OpenGL® SuperBible

Sixth Edition

Comprehensive Tutorial and Reference

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

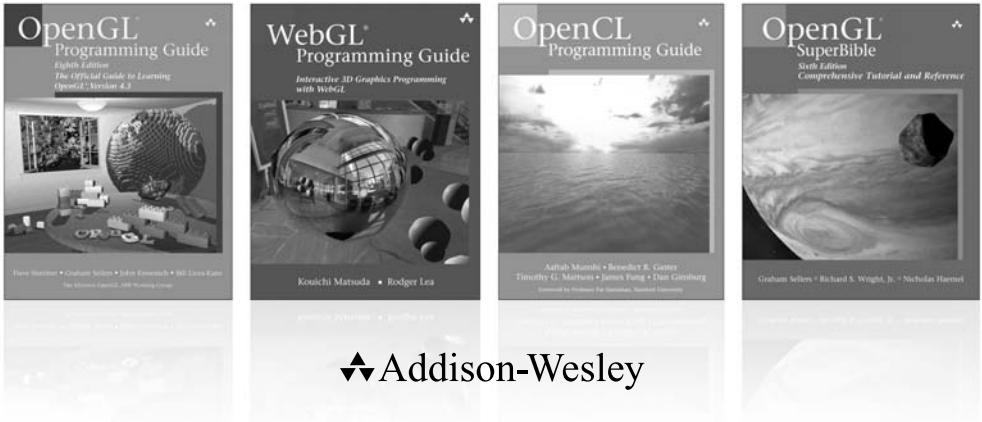
OpenGL®

SuperBible

Sixth Edition

OpenGL Series

from 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 Books Online

OpenGL®

SuperBible

Sixth Edition

Comprehensive Tutorial and Reference

Graham Sellers
Richard S. Wright, Jr.
Nicholas Haemel

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Wright, Richard S., Jr., 1965- author.
OpenGL superBible : comprehensive tutorial and reference.—Sixth edition / Graham Sellers