, 682
Bitangent vectors, 582
Bitmaps, fonts, 655–659
Blend equations, 383, 386–387
Blending, 382–387, 413
Blinn-Phong lighting models, 577–579
Blit, 461
Block Partitioned Texture Compression. *See* BPTC
Blocks, 516
  atomic operations, 143–144
  interfaces, 31–33
  packing RGTC, 522
  palettizing RGTC, 522
  shader storage, 140–147
  texels, fetching, 520
  uniforms, 121–135. *See also* Uniforms
    declaring, 121, 680

        default, 117–120
        indexes, 126
Bloom programs, 438–444
Boolean occlusion queries, 543
`bool` scalar type, 206
Bounces, increasing, 643
Bound buffers, copying, 109
Boxes
    bounding, 537
    filters, 486
    ray tracing, 646
BPTC (Block Partitioned Texture Compression), 196
Brute force approaches, 646
Bubbles, 557
Buffers, 10, 100–117
    access to, synchronizing, 692–694
    atomic counters, 149
    back, 390
    bindings, 110, 133
    commands, 668, 699
    copying, 107–109
    depth, 626
    feedback, 549
    filling, 107–109
    formatting, 103
    G-buffers, 614
    indirect draws, configuring, 275
    initializing, 103
    mapping, 100, 102–103, 104, 699, 709–713
    memory, allocating, 100–107
    objects, updating, 104
    packets, 668–677, 676–677
    persistent maps, 665, 691–692
    ranges, mapping, 106
    stencil, 373–374
    storage flags, 102
    TBOs (texture buffer objects), 288
    UBOs (uniform buffer objects), 121
    unmapping, 105
    updating, 378–379
    velocity, 493
    vertex shaders, feeding from, 109–117
    write combined memory, 699
Building applications, 717
Built-in data manipulation functions, 218–219
Built-in functions, 213–219
Built-in math functions, 215–217
Built-in matrices, 213–215
Built-in variables, 25
Bump mapping, 582
Bumpy surfaces, 625

C, corresponding OpenGL type tokens, 108
Calculating
    ambient light, 568
    bitangent vectors, 582
    Blinn-Phong lighting models, 577
    indexes, 270
    light, 619
    lighting, 640
    rim lighting, 580
    shadows, 600
    TBN (tangent, bitangent, normal) matrix, 584
Callback functions, 731
Cameras, 490. *See also* Images
Candidate draws, culling, 685
Cartesian coordinates, 40, 583
Casting shadows, 599–605
`ccw` layout qualifiers, 319
Cell shading, 610–613
Central processing units. *See* CPUs
Centroid sampling, 424–428
Chaining packet buffers, 675
Characters, ASCII, 652
Clamping
    depth, enabling, 379–380
    levels of detail, 515
    tone mapping, 435
Classes, image data formats, 185
Classifications of extensions, 50
Clipping, 39–40, 298–305
    depth clamping, 380
    guard bands, 301–302
    objects, 304
    user-defined, 302–305
Clip spaces, 18, 39, 72
Clocks, 5
Coefficients, exposure, 435
Colors
    distance fields, 654–655
    fragments, 44, 46, 48
    Julia sets, 634
    masking, 388–390
    one-dimensional color lookup tables, 610
    outputs, 382–387
    specular highlights, 569
    sRGB color spaces, 446–448
    vertex shaders, 266
Column-major layouts, 66
Column-primary layouts, 66
Columns, 485
Commands, 4. *See also* Functions
    buffers, 668, 699
    drawing, 251–280
        applying instanced arrays, 265–271
        indexed, 251–257
        indirect draws, 271–280
        instancing, 257–265
    formatting, 662
    `SwapBuffers()`, 702
Comment chunks, 755

Committing
  pages, 510
  textures, 513
Common vector operators, 60–64
  cross products, 62–63
  dot products, 60–61
  length of vectors, 63
  reflection, 63–64
  refraction, 63–64
Communication
  compute shaders, 474–479
  between tessellation shader invocations, 322
Comparisons, depth testing, 377
Compatibility
  APIs (application programming interfaces), 6
  backward, 8, 9
  profiles, 9
  texture target view, 200
Compilers, 219–224
  retrieving logs, 221
Compiling shaders, performance, 716–719
Completeness, framebuffers, 403–407
Components, G-buffers, 618
Compression, textures, 195–199, 516–525, 715
Computer programs, 468
Compute shaders, 48–49, 467, 704
  applying, 468–469
  communication, 474–479
  culling, 685
  examples, 479–501
  executing, 469–474
  flocking, 492–501
  objects, culling, 686
  parallel prefix sums, 479–492
  synchronizing, 475–479
Concatenating transformations, 80–81
Conditional rendering, 541–543
Conditions
  inverting, 544
  race, 476, 477, 478, 692
Configuring
  atomic counters, 149
  cube geometry, 136, 253
  custom culling geometry shaders, 342
  framebuffers, 390, 398
  indirect draws, 275
  Julia sets, 633
  pipelines (graphics), 226
  scissor testing, 372
  shadow matrices, 603
  tessellation geometry shaders, 347
  uniforms, 119–120
  values, subroutine uniforms, 235
  vertices, attributes, 112
Connecting vertices, 289
Consuming G-buffers, 617–619
Control flow, queries, 545
Control points, 33, 91, 94, 306, 319, 329
Control shaders (tessellation), 33–35
Conventions, coordinates, 41
Converting .DDS files, 745–746
Coordinates
  barycentric, 36, 306–307, 311
  Cartesian, 40, 583
  clip spaces, 72
  conventions, 41
  floating-point textures, 167
  interpolation, 368
  normalized device space, 72
  objects, 71
  spaces, 70–72
  textures, 162–164, 209
  transformations, 65, 72–80
  views, 71–72
  windows, 41
  world, 71
  world-space, 615
Copying
  arrays, textures, 411
  buffers, 107–109
  data, 696
  between framebuffers, 461–464
  zero copy, 691–699
Cores
  profile OpenGL, 8–10
  shaders, 5
Counters
  atomic, 147–152
  performance, 706
C++ prefix sum implementation, 480
CPUs (central processing units), 102
  multi-threading, 662–667
  optimization, 661–677
  packet buffers, 668–677
  performance analysis tools, 699–726
  queues, 700
  zero copy, 691–699
Cross products, 62–63
Cube geometry, 136
  configuring, 253
  drawing, 254
Cubemaps, 592–597
  rendering to, 402–403
Cubic Bézier curves, 92, 93
Cubic Bézier patches, tessellation, 329–333
Cubic Bézier splines, 95
Cubic Hermite splines, 97
Culling, 41–43
  back-face, 342
  computer shaders, 685
  objects, 686
Curly braces ({ }), 211
Curves, 89
  Hermite, 217

math (for 3D graphics), 91–94
transfer, 436
Custom culling geometry shaders, 342
Customizing resolve operations, 419
`cw` layout qualifiers, 319

Data, 99
atomic counters, 147–152
buffers, 100–117
allocating memory, 100–107
copying/filling, 107–109
feeding vertex shaders from, 109–117
chunks, 751–752
copying, 696
fixed-point, 246
G-buffers, unpacking from, 617
management, 503
eliminating binding, 504–509
generating compressed data, 519–525
high-quality texture filtering, 527–531
packed data formats, 525–527
RGTC (Red–Green Texture Compression) compression schemes, 516–519
sparsely populated textures, 509–515
texture compression, 516–525
packed data formats, 247
partitioning, 692
per patch, 306
reading, 707–709
shader storage blocks, 140–147
tessellation, passing between shaders, 319–322
textures, 152–202
copying into, 463–464
reading back, 464–466
threading, generating, 662–667
under-sampling, 412
uniforms, 117–140
Data manipulation functions, built-in, 218–219
Data stores, 100, 664
allocating, 101
mapping, 105
Data types, 206–212
scalars, 206–207
support, 119
`dds2ktx` program, 745–746
.DDS files, converting, 745–746
Debug contexts, 730–737
Debuggers, 705, 717
Debugging applications, 729–737
Decay, 608
`DECLARE_MAIN` macro, 14
Declaring
arrays, 211
atomic counters, 148
interface variables, 31
multiple inputs in vertex shaders, 113
multiple outputs in fragment shaders, 396
shader storage blocks, 140, 141
uniforms, 121
blocks, 123, 680
subroutines, 232
vertices, 28, 247
Decommitting pages, 510
Decompressors, 519. *See also* Compression
Decrementing atomic counters, 149
Default blocks, uniforms, 117–120
Deferred procedure calls. *See* DPCs
Deferred shading, 613–624
downsides to, 622–624
normal mapping, 619–622
Degenerate primitives, 24
Dependency (blocks), 516
Depth
buffers, 47, 48, 378–379, 626
clamping, enabling, 379–380
of field, 488, 490
in lighting models, 602
testing, 376–380
Depth testing, 47
Design, 4
Desktop Window Manager. *See* DWM
Destination factors, 383
Detail to images, adding, 586
Detecting edges, 426–428
Diffuse albedo, 576
Diffuse light, 569
Directions, adding, 627
Disabling
interpolation, 366–367
rasterizers, 288
scissor testing, 371
vertex attributes, 113
Disassembling packets, 672
Discarding
geometry in geometry shaders, 341–344
rasterizers, 296
Displacement mapping, 323
tessellation evaluation shaders, 607
Distance
fields, textures, 647–655
focal, 488
Division, perspective, 40
Domains, patches, 329
Do-nothing computer shaders, 468
Dot products, 60–61, 584
`double` scalar type, 206
Downsides to deferred shading, 622–624
DPCs (deferred procedure calls), 702
Draining pipelines, 557
Drawing
asteroids, 274–279
commands, 251–280
applying instanced arrays, 265–271

Drawing (*continued*)
indexed, 251–257
indirect draws, 271–280
instancing, 257–265
cube geometry, 254
data written to transform feedback buffers, 553
geometry, 258
grass, 261–262
points, 250
stereo windows, 409
triangles, 24–26
Dual encodings (RGTC), 519
Dual-source blending, 385–386
DWM (Desktop Window Manager), 701

EAC (Ericsson Alpha Compression), 197
Early testing, fragments, 380–382
Edges, detecting, 426–428
Editions, 7
Effects, atmospheric, 605–609
Elements
iterating, 325
types, 211
Eliminating binding, 504–509
`EmitVertex()` function, 37, 38, 335, 340
`EnableDisable` function, 674
Enabling depth clamping, 379–380
Encoding ASCII characters, 652
Ending pipelines with transform feedback, 288
Endpoints, 517
`EndPrimitive()` function, 37, 38, 335, 341
Engines, tessellation, 35. *See also* Tessellation
Environment maps, 587–597
filtering, 598
`equal_spacing` mode, 317
Equations. *See also* Math (for 3D graphics)
blend, 383, 386–387
quadratic, 92
Equirectangular environment maps, 590–591
Ericsson Alpha Compression. *See* EAC
Ericsson Texture Compression. *See* ETC2
ETC2 (Ericsson Texture Compression), 197
Euler angles, 78–79
Evaluation shaders (tessellation), 35–37, 607
Evolution of OpenGL, 6–10
Examples of compute shaders, 479–501
Exclusive prefix sums, 480
Executing
compute shaders, 469–474
hardware, 4
packet buffers, 671, 677
Exploding models, 344, 345
Exponents, sharing, 198–199
Exposure coefficients, 435
Extensions, 8, 49–54
Extinction, 606

Faces
finding normal, 342
pushing, 345
Factors, shininess, 570
Features, 713, 759–796
Feedback
buffers, 549
transform, 280–298
Fences, synchronization, 557–562
Fetching
blocks, texels, 520
vertices, 28
Fields
distance, textures, 647–655
stacking, 209
stars, rendering, 450–453
Files
.DDS, converting, 745–746
.KTX, 160–161, 743–745
objects, loading, 116–117
.SBM, 746–748, 749–757
chunk headers, 750–751
comment chunks, 755
defined chunks, 751–755
file headers, 749–750
object list chunks, 755–757
textures, loading from, 160–164
Filling buffers, 107–109
Filtering, 412
antialiasing, 413–415
box filters, 486
environment maps, 598
high-quality texture, 527–531
mipmaps, 171–173
textures, 167–169
variables, 487
Filtering modes, samplers, 164
Fixed-function stages, 5
Fixed-point data, 246
Flags
buffers
mapping, 106
storage, 102
`GL_DYNAMIC_STORAGE_BIT`, 102
`flags` parameter, 101
Flat inputs, 366
`flat` qualifier, 367
Flexible indirect rendering, 678–683
Floating-point framebuffers, 430–444
Floating-point numbers, 525
Floating-point textures, 167
Floats, arrays, 120
`float` scalar type, 206
Flocking compute shaders, 492–501
Flow control barriers, 476
Focal depth, 488
Focal distance, 488

Focus, images, 488
Fog, 606–609. *See also* Atmospheric effects
Fonts
  bitmaps, 655–659
  distance fields, 652–654
Format layout qualifiers, 186
Formatting
  applications, simple example of, 14–16
  buffers, 103
  commands, 662
  compression, 195
  framebuffers, 390, 428–448
  internal formats, 153
  .KTX (Khronos TeXture), 160–161
  multi-sample textures, 417
  packed data formats, 247, 525–527
  packet buffers, 668–672
  RGTC (Red–Green Texture Compression), 516–519
  screenshots, 460–461
  stereo windows, 408
  textures, 152–154, 199–202
Fractals
  Julia sets, 696, 697
  rendering, 698
`fractional_even_spacing` mode, 317
`fractional_odd_spacing` mode, 317
Fractional segments, 317
Fragments
  colors, 48
  depth testing, 376–380
  lighting, 619
  Phong shading, 574
  quad, 556
Fragment shaders, 17, 43–46, 366–369
  ambient occlusion, 631
  array textures, 400
  colors
    masking, 388–390
    output, 382–387
  cubemaps, 596
  deferred shading, 621
  distance field rendering, 651
  equirectangular environment maps, 590–591
  fog, 609
  Gouraud shading, 572
  high-frequency output, 422
  images loads/stores, 188
  interpolation, 366–369
  logical operations, 387–388
  normal mapping, 586
  off-screen rendering, 390–412
  per-fragment shininess, 598–599
  Phong shaders, 575–576
  pre-fragment tests, 369–382
  ray tracing in, 634–647
  shadow mapping, 604
  skybox rendering, 595
  spherical environment mapping, 589
  spinning cubes, 138
  terrain rendering, 327
  texture coordinates, 163
  toon, 612
Framebuffers, 704
  antialiasing. *See* Antialiasing
  attachments
    multiple, 395–397
    rendering with no, 428–430
  completeness, 403–407
  configuring, 390
  copying between, 461–464
  floating-point, 430–444
  formatting, 390, 428–448
  G-buffers. *See* G-buffers
  integers, 444–446
  layers, 397
  operations, 47–48
  reading from, 458–461
  sRGB color spaces, 446–448
  stacks, implementing, 642
Frame Profiler, 706
Frames, AFR (alternate frame rendering) mode, 719
Free-form Bézier patches, 329
Front end, 12
Front-facing, 41
Frustrum matrices, 88
Functionality of extensions, 53
Functions, 759–796. *See also* Methods
  `barrier()`, 476, 478
  blend, 383–385
  built-in, 213–219
  built-in data manipulation, 218–219
  built-in math, 215–217
  callback, 731
  `EmitVertex()`, 37, 38, 335, 340
  `EnableDisable`, 674
  `EndPrimitive()`, 37, 38, 335, 341
  `glAttachShader()`, 20, 468
  `glBeginConditionalRender()`, 541541
  `glBeginQuery()`, 535, 545, 551
  `glBeginQueryIndexed()`, 551
  `glBeginTransformFeedback()`, 287
  `glBindBuffer()`, 101, 284, 710
  `glBindBufferRange()`, 284
  `glBindFramebuffer()`, 391, 393
  `glBindSampler()`, 165
  `glBindTexture()`, 152
  `glBindTransformFeedback()`, 553, 713
  `glBindVertexArray()`, 713
  `glBindVertexArrays()`, 21
  `glBindVertexBuffer()`, 249
  `glBlendColor()`, 383
  `glBlendEquation()`, 386

Functions (*continued*)
glBlendEquationSeparate(), 386
glBlendFunc(), 383
glBlendFuncSeparate(), 383
glBlitFramebuffer(), 463
glBufferData(), 122, 141, 709
glBufferStorage(), 101, 107, 696
glBufferSubData(), 102, 128, 666
glCheckFramebufferStatus(), 406
glClear(), 372
glClearBufferfv(), 372
glClearBufferiv(), 374
glClearBufferSubData(), 687
glClearBufferv(), 541
glClearTexSubImage(), 154
glClientWaitSync(), 559, 560, 692
glColorMask(), 389, 537
glColorMaski(), 389
glCompilerShader(), 220
glCompileShader(), 19, 468, 717
glCopyBufferSubData(), 108, 109
glCopyImageSubData(), 464
glCopyNamedBufferSubData(), 108, 696
glCreateBuffers(), 100
glCreateFramebuffers(), 390
glCreateProgram(), 20
glCreateShader(), 19, 336, 468
glCreateTextures(), 152
glCreateVertexArrays(), 20
glDebugMessageCallback(), 731, 734
glDebugMessageControl(), 734
glDeleteQueries(), 535
glDeleteShader(), 20
glDeleteSync(), 562
glDeleteTextures(), 161
glDepthFunc(), 377
glDepthMask(), 378
glDepthRange(), 41, 361
glDisable(), 378, 674
glDispatchCompute(), 469, 474,541
glDispatchComputeIndirect(), 469
glDrawArrays(), 22, 24, 25, 113, 137, 251,
257, 344, 541, 666
glDrawArraysIndirect(), 273, 469
glDrawArraysInstanced(), 260, 265
glDrawArraysInstancedBaseInstance(), 469
glDrawArraysInstancedBaseInstanced(), 270
glDrawArraysInstancedBaseVertex(), 678
glDrawBuffer(), 393, 397, 538
glDrawBuffers(), 397
glDrawElements(), 251, 272, 344, 545
glDrawElementsBaseVertex(), 678
glDrawElementsIndirect(), 272, 273
glDrawElementsInstanced(), 260, 265
glDrawTransformFeedback(), 713
glEnable(), 674
glEndConditionalRender(), 541541
glEndQuery(), 536, 545, 551
glEndQueryIndexed(), 551
glEndTransformFeedback(), 287
glFenceSync(), 558, 692
glFinish(), 557, 692
glFlush(), 557
glFlushMappedBufferRange(), 666
glFramebufferParameteri(), 429
glFramebufferTexture(), 395, 403, 418, 419
glFramebufferTextureLayer(), 401
glFrontFace(), 42
glGenerateMipMap(), 466
glGenFramebuffers(), 391
glGenQueries(), 534
glGenTransformFeedbacks(), 553
glGetCompressedTexImage(), 741
glGetError(), 406, 534, 733
glGetGraphicsResetStatus(), 739
glGetIntegeri(), 470
glGetIntegerv(), 135, 340
glGetInternalformativ(), 511
glGetProgramBinary, 236
glGetProgramiv(), 471
glGetQueryObjectiv(), 536, 538, 545
glGetShaderiv, 221
glGetShaderiv(), 717
glGetTexImage(), 465, 740
glGetTextureHandleARB(), 504
glGetTextureSamplerHandleARB(), 504
glGetTextureStorage2D(), 509
glGetUniformBlockIndex(), 131
glGetUniformLocation(), 118–119
glLinkProgram(), 20, 224, 468, 717
glLogicOp(), 388
glMakeCurrent(), 721
glMakeTextureHandleResidentARB(), 507
glMapBuffer(), 105, 122, 709
glMapBufferRange(), 128, 141, 666, 710
glMapNamedBufferRange(), 691, 692
glMapNamedBufferRanger(), 106
glMapNamedBufferStorage(), 691, 692
glMemoryBarrier(), 146, 147, 151
glMinSampleShading(), 423
glMultiDrawArraysIndirect(), 678
glMultiDrawArraysIndirectCountARB(), 684
glMultiDrawElementsIndirect(), 678
glMultiDrawElementsIndirectCountARB(),
684
glNamedBufferStorage(), 101, 103
glNamedFramebufferDrawBuffer(), 393
glNamedFramebufferTexture(), 395
glObjectLabel(), 736
glObjectPtrLabel(), 736
glPatchParameteri(), 33
glPointParameteri(), 453
glPointSize(), 23, 38, 250
glPolygonMode(), 319

glPopDebugGroup(), 733
glPrimitiveRestartIndex(), 257
glPushDebugGroup(), 733
glQueryCounter(), 547, 548
glReadBuffer(), 459
glReadPixels(), 458, 465, 683, 707, 708, 709, 740
glSampleCoverage(), 420
glScissor(), 370
glScissorIndexed(), 370
glScissorIndexedv(), 370
glShaderSource(), 19, 468
glStencilFuncSeparate(), 375, 376
glStencilMaskSeparate(), 376
glStencilOpSeparate(), 374, 375
glTexPageCommitmentARB(), 510, 512
glTexparameteri(), 601
glTexStorage2D(), 152, 592
glTexStorageSubImage2D(), 658
glTexSubImage2D(), 466, 524, 592, 697
glTextBuffer(), 295
glTexturePageCommitmentEXT(), 510, 512
glTextureParameteri(), 510
glTextureStorage2DMultisample(), 417
glTextureStorage3DMultisample(), 417
glTextureStorageSubImage2D(), 512
glTextureSubImage2D(), 170
glTextureView(), 199
glTransformFeedbackVaryings(), 281
glUniform(), 469
glUniform*(), 119
glUniformMatrix*(), 120
glUseProgram(), 22, 134, 469
glVertexArrayAttribFormat(), 111
glVertexArrayVertexBuffer(), 112
glVertexAttrib*(), 29, 31
glVertexAttribDivisor(), 500
glVertexAttribFormat(), 525
glVertexAttribPointer(), 244–246, 714, 715
glVertexAtttribDivisor(), 268
glVertextAttribPointer(), 266
glViewPort(), 41, 137, 361
glViewportArrayv, 370
glWaitSync(), 559, 560
glXMakeCurrent(), 721
init(), 731
main(), 17, 335, 342, 524
make_face, 348
matrixCompMult(), 214
max, 569
MemoryBarrier(), 147, 687
NextPacket, 671
overloading, 183, 213
packRGTC, 523
palettizeTexel, 523
startup(), 663
stencil, 373–374
texelFetch(), 167
texture(), 167
textureGatherOffset, 520
vectors, 213–215

Gamma correction, 446
G-buffers, 614
  components, 618
  consuming, 617–619
  generating, 614–617
  initializing, 616
  writing to, 616
Generating
  compressed data, 519–525
  data, applying threads to, 662–667
  distance fields, 648
  G-buffers, 614–617
  geometry in geometry shaders, 346–349
  query objects, 549
  vertices, 347
Geometry
  cubes
    configuring, 253
    drawing, 254
  displacement mapping, 323
  drawing, 258
  Phong shading, 573
  primitive restarts, 255–257
  rendering, 267
  uniforms, transforming, 135–140
Geometry shaders, 37–39, 333–334, 704
  geometry
    discarding in, 341–344
    generating, 346–349
    modifying in, 344–345
  layers, rendering with, 399, 411
  multiple streams of storage, 352–353
  multiple viewport transformations, 361–364
  new primitive types, 353–361
  pass-through, 334–336
  primitive types, modifying, 349–352
  quads, rendering, 355–361
  tessellation, configuring, 347
  triangles, 348
Gimbal locks, 79
GL_DYNAMIC_STORAGE_BIT flag, 102
GL (graphics library) functions, 6
  glAttachShader() function, 20, 468
  glBeginConditionalRender() function, 541541
  glBeginQuery() function, 535, 545, 551
  glBeginQueryIndexed() function, 551
  glBeginTransformFeedback() function, 287
  glBindBuffer() function, 101, 284, 710
  glBindBufferRange() function, 284
  glBindFramebuffer() function, 383, 391
  glBindSampler() function, 165

GL (graphics library) functions (*continued*)
glBindTexture() function, 152
glBindTransformFeedback() function, 553, 713
glBindVertexArray() function, 713
glBindVertexArrays() function, 21
glBindVertexBuffer() function, 249
glBlendColor() function, 383
glBlendEquation() function, 386
glBlendEquationSeparate() function, 386
glBlendFunc() function, 383
glBlendFuncSeparate() function, 383
glBlitFramebuffer() function, 463
glBufferData() function, 122, 141, 709
glBufferStorage() function, 101, 107, 696
glBufferSubData() function, 102, 128, 666
glCheckFramebufferStatus() function, 406
glClearBufferfv() function, 15, 372
glClearBufferiv() function, 374
glClearBufferSubData() function, 687
glClearBufferv() function, 541
glClear() function, 372
glClearTexSubImage() function, 154
glClientWaitSync() function, 559, 560, 692
glColorMask() function, 389, 537
glColorMaski() function, 389
glCompilerShader() function, 220
glCompileShader() function, 19, 468, 717
glCopyBufferSubData() function, 108, 109
glCopyImageSubData() function, 464
glCopyNamedBufferSubData() function, 108, 696
glCreateBuffers() function, 100
glCreateFramebuffers() function, 390
glCreateProgram(), 20
glCreateShader() function, 19, 336, 468
glCreateTextures() function, 152
glCreateVertexArrays() function, 20
glDebugMessageCallback() function, 731, 734
glDebugMessageControl() function, 734
glDeleteQueries() function, 535
glDeleteShader() function, 20
glDeleteSync() function, 562
glDeleteTextures() function, 161
glDepthFunc() function, 377
glDepthMask() function, 378
glDepthRange() function, 41, 361
glDisable() function, 378, 674
glDispatchCompute() function, 469, 474, 541
glDispatchComputeIndirect() function, 469
glDrawArrays() function, 22, 24, 25, 113, 137, 251, 257, 344, 541, 666
glDrawArraysIndirect() function, 273, 469
glDrawArraysInstancedBaseInstanced() function, 270
glDrawArraysInstancedBaseInstance() function, 469
glDrawArraysInstancedBaseVertex() function, 678
glDrawArraysInstanced() function, 260, 265
glDrawBuffer() function, 393, 397, 538
glDrawBuffers() function, 397
glDrawElementsBaseVertex() function, 678
glDrawElements() function, 251, 272, 344, 545
glDrawElementsIndirect() function, 272, 273
glDrawElementsInstanced() function, 260, 265
glDrawTransformFeedback() function, 713
glEnable() function, 674
glEndConditionalRender() function, 541541
glEndQuery() function, 536, 545, 551
glEndQueryIndexed() function, 551
glEndTransformFeedback() function, 287
glFenceSync() function, 558, 692
glFinish() function, 557, 692
glFlush() function, 557
glFlushMappedBufferRange() function, 666
glFramebufferParameteri() function, 429
glFramebufferTexture() function, 395, 403, 418, 419
glFramebufferTextureLayer() function, 401
glFrontFace() function, 42
glGenerateMipMap() function, 466
glGenFramebuffers() function, 391
glGenQueries() function, 534
glGenTransformFeedbacks() function, 553
glGetCompressedTexImage() function, 741
glGetError() function, 406, 534, 733
glGetGraphicsResetStatus() function, 739
glGetIntegeri() function, 470
glGetIntegerv() function, 135, 340
glGetInternalformativ() function, 511
glGetProgramBinary function, 236
glGetProgramiv() function, 471
glGetQueryObjectiv() function, 536, 538, 545
glGetShaderiv() function, 221, 717
glGetTexImage() function, 465, 740
glGetTextureHandleARB() function, 504
glGetTextureSamplerHandleARB() function, 504
glGetTextureStorage2D() function, 509
glGetUniformBlockIndex() function, 131
glGetUniformLocation() function, 118–119
glLinkProgram() function, 20, 224, 468, 717
glLogicOp() function, 388
glMakeCurrent() function, 721
glMakeTextureHandleResidentARB() function, 507
glMapBuffer() function, 105, 122, 709

glMapBufferRange() function, 128, 141, 666, 710
glMapNamedBufferRange() function, 106, 691, 692
glMapNamedBufferStorage() function, 691, 692
glMemoryBarrier() function, 146, 147, 151
glMinSampleShading() function, 423
glMultiDrawArraysIndirectCountARB() function, 684
glMultiDrawArraysIndirect() function, 678
glMultiDrawElementsIndirectCountARB() function, 684
glMultiDrawElementsIndirect() function, 678
glNamedBufferStorage() function, 101, 103
glNamedFramebufferDrawBuffer() function, 393
glNamedFramebufferTexture() function, 395
glObjectLabel() function, 736
glObjectPtrLabel() function, 736
glPatchParameteri() function, 33
gl_PerVertex structure, 320
glPointParameteri() function, 453
glPointSize() function, 20, 38, 250
glPolygonMode() function, 319
glPopDebugGroup() function, 733
glPrimitiveRestartIndex() function, 257
glPushDebugGroup() function, 733
glQueryCounter() function, 547, 548
glReadBuffer() function, 459
glReadPixels() function, 458, 465, 683, 707, 708, 709, 740
glSampleCoverage() function, 420
glScissor() function, 370
glScissorIndexed() function, 370
glScissorIndexedv() function, 370
glShaderSource() function, 19, 468
glStencilFuncSeparate() function, 375, 376
glStencilMaskSeparate() function, 376
glStencilOpSeparate() function, 374, 375
glTexPageCommitmentARB() function, 510, 512
glTexparameteri() function, 601
glTexStorage2D() function, 152, 592
glTexStorageSubImage2D() function, 658
glTexSubImage2D() function, 466, 524, 592, 697
glTextBuffer() function, 295
glTexturePageCommitmentEXT() function, 510, 512
glTextureParameteri() function, 510
glTextureStorage2DMultisample() function, 417
glTextureStorage3DMultisample() function, 417
glTextureStorageSubImage2D() function, 512
glTextureSubImage2D() function, 170
glTextureView() function, 199
glTransformFeedbackVaryings() function, 281
glUniform() function, 469
glUniform*() function, 119
glUniformMatrix*() function, 120
glUseProgram() function, 22, 134, 469
glVertexArrayAttribFormat() function, 111
glVertexArrayVertexBuffer() function, 112
glVertexAttribDivisor() function, 500
glVertexAttribFormat() function, 525
glVertexAttrib*() function, 29, 31
glVertexAttribPointer() function, 244–246, 714, 715
glVertexAtttribDivisor() function, 268
glVertextAttribPointer() function, 266
glViewportArrayv function, 370
glViewPort() function, 41, 137, 361
glWaitSync() function, 559, 560
glXMakeCurrent() function, 721
Global illumination, 624
Global work groups, 470–471, 473
GLSL (OpenGL Shading Language)
  built-in functions, 213–219
  compilers, 219–224
  data types, 206–212
  overview of, 206–219
  storage qualifiers, 366
GLuint variables, 100
Gouraud shading, 571, 572
GPU PerfStudio, 703–706
GPUs (graphics processing units), 4, 5, 102
  multiple GPUs, 719–721
  work generation, 683–690
GPUView, 699–703
Gradients, 528, 634
Graphics
  3D. *See* 3D graphics
  acceleration, 8
  pipelines, 3. *See also* Pipelines (graphics)
  programs, 468
  resetting, 737–740
  two-dimensional, 647–659
Graphics library functions. *See* GL functions
Graphics processing units. *See* GPUs
Grass, drawing, 261–262
Grids, 262
Groups, debug, 735
Guard bands, 301–302

Half-spaces, 40
Handles, 504
Hangs, 738
Hardware, 4
  queues, 700
  timer queries, 546

Hatching patterns, 517
Hazards, 151
HDR (high dynamic range), 431, 432–433, 615
Heads-up display. *See* HUD
Height maps. *See* Displacement mapping
Hermite curves, 217
Hermite interpolation, 216
Hermite splines, 97
High dynamic range. *See* HDR
Higher-order surfaces, 329
High-frequency output, 422
Highlights, specular, 569–573. *See also* Lighting
High-quality texture filtering, 527–531
Histograms, 434
History of OpenGL, 6–10
Homogeneous vectors, 59
HUD (heads-up display), 546, 705

Identify matrices, 73–75
IEEE standards, 206
Illumination, global, 624. *See also* lighting
Images
  access, synchronizing, 194
  atomic operations, 188–194
  blocks, 516
  columns, 485
  data format classes, 185
  depth of field, 488, 490
  detail to, adding, 586
  focus, 488
  mipmaps, 171
  RED, 518
  types, 183
  units, 183, 184
  variables, 182
  viewing, 458–466
Immutable objects, 101
Implementation, 206, 642
Inclusive prefix sums, 480
Indexes
  calculating, 270
  data chunks, 752–753
  drawing commands, 251–257
  queries, 551–552
  regions, 517
  restart, 257
  uniform blocks, 126
`indices` parameter, 272
Indirect draws, 271–280. *See also* Drawing
Indirect rendering, 678–683
Infinite loops, shaders, 738
In flight, 4
Inheriting states, 675
`init()` function, 731
Initializing
  buffers, 103
  G-buffers, 616
  textures, 152–154
`in` keyword, 30
Inner loops, Julia sets, 633
Inner products, 60
Inputs
  arrays, 339
  compute shaders, 471–474
  flat, 366
  multiple vertex shaders, 113–116
  naming, 230
  smooth, 366
  vertex shaders, 244–249, 276
Inscattering, 606
Inserting
  barriers into applications, 146–147
  packets, optimizing, 676
Instancing, 257–265
  arrays, applying, 265–271
  rendering, 269, 270
  vertex shaders, 267
Integers
  framebuffers, 444–446
  interpolation, 366
  signed/unsigned, 207
  types, 183
Interfaces, 3
  blocks, 31–33
  matching, 227–231
  variables, declaring, 31
Interleaved attributes, 114, 116
Internal formats, 153
Interpolation, 44, 89
  curves, 91
  fragment shaders, 366–369
  Gouraud shading, 571
  linear, 528, 529
  smooth, 529
  splines, 95
Interrupt service routine. *See* ISR
Intersections
  points, 639
  ray-sphere intersection tests, 638
  testing, 645
Intra-member rules, 497
`int` scalar type, 206
Inverting conditions, 544
Invocations, 206
IRIS GL, 6
Isolines, tessellation using, 312–315
ISR (interrupt service routine), 702
Items, work, 48, 470
Iterating
  elements, 325
  flocking algorithms, 493

Jaggies, 422
Julia sets
  fractals, 696, 697
  rendering, 632–634

Keywords
    in, 30
    out, 30
.KTX files, 160–161, 743–745
ktxtool program, 743–745

Landscapes
    distance fields, 655
    fog, applying, 609
Languages
    built-in functions, 213–219
    compilers, 219–224
    data types, 206–212
    overview of, 206–219
Layers, 178
    abstraction, 4
    framebuffers, 397
    geometry shaders, rendering with, 411
    off-screen rendering, 397–403
Layouts
    column-major, 66
    column-primary, 66
    format layout qualifiers, 186
    qualifiers, 29
        ccw, 319
        cw, 319
        geometry shaders, 334
        multiple framebuffer attachments, 396
    uniforms, 122
.length() method, 212
Length of vectors, 63
Levels, generating mipmaps, 173
Levels of detail, clamping, 515
Libraries, sb7::vmath, 135
Light bloom, 438–444
Lighting, 568–609
    atmospheric effects, 605–609
    Blinn-Phong, 577–579
    calculating, 619, 640
    casting shadows, 599–605
    environment mapping, 587–597
    fragments, 619
    material properties, 597–599
    normal mapping, 582–586
    Phong, 568–577
        ambient light, 568
        diffuse light, 569
        shading, 573–577, 584
        specular highlights, 569–573
    rim lighting, 579–582
Linear interpolation, 369, 528, 529. *See also* Interpolation
Linear texture filtering, 527
Lines, 89
    clipping, 298–305
    endpoints, 517
Linkers, logs, 223–225
Linking programs, 225–227
Lists, object chunks, 755–757
Loading
    2D array textures, 178–180
    objects from files, 116–117
    textures from files, 160–164
Local workgroups, 48, 470–471, 473
Locations, uniforms, 118–119
Logical operations, 387–388
Logs
    compilers, retrieving, 221
    linkers, 223–225
Lookout matrices, 84–86
Loops
    infinite, shaders, 738
    inner, Julia sets, 633
    rendering, 137, 139

Macros, DECLARE_MAIN, 14
Magnification, 168
main() function, 17, 335, 342, 524
make_face function, 348
Management
    data, 503
        eliminating binding, 504–509
        generating compressed data, 519–525
        high-quality texture filtering, 527–531
        packed data formats, 525–527
        RGTC (Red–Green Texture Compression) compression schemes, 516–519
        sparsely populated textures, 509–515
        texture compression, 516–525
    pipelines (graphics), 533
        occlusion queries, 535–545
        pipeline state queries, 555–556
        queries, 534–556
        synchronization, 556–562
        timer queries, 545–549
        transform feedback queries, 549–555
    textures, committing, 513
Mandlebrot sets, rendering, 632
ManuallyLaidOutBlock uniform blocks, 125
Mapping
    buffers, 100, 102–103, 104, 699
        optimizing, 709–713
        ranges, 106
    bump, 582
    cubemaps, 402–403, 592–597
    data stores, 105
    displacement, 323
    environment, 587–597
    mipmaps, 153. *See also* Mipmaps
    normal, 582–586
    persistent maps, 664, 665, 691–692
    shadows, 600–605
    texture compression, 715
    tone, 434–438
    unmappable resources, 694–699
    vertex shaders, 248

Masking
  colors, 388–390
  sample masks, 420
Matching
  blocks, 32
  interfaces, 227–231
Material properties, 597–599, 680
Math
  for 3D graphics, 55
    common vector operators, 60–64
    curves, 91–94
    matrices, 64–69
    overview of, 56–57
    splines, 94–97
    transformations, 69–89
    vectors, 57–60
  built-in math functions, 215–217
Matrices, 208–210
  built-in, 213–215
  construction of, 66–69
  frustrum, 88
  identify, 73–75
  lookout, 84–86
  math (for 3D graphics), 64–69
  operators, 66–69
  orthographic, 89
  perspective, 88–89
  rotating, 68, 76–79
  scaling, 79–80
  shadows, 602
  TBN (tangent, bitangent, normal), 583, 619
  transformation, 72, 117
  translating, 68, 75–76
  updating, 137
`MatrixCompMult()` function, 214
`max` function, 569
Memory
  access, 692
  allocating, 100–107
  atomic operations, 142–144
  bandwidth, 195, 614
  buffers. *See* Buffers
  optimization, 723–726
  resident, 506
  synchronizing, 145–147
`MemoryBarrier()` function, 147, 687
Meshes, transforming, 492
Methods
  `glClearBufferfv()`, 15
  `length()`, 212
  `render()`, 14, 16
  `run`, 14
  special, 212
  `startup()`, 14
Minification, 168
Mipmaps, 153, 169–171
  applying, 174
  filtering, 171–173
  levels, generating, 173
  tails, 512
Models
  exploding, 344, 345
  lighting, 568–609. *See also* Lighting
  spaces, 71
Model-view transformations, 83–84, 137
Modes
  AFR (alternate frame rendering), 719
  `equal_spacing`, 317
  filtering, samplers, 164
  `fractional_even_spacing`, 317
  `fractional_odd_spacing`, 317
  GPU PerfStudio, 704
  points, tessellation, 315–316
  polygons, 319
  subdivision, tessellation, 316–319
  texture filtering, 527
  wrapping
    samplers, 164
    textures, 174–177
Modifying
  geometry in geometry shaders, 344–345
  patterns, 507
  primitive types in geometry shaders, 349–352
  vertex shaders, 30
Moving uniforms, 118–119
Multiple framebuffer attachments, 395–397
Multiple GPUs, 719–721
Multiple outputs in fragment shaders, declaring, 396
Multiple streams of storage, 352–353
Multiple textures, 166–167
Multiple vertex shader inputs, 113–116
Multiple viewport transformations, 361–364
Multi-sample antialiasing, 156, 415–417
Multi-sample textures, 417–421
Multi-sampling, 47
Multi-threading, 662–667, 721–723

Naming, 100
  inputs, 230
  output, 230
  packed data formats, 526
  textures, 392
NaN (not a number), 207
N-bytes, 123
Nearest texture filtering, 527
Negative reflection, 570
New primitive types, geometry shaders, 353–361
`NextPacket` function, 671
Noise in ambient occlusion, 628
Non-photo-realistic rendering, 610–613
`noperspective` qualifier, 368, 369

Normalization, 58, 246
Normalized device spaces, 40, 72
Normal mapping, 582–586
  deferred shading, 619–622
  texture compression, 715
Not a number. *See* NaN
Numbers, floating-point, 525
NVIDIA GeForce GTX 560 SE graphics cards, 701
Nyquist rates, 412

Objects
  buffers, updating, 104
  clipping, 304
  coordinates, 71
  culling, 686
  depth clamping, 380
  files, loading, 116–117
  framebuffers, 390
  immutable, 101
  lighting models. *See* Lighting
  list chunks, 755–757
  pipelines, 718
  programs, 17
  queries, generating, 549
  samplers, 164
  separable program, 717
  shaders, 17
  shadows. *See* Shadows
  spaces, 71
  stacks, 642
  synchronization, 557–562
  TBOs (texture buffer objects), 288
  UBOs (uniform buffer objects), 121
  VAOs (vertex array objects), 295, 493
Occlusion queries, 535–545
Off-screen rendering, 390–412
  framebuffer completeness, 403–407
  layers, 397–403
  multiple framebuffer attachments, 395–397
  in stereo, 407–412
Offsets
  polygons, 604
  uniform blocks with, 124, 125
One-dimensional color lookup tables, 610
Opacity, 15
OpenGL Shading Language. *See* GLSL
OpenMP (Open Multi-Processing), 662–664
Open Multi-Processing. *See* OpenMP
Operating systems, 5
Operations
  logical, 387–388
  resolve, customizing, 419
  stencil, 375
Operators, 210
  common vector, 60–64
  matrices, 66–69
Optimization, 661. *See also* Performance
  buffers, mapping, 709–713
  CPUs (central processing units), 661–677
  GPU work generation, 683–690
  indirect rendering, 678–683
  memory, 723–726
  multi-threading, 662–667
  overhead, reducing, 677–699
  packet buffers, 668–677
  performance analysis tools, 699–726
  rules for shaders, 576
  zero copy, 691–699
Order, winding, 42
Origins of OpenGL, 6–10
Orthogonal, 67
Orthographic matrices, 89
Orthonormal, 67
`out` keyword, 30
Outputs
  colors, 382–387
  compute shaders, 471–474
  control points, 319
  font rendering demos, 659
  high-frequency, 422
  naming, 230
  patches, 36
  variables, 228
  vertex shaders, 30, 249–251
Overdrawing, 614
Overhead, reducing, 677–699
Overloading, 15, 183, 213

Packed data formats, 247, 525–527
Packets
  buffers, 668–677
  present, 702
  specialization, 676–677
  standard DMA, 702
  standard queue, 702
Packing RGTC blocks, 522
`PackRGTC` function, 523
Pages, 509, 510
`PalettizeTexel` function, 523
Palettizing RGTC blocks, 522
Parallelism, 4
Parallel prefix sums, 479–492
Parameters
  `bindingindex`, 110
  `flags`, 101
  `indices`, 272
  points, 453–454
  `primitiveMode`, 287
  `readoffset`, 109
  shininess, 570
  `stride`, 111, 115
  uniforms, queries, 127–128
  `writeoffset`, 109

Particle systems, 448
  OpenMP (Open Multi-Processing), 664
Partitioning data, 692
Passing
  data
    between shaders, 319–322
    into shaders, 117
    from stage to stage, 30–33
    to vertex shaders, 28–28
  primitives, 40
Pass-through geometry shaders, 334–336
Pass-through vertex shaders, 346
Patches, 33, 305, 306
  cubic Bézier, tessellation, 329–333
  output, 36
Patterns
  hatching, 517
  modifying, 507
Pausing transform feedback, 286–288
Pentium processors, 8
Performance, 8, 661
  ALU (arithmetic and logic unit), 715
  bubbles, 557
  counters, 706
  CPUs (central processing units), 661–677
  extensions, 50. *See also* Extensions
  GPU work generation, 683–690
  indirect rendering, 678–683
  memory, 723–726
  multi-threading, 662–667
  overhead, reducing, 677–699
  packet buffers, 668–677
  shaders, compiling, 716–719
  zero copy, 691–699
Performance analysis tools, 699–726
  applications, tuning for speed, 706–726
  GPU PerfStudio, 703–706
  GPUView, 699–703
  WPT (Windows Performance Toolkit), 699–703
Per-fragment shininess, 598–599
Per patch data, 306
Persistent maps, 103, 664, 665, 691–692. *See also* Mapping
  fractals, rendering, 698
Perspective
  division, 40
  matrices, 88–89
Perspective-correct interpolation, 367–369
Phong lighting models, 568–577
  ambient light, 568
  diffuse light, 569
  shading, 573–577, 584
  specular highlights, 569–573
Physical simulation (transform feedback), 288–298
Pipelines (graphics), 3, 10–11, 27
  clipping, 39–40
  compute shaders, 48–49
  configuring, 226
  culling, 41–43
  draining, 557
  extensions, 49–54
  fragment shaders, 43–46
  framebuffer operations, 47–48
  geometry shaders, 37–39
  management, 533
  overview of, 4–6
  program objects, 718
  queries, 534–556
    occlusion, 535–545
    pipeline state, 555–556
    timer, 545–549
    transform feedback, 549–555
  rasterization, 43
  synchronization, 556–562
  tessellation, 33–37
  transform feedback, 280–298
  vertex shaders, 17
    inputs, 244–249
    passing data from stage to stage, 30–33
    passing data to, 28–28
  viewports, transforming, 40–41
Pixels, 10–11, 17
  framebuffer operations, 47–48
  interpolation. *See* Interpolation
Planes
  adding, 645
  depth clamping, 379
Points
  binding, 100
    assigning, 131
    transform feedback, 284
  control, 306. *See also* Control points
  drawing, 250
  intersections, 639
  parameters, 453–454
  rotating, 456–458
  sampling, 168, 527
  shaped, 454–456
  sizes, 250
  sprites, 448–458
  tessellation, 315–316
  textures, 449–450
Polygons, 11
  modes, 319
  offsets, 604
  smoothing, 414
Populating sparsely populated textures, 509–515
Positioning vertices, 117
Post-rasterization processing, 370
PowerPC processors, 8

Prefixes, parallel prefix sums, 479–492
Pre-fragment tests, 369–382
  depth testing, 376–380
  early testing, 380–382
  scissor testing, 369–372
  stencil testing, 372–376
Pre-optimizing shaders, 719
Present packets, 702
`PrimitiveMode` parameter, 287
Primitive processing, 305
  geometry shaders, 333–334
    changing primitive types, 349–352
    discarding geometry in, 341–344
    generating geometry in, 346–349
    modifying geometry in, 344–345
    multiple streams of storage, 352–353
    multiple viewport transformations, 361–364
    new primitive types, 353–361
    pass-through, 334–336
    using in applications, 336–341
  tessellation, 306–322
    communication between shader invocations, 322
    cubic Bézier patches, 329–333
    passing data between shaders, 319–322
    primitive modes, 307–316
    subdivision modes, 316–319
    terrain rendering, 323–328
Primitives, 10–11
  geometry shaders, 37
  passing, 40
  query results, 552–555
  restarts, geometry, 255–257
  synchronization, 107
  tessellation. *See* Tessellation
  types
    geometry shaders, modifying, 349–352
    new in geometry shaders, 353–361
  winding order of, 318–319
Printing interface information, 231
Processing
  .KTX files, 743–745
  post-rasterization, 370
  primitives. *See* Primitive processing
  vertices, 244–251
Processors, 8–10
Products
  cross, 62–63
  dot, 60–61
  inner, 60
  vectors, 62
Profiles, 9
Programs, 13. *See also* Applications; Tools
  binaries, 235–238, 719
  bloom, 438–444
  computer, 468
  creating, 14–16
  `dds2ktx`, 745–746
  graphics, 468
  `ktxtool`, 743–745
  linking, 225–227
  objects, 17
  pipeline objects, 225, 718
  `sb6mtool`, 746–748
  separable objects, 717
  shaders, 17–24
  triangles, drawing, 24–26
Projection
  2D data, 69
  matrices, updating, 137
  transformations, 86–88
Properties, material, 597–599
`props` arrays, 229
Publication dates, 7
Pulling (vertex), 28
Pythagoras's theorem, 63

Quad fragments, 556
Quadratic Bézier curves, 92
Quads
  geometry shaders, rendering, 355–361
  tessellation using, 307–310
Qualifiers
  centroid, applying, 424
  `flat`, 367
  format layout, 186
  interpolation, 367
  layouts, 29
    `ccw`, 319
    `cw`, 319
    geometry shaders, 334
    multiple framebuffer attachments, 396
  `noperspective`, 368, 369
  `shared` storage, 475
  `smooth`, 367
  storage, fragment shaders, 366–369
Quaternions, 79, 81–83
Queries, 534–556
  control flow, 545
  indexes, 551–552
  objects, generating, 549
  occlusion, 535–545
  pipeline state, 555–556
  results
    applying, 537–540
    primitive, 552–555
    rendering, 540
    retrieving, 536–537
  timer, 545–549
  transform feedback, 549–555
  uniform parameters, 127–128

Queues
    CPUs (central processing units), 700
    hardware, 700
    software, 700
    standard queue packets, 702
Quintic Bézier curve, 94

Race conditions, 476, 477, 478, 692
Radians, 217
Range-checked reads, 740–742
Ranges, mapping buffers, 106
Rasterization, 43, 365, 366
    fragment shaders. *See* Fragment shaders
    post-rasterization processing, 370
Rasterizers, 11
    disabling, 288
    discarding, 296
Rates, sample rate shading, 421–423
RAW (read-after-write) hazard. *See* RAW145
Ray-sphere intersection tests, 638
Ray tracing in fragment shaders, 634–647
Read-after-write. *See* RAW
Reading
    from framebuffers, 458–461
    range-checked reads, 740–742
    states, 707–709
    textures, 156–160, 164–177
Read–modify–write cycle, 143
`readoffset` parameter, 109
Rectangles
    pages, 509
    scissor testing, 369–372
    summed area tables, 487
    textures, 155
Red–Green Texture Compression. *See* RGTC
RED images, 518
Reducing overhead, 677–699
Redundancy, state setting commands, 672
Reflection, 63–64
    negative, 570
    vectors, 571
Refraction, 63–64
`ref` value, 374
Regions
    hatching patterns, 517
    pages, 509
Rendering, 11, 567
    AFR (alternate frame rendering) mode, 719
    alternative rendering methods, 613–647
    attachments, 392
    with blending functions, 385
    colors, distance fields, 654–655
    conditional, 541–543
    to cubemaps, 402–403
    flexible indirect, 678–683
    fonts, distance fields, 652–654
    fractals, 698
    framebuffers with no attachments, 428–430
    geometry, 267
    indirect, 678–683
    instancing, 258, 269, 270
    Julia sets, 632–634
    layers with geometry shaders, 411
    lighting models, 568–609
    loops, 137, 139
    Mandlebrot sets, 632
    multiple viewports, 363
    non-photo-realistic, 610–613
    off-screen, 390–412
        framebuffer completeness, 403–407
        layers, 397–403
        multiple framebuffer attachments,
            395–397
    quads using geometry shaders, 355–361
    query results, 540
    shadow maps, 605
    skyboxes, 594
    spherical environment maps, 587–589
    star fields, 450–453
    in stereo, 407–412
    terrain, 323–328
    textures, 394, 395
    two-dimensional graphics, 647–659
    with user clip distances, 305
    without tessellation control shaders,
            321–322
    without triangles, 631–647
`render()` method, 14, 16
Resetting
    graphics, 737–740
    packet buffers, 674
Resident memory, 506
Resolution, 5
Resolve operations, customizing, 419
Resources, unmappable, 694–699
Results
    atomic counters, 151
    queries
        applying, 537–540
        primitive, 552–555
        rendering, 540
        retrieving, 536–537
Retrieving query results, 536–537
Return values, framebuffer completeness, 405
RGTC (Red–Green Texture Compression), 196,
            516–519
Rim lighting, 579–582
Robustness, 737–742
Rotating
    matrices, 68, 76–79
    points, 456–458
Roughness, applying, 597
Rules
    flocking algorithms, 497
    shaders, 576
`run()` method, 14

Sample masks, 420
Sample rate shading, 421–423
Samplers
  antialiasing, 415. *See also* Antialiasing
  filtering modes, 164
  types, 157–160
  uniform blocks, declaring inside, 505, 507
  variables, 156, 504
  wrapping modes, 164
Sampling
  centroid, 424–428
  points, 168, 527
`sb6mtool` program, 746–748
`sb7::vmath` library, 135
.SBM files, 746–748, 749–757
  chunk headers, 750–751
  comment chunks, 755
  defined chunks, 751–755
  file headers, 749–750
  object list chunks, 755–757
Scalability, 4
Scalars, 65, 206–207
Scaling matrices, 79–80
Schematics, tessellation, 307
Scintillation (aliasing artifacts), 169
Scissor testing, 47, 369–372
Screenshots, formatting, 460–461
Screen-space ambient occlusion. *See* SSAO
Screen-space techniques, 624–631
Security, 737–742
Segments, fractional, 317
Separable program objects, 717
Separate attributes, 114
Separate programs, 225–227
Serialization, 144
SGI (Silicon Graphics Inc.), 6
Shaders
  applying, 17–24
  atomic counters, declaring, 148
  atomic operations, 143–144
  barriers, applying, 147
  compiling, 716–719
  compute, 48–49. *See also* Compute shaders
  control (tessellation), 33–35
  cores, 5
  evaluation (tessellation), 35–37
  fragments, 17, 43–46. *See also* Fragment shaders
    spinning cubes, 138
    terrain rendering, 327
  geometry, 37–39. *See also* Geometry shaders
  infinite loops, 738
  logs, retrieving compiler, 221
  objects, 17
  optimization rules, 576
  pre-optimizing, 719
  programs, linking, 225–227
  rim lighting, 581
  storage, blocks, 140–147
  subroutines, 231–235
  TCSs (tessellation control shaders), 306
  testing, 381. *See also* Testing
  textures
    reading from, 156–160
    writing in, 182–194
  updating, 112
  vertex, passing data to, 28–28
  vertices, 17
Shading. *See also* Shadows
  cell, 610–613
  deferred, 613–624
  Gouraud, 571, 572
  normal mapping, 619–622
  Phong lighting models, 573–577, 584
  ray tracing, 636
  sample rate, 421–423
Shadows
  casting, 599–605
  mapping, 600–605
  matrices, 602
Shaped points, 454–456
`shared` storage qualifier, 475
Sharing
  data, partitioning, 692
  exponents, 198–199
  layouts, uniforms, 122
Shininess, per-fragment, 598–599
Shininess factors, 570
Side effects, 473
Signaled state, 557
Silicin Graphics Inc. *See* SGI
Sizes of input arrays to geometry shaders, 339
Skyboxes, rendering, 594
Slices, time, 475
Smoothing polygons, 414
Smooth inputs, 366
Smooth interpolation, 529
`smooth` qualifier, 367
SoAs (structure-of-arrays), 115
Software, 700. *See also* Applications
Sorting values, 521
Source factors, 383
Spaces
  clip, 72
  coordinates, 70–72
  models, 71
  normalized device, 72
  objects, 71
Sparsely populated textures, 509–515
`sparsetexture` application, 512–514
Specialization, packet buffers, 676–677
Special methods, 212
Specifying bindings, uniform blocks, 133
Specular albedo, 576
Specular highlights, 569–573
Speed, tuning applications for, 706–726

Spherical environment maps, 587–589
Splines, 89, 94–97
Sprites (points), 448–458
SRGB color spaces, 446–448
SSAO (screen-space ambient occlusion), 624
Stacking
  fields, 209
  implementation, 642
Stages, 5
  data from stage to, passing, 30–33
  of pipelines, 28
Standard DMA packets, 702
Standard layouts, 123. *See also* Layouts
Standard operators, 210
Standard queue packets, 702
Standards, IEEE, 206
Star fields, rendering, 450–453
Starting
  occlusion queries, 537
  transform feedback, 286–288
`startup()` function, 14, 663
States
  packet buffers, chaining, 675
  reading, 707–709
Stencil testing, 47, 372–376
Stereo, rendering in, 407–412
Stopping transform feedback, 286–288
Storage, 101. *See also* Data stores
  allocating, 509, 691
  atomic counters, 148
  atomic operations, 143–144
  buffers, flags, 102
  multiple streams of, geometry shaders, 352–353
  qualifiers, fragment shaders, 366–369
  resident memory, 506
  shaders, blocks, 140–147
  textures. *See* Textures
  world-space coordinates, 615
Streams, transform feedbacks, 554
`stride` parameter, 111, 115
Stripping tools, 255
Structure-of-arrays. *See* SoAs
Structures, 210–212
  acceleration, 646
  `gl_PerVertex`, 320
  packet data, 669
  VERTEX, 245
Subdivision modes, tessellation, 316–319
Subroutines
  shaders, 231–235
  uniforms
    configuring values, 235
    declaring, 232
Summed area tables, 485, 487
Sums, parallel prefix, 479–492
Support
  data types, 119
  packed data formats, 527
  swizzling, 209
  textures, 510
  types, 108
Surfaces
  bumpy, 625
  higher-order, 329
`SwapBuffers()` command, 702
Swizzling, 209
Synchronizing, 666, 701
  access
    to atomic counters, 151–152
    images, 194
    memory, 145–147
  access to buffers, 692–694
  compute shaders, 475–479
  objects, 557–562
  pipelines (graphics), 556–562
  primitives, 107
Systems, updating, 291

Tables
  one-dimensional color lookup, 610
  summed area, 485, 487
Tails, mipmaps, 512
Tangent, bitangent, normal. *See* TBN matrix
Tangents
  space normals, 619
  vectors, 582
Targets, 100
  textures, 154–156
  views, 200
TBN (tangent, bitangent, normal) matrix, 583, 584, 619
TBOs (texture buffer objects), 288
TCSs (tessellation control shaders), 306, 704
Terrain, rendering, 323–328
TESs (tessellation evaluation shaders), 306, 704
Tessellation, 33–37, 227, 250
  communication between shader invocations, 322
  control shaders, 33–35
  cubic Bézier patches, 329–333
  data, passing between shaders, 319–322
  engines, 35
  evaluation shaders, 35–37, 607
  geometry shaders, configuring, 347
  landscapes, applying fog, 609
  point modes, 315–316
  primitives
    modes, 307–316
    processing, 306–322
  schematics, 307
  subdivision modes, 316–319
  terrain rendering, 323–328

transform feedback queries, 549
triangles, 37
using isolines, 312–315
using quads, 307–310
using triangles, 310–312
Tessellation control shaders. *See* TCSs
Tessellation evaluation shaders. *See* TESs
Testing
depth, 47
intersections, 645
pre-fragment, 369–382
depth testing, 376–380
early testing, 380–382
scissor testing, 369–372
stencil testing, 372–376
ray-sphere intersection, 638
ray-sphere intersection tests, 638
scissor, 47
stencil, 47
`texelFetch()` function, 167
Texels
blocks, fetching, 520
as light, 610
texture filtering, 528
Texture buffer objects. *See* TBOs
`texture()` function, 167
`textureGatherOffset` function, 520
Textures, 11, 152–202
arrays, 177–182, 397
copying, 411
viewing, 400
atlas, 509
attachments, 392
bindless, 504, 682
committing, 513
compression, 195–199, 516–525, 715
coordinates, 162–164, 209
data
copying into, 463–464
reading back, 464–466
distance fields, 647–655
files, loading from, 160–164
filtering, 167–169
floating-point formats, 432
formatting, 152–154
framebuffers, configuring, 390
gradients, 634
high-quality texture filtering, 527–531
initializing, 152–154
interpolation, 368
mipmaps, 169–171
multiple, 166–167
multi-sample, 417–421
points, 449–450
reading, 164–177
rectangles, 155
rendering, 394, 395
resident, applying, 508
shaders
reading from, 156–160
writing in, 182–194
sparsely populated, 509–515
support, 510
targets, 154–156
types, 154–156
unbinding, 504
units, 152
updating, 153
views, 199–202
wrapping modes, 174–177
Threading
data, generating, 662–667
multi-threading, 662–667, 721–723
Tightly packed arrays, 115
Time, slices, 475
Timer queries, 545–549
Tokens, types, 108
Tone mapping, 434–438
Tools, 743
`dds2ktx` program, 745–746
`ktxtool` program, 743–745
performance analysis, 699–726
GPU PerfStudio, 703–706
GPUView, 699–703
tuning applications for speed, 706–726
WPT (Windows Performance Toolkit), 699–703
`sb6mtool` program, 746–748
stripping, 255
Toon shaders, 611, 612
Transfer curves, 436
Transformation, 69–89
concatenation, 80–81
coordinates, 65, 70–72, 72–80
geometry, uniforms, 135–140
matrices, 72, 117
meshes, 492
model-view, 83–84, 137
multiple viewport, geometry shaders, 361–364
projection, 86–88
transform feedback, 280–298
ending pipelines with, 288
physical simulation, 288–298
viewports, 40–41
Transform feedback queries, 549–555
Translating matrices, 68, 75–76
Triangles, 11
clipping, 299, 300. *See also* Clipping
drawing, 24–26
geometry shaders, 348
Phong shading, 573
rendering without, 631–647

Triangles (*continued*)
  strips, 257
  tessellation, 37, 310–312
Tuning applications for speed, 706–726
Two-dimensional graphics, 647–659, 655–659
Types
  data, 206–212
    packed, 525
    support, 119
  elements, 211
  images, 183
  integers, 183
  rendering, 11
  samplers, 157–160
  textures, 154–156
  tokens, 108
  vertices, attributes, 246

UBOs (uniform buffer objects), 121
Unbinding textures, 504
Under-sampling data, 412
Uniform buffer objects. *See* UBOs
Uniforms, 117–140
  blocks, 121–135
    declaring, 121, 680
    defaults, 117–120
    indexes, 126
    specifying bindings, 133
  configuring, 119–120
  geometry, transforming, 135–140
  moving, 118–119
  parameters, queries, 127–128
  samplers, declaring blocks inside, 505, 507
  subroutines
    configuring values, 235
    declaring, 232
Unions of packets, 670
Units
  images, 183, 184
  textures, 152
  vectors, 58
Unmappable resources, 694–699
Unmapping buffers, 105. *See also* Mapping
Unpacking data from G-buffers, 617
Unsignaled state, 557
`unsigned int` scalar type, 206
Updating
  buffers, objects, 104
  depth buffers, 378–379
  flocking, 499
  OpenMP (Open Multi-Processing), 664
  projection matrices, 137
  shaders, 112
  stencil buffers, 376
  systems, 291
  textures, 153
  vertices, attributes, 29
User-defined clipping, 302–305
User-defined framebuffers, 392. *See also* Framebuffers

Values
  `primitiveMode` parameter, 287
  ref, 374
  return, framebuffer completeness, 405
  sorting, 521
VAOs (vertex array objects), 20, 109, 295, 493, 704
Variables, 25
  filtering, 487
  GLuint, 100
  images, 182
  interfaces, declaring, 31
  output, 228
  point sizes, 250
  samplers, 156, 504
Varyings, 281–285
  geometry shaders, 338
Vectors, 57–60, 208–210
  bitangent, 582
  common operators, 60–64
  functions, 213–215
  length of, 63
  products, 62
  reflection, 571
  rim lighting, 580
  tangent, 582
Velocity, buffers, 493
Versions, 7, 759–796
Vertex array objects. *See* VAOs
Vertex shaders
  array textures, 400
  attributes, applying, 113
  buffers, feeding from, 109–117
  colors, 266
  cubemaps, 596
  flocking, 500
  Gouraud shading, 572
  inputs, 244–249, 276
  instancing, 267
  mapping, 248
  material indexes, passing through, 681
  multiple inputs, 113–116
  normal mapping, 585
  outputs, 249–251
  passing data, 28–29, 30–33
  pass-through, 346
  Phong shading, 575
  shadow mapping, 603
  skybox rendering, 594
  spherical environment mapping, 588
  spinning cubes, 138
  systems, updating, 291
  texture coordinates, 163
  toon, 611

VERTEX structure, 245
Vertices, 6, 12
  attributes, 28–29, 110
    configuring, 112
    disabling, 113
    shader storage blocks, 142
    types, 246
  base vertex, 254–255
  buffers, bindings, 110
  chunks, 753–755
  connections, 289
  declaring, 247
  generating, 347
  positioning, 117
  processing, 244–251
  shaders. *See* Vertex shaders
Viewing
  array textures, 400
  extensions, 51
  images, 458–466
Viewports
  multiple transformation, 361–364
  transformation, 40–41
Views
  big-picture, 11
  coordinates, 71–72
  model-view transformations, 83–84
  space, 620
  textures, 199–202
Virtual framebuffers, 430. *See also* Framebuffers
`vmath` library, 61
VSync, 701

Waiting, back-to-back, 536
WAR (write-after-read) hazard, 145
WAW (write-after-write) hazard, 145
Weather, 605–609
Whole framebuffer completeness, 404
Winding order, 42
  of primitives, 318–319
Windows
  coordinates, 41
  stereo
    drawing, 409
    formatting, 408
Windows Performance Toolkit. *See* WPT
Workgroups, 470–471, 473
  local, 48
Work items, 48, 470
Workstations, 6
World coordinates, 71
World-space coordinates, 615
WPT (Windows Performance Toolkit), 699–703
Wrapping modes
  samplers, 164
  textures, 174–177
Write-after-read. *See* WAR
Write-after-write. *See* WAW
Write combined memory buffers, 699
`writeoffset` parameter, 109
Writing
  to G-buffers, 616
  textures in shaders, 182–194

Zero copy, 691–699

This page intentionally left blank

informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE

PEARSON **InformIT** is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the leading brands, authors, and contributors from the tech community.

Addison-Wesley Cisco Press IBM Press Microsoft Press

PEARSON IT CERTIFICATION PRENTICE HALL que SAMS vmware PRESS

LearnIT at InformIT

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials. Looking for expert opinions, advice, and tips? **InformIT has a solution**.

- Learn about new releases and special promotions by subscribing to a wide variety of monthly newsletters. Visit **informit.com/newsletters**.
- FREE Podcasts from experts at **informit.com/podcasts**.
- Read the latest author articles and sample chapters at **informit.com/articles**.
- Access thousands of books and videos in the Safari Books Online digital library. **safari.informit.com**.
- Get Advice and tips from expert blogs at **informit.com/blogs**.

Visit **informit.com** to find out all the ways you can access the hottest technology content.

Are you part of the IT crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube and more! Visit **informit.com/socialconnect**.

ALWAYS LEARNING PEARSON

Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to **informit.com/register** to sign in or create an account. You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

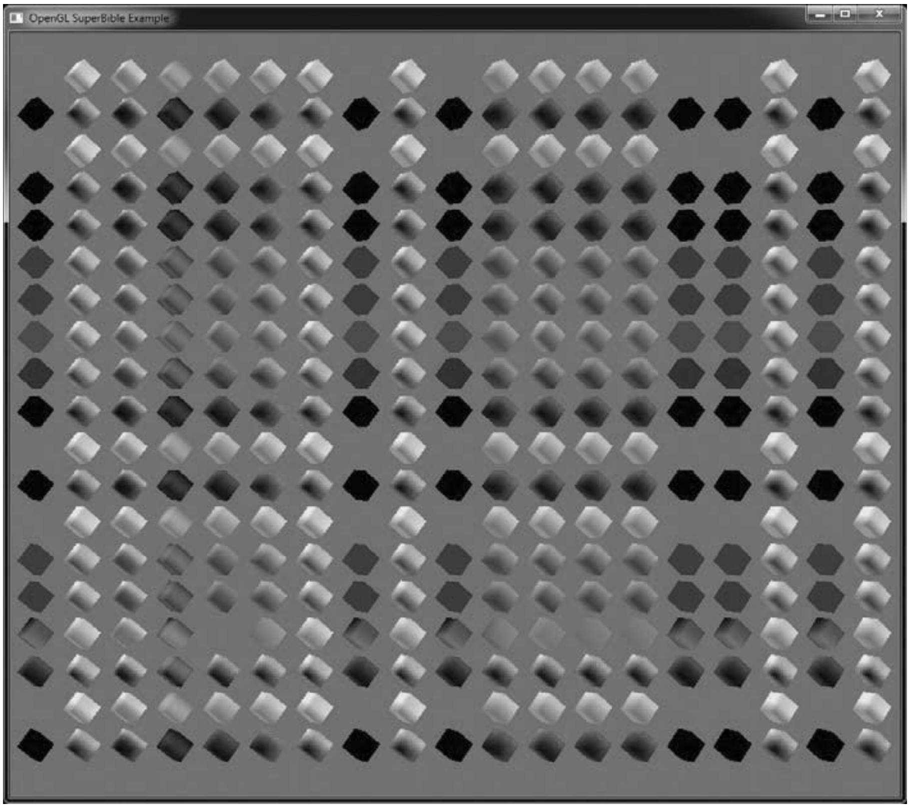
Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.

About InformIT — THE TRUSTED TECHNOLOGY LEARNING SOURCE

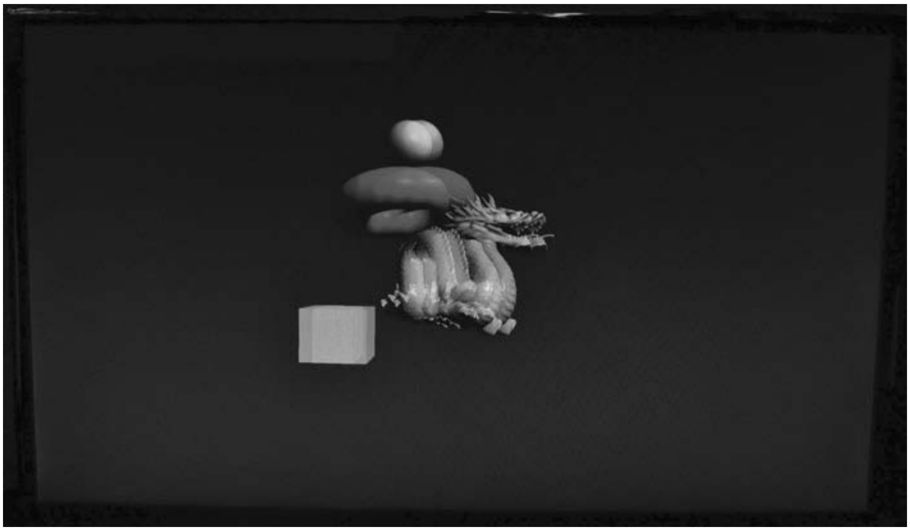
INFORMIT IS HOME TO THE LEADING TECHNOLOGY PUBLISHING IMPRINTS Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que, and Sams. Here you will gain access to quality and trusted content and resources from the authors, creators, innovators, and leaders of technology. Whether you're looking for a book on a new technology, a helpful article, timely newsletters, or access to the Safari Books Online digital library, InformIT has a solution for you.

informIT.com
THE TRUSTED TECHNOLOGY LEARNING SOURCE

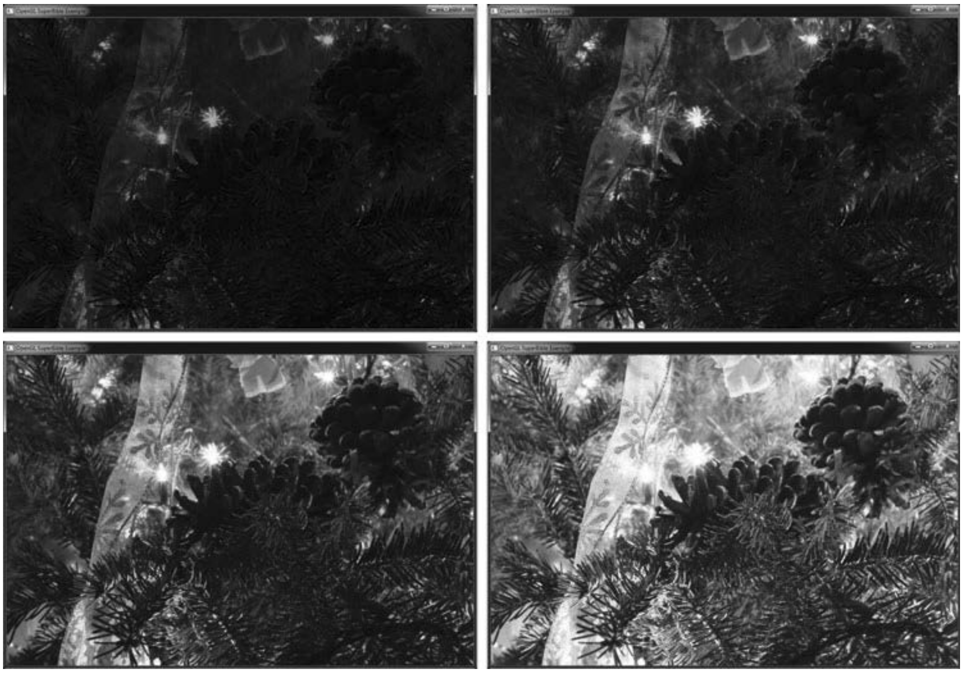
Addison-Wesley | Cisco Press | Exam Cram
IBM Press | Que | Prentice Hall | Sams
SAFARI BOOKS ONLINE



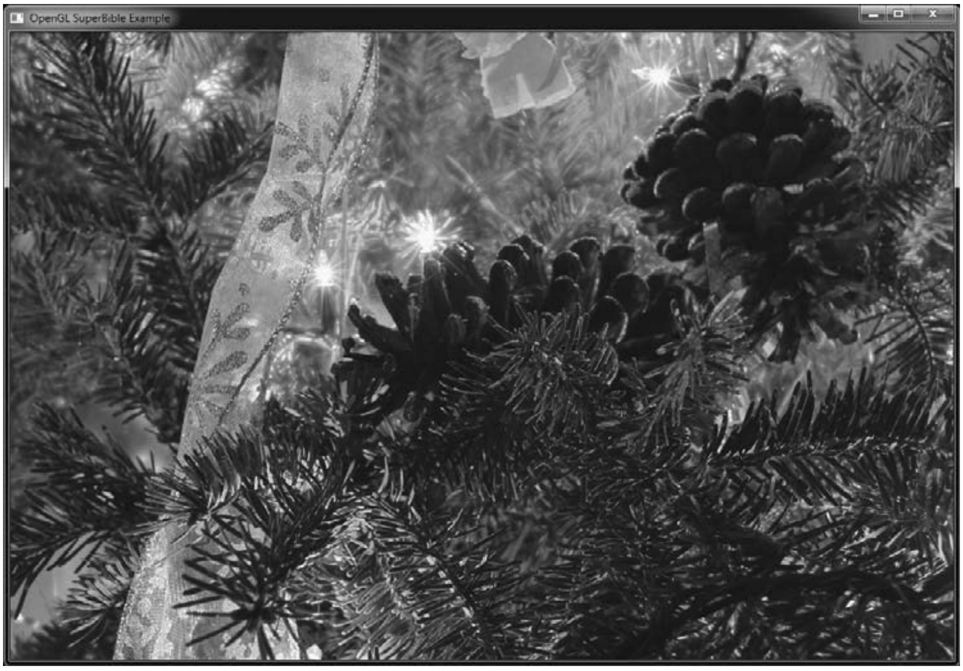
Color Plate 1: All possible combinations of blend function



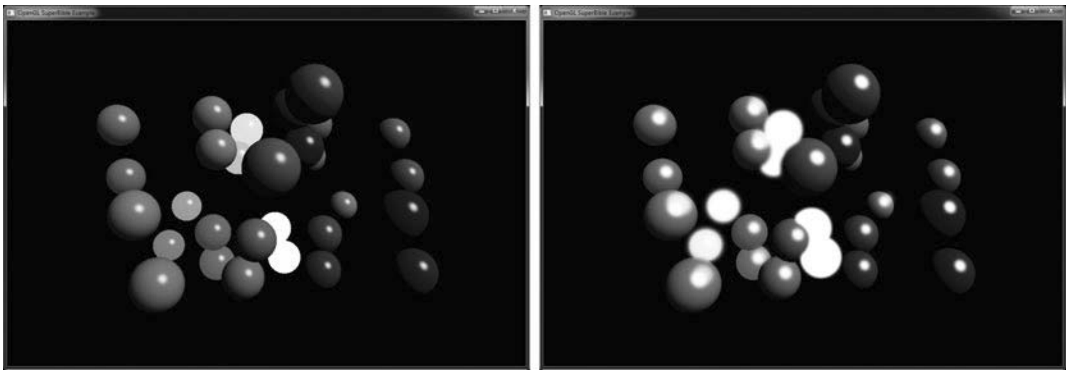
Color Plate 2: Rendering to a stereo display



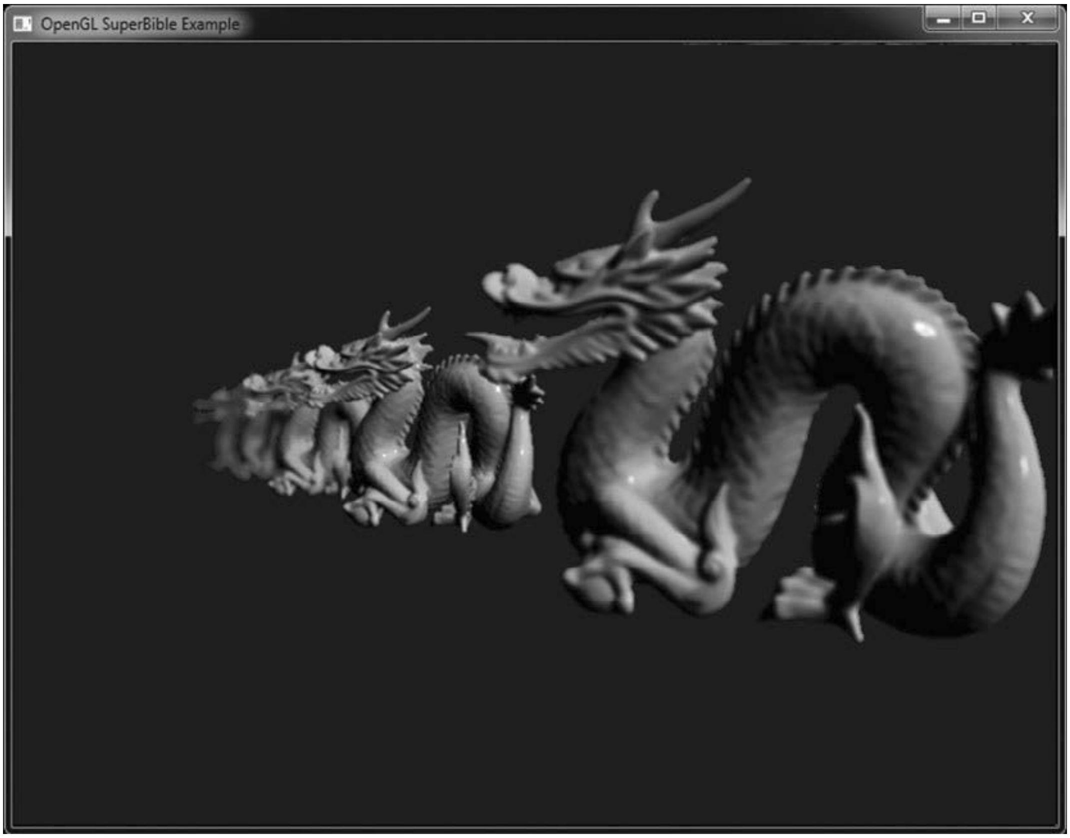
Color Plate 3: Different views of an HDR image



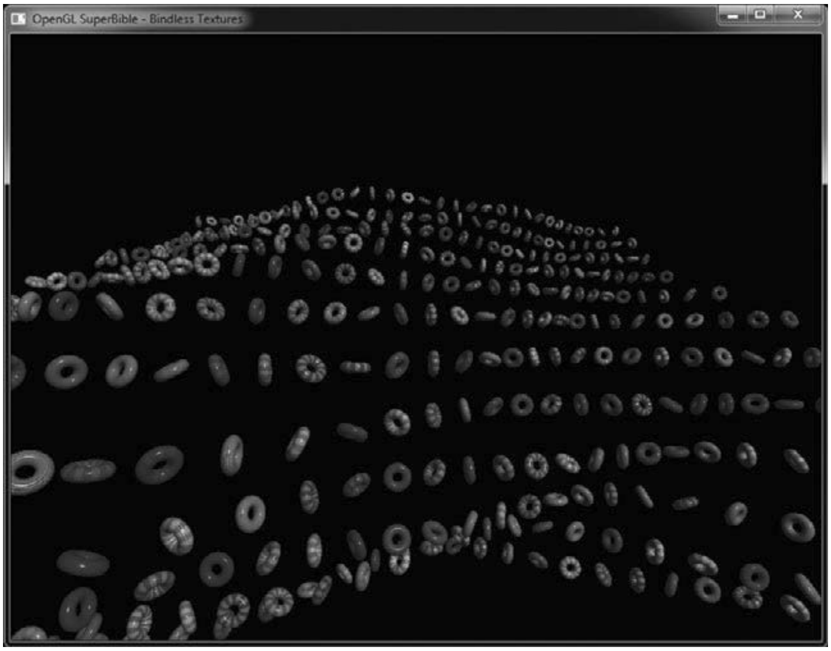
Color Plate 4: Adaptive tone mapping



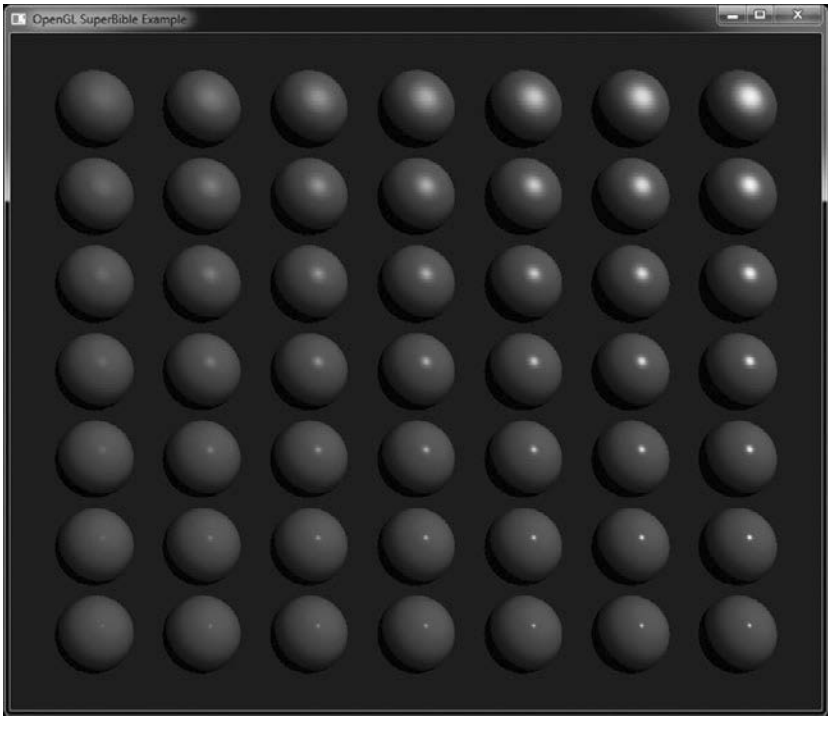
Color Plate 5: Bloom filtering: no bloom (left) and bloom (right)



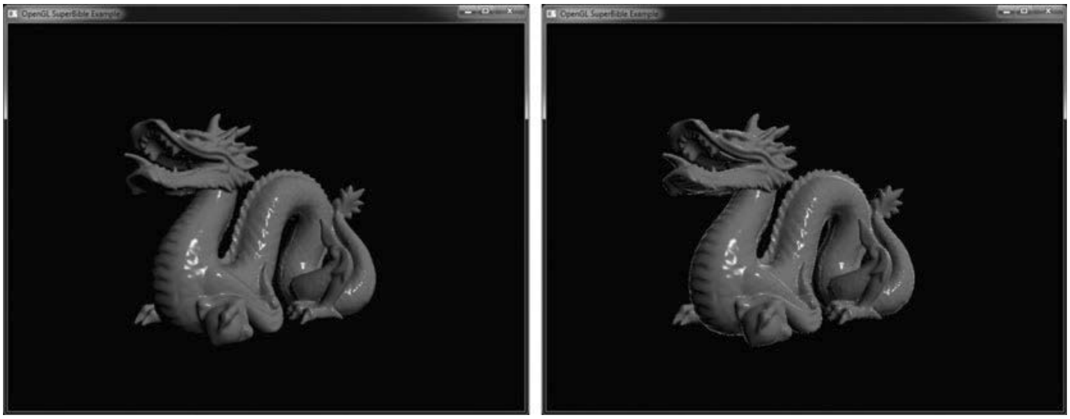
Color Plate 6: Depth of field applied to an image



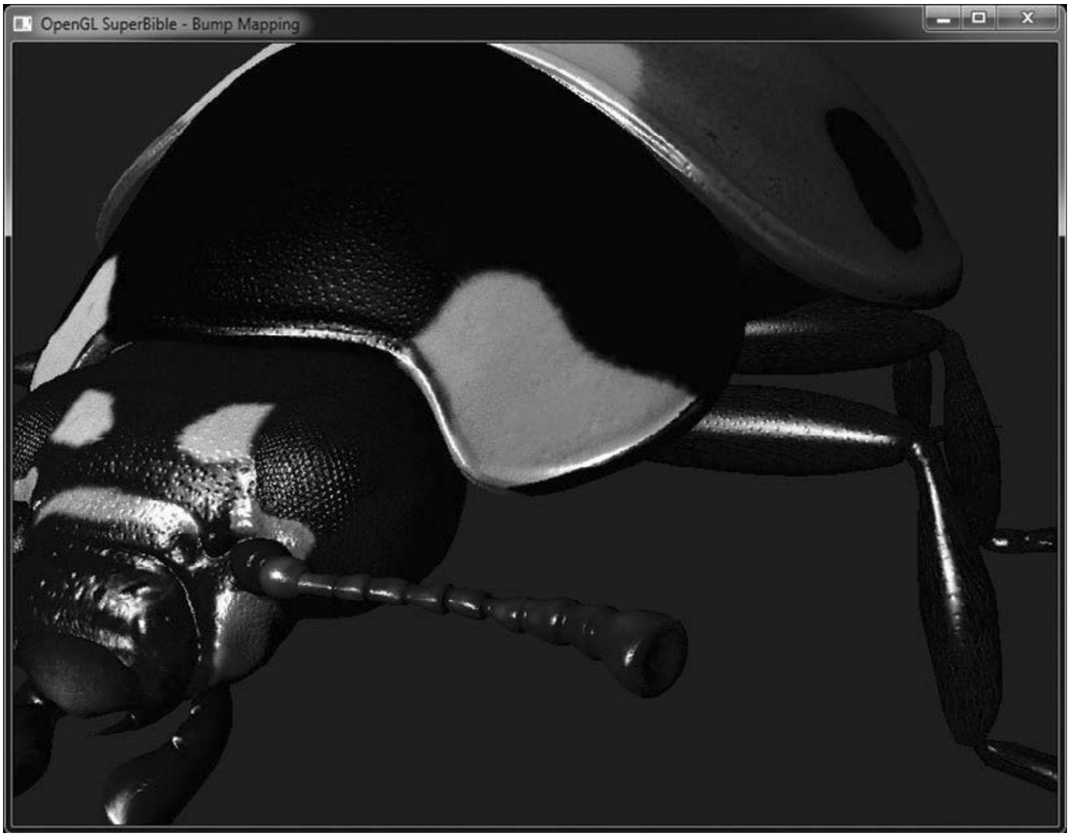
Color Plate 7: Output of bindless texture example



Color Plate 8: Varying specular parameters of a material

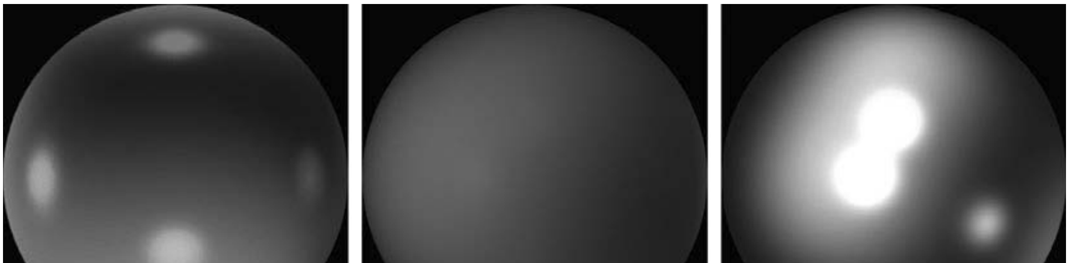


Color Plate 9: Result of rim lighting example



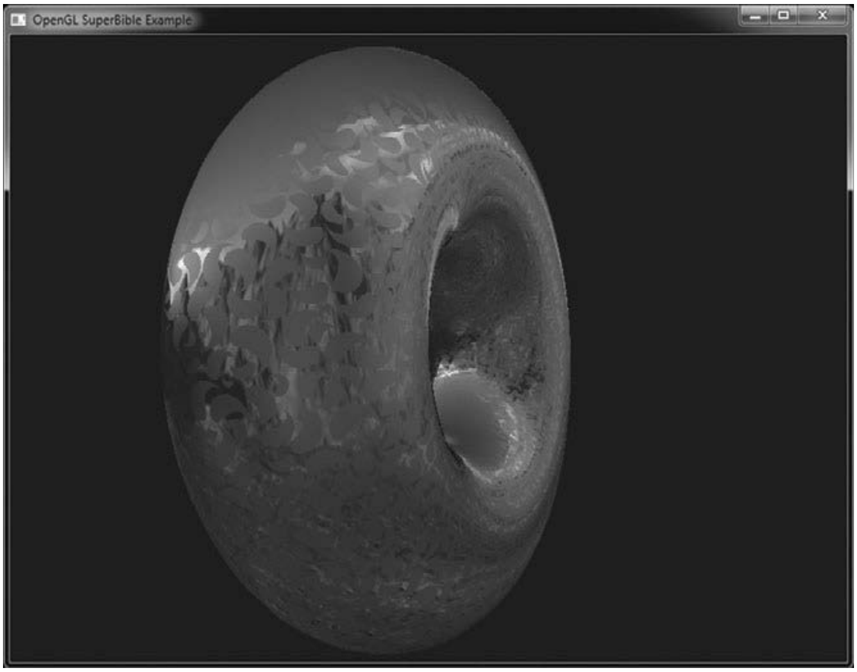
Color Plate 10: Normal mapping in action

Color Plate 11: Depth of field applied in a photograph

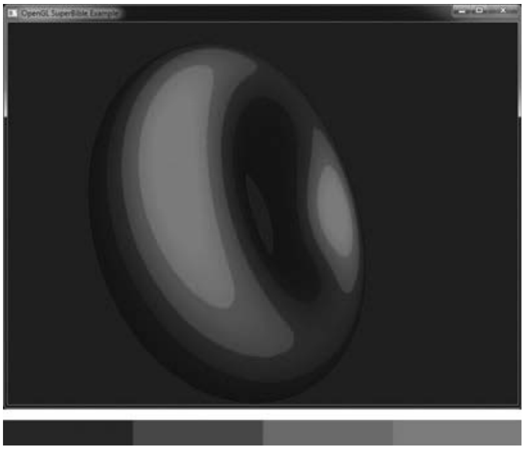

Color Plate 12: A selection of spherical environment maps

OpenGL SuperBible Example

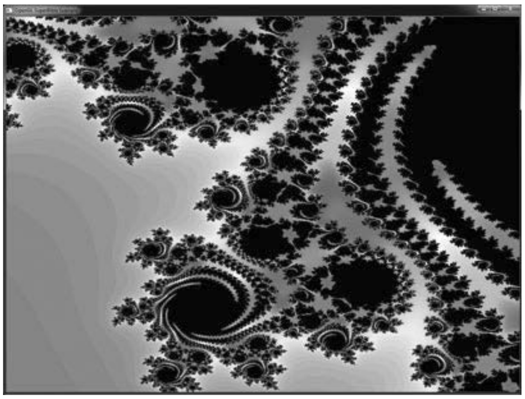
Color Plate 13: A golden environment-mapped dragon



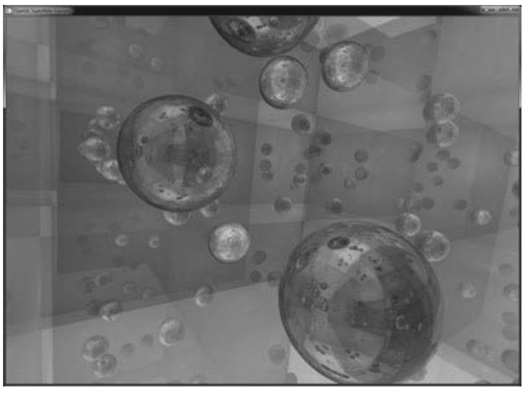
Color Plate 14: Result of per-pixel gloss example



Color Plate 15: Toon shading output with color ramp



Color Plate 16: Real-time rendering of the Julia set



Color Plate 17: Ray tracing with four bounces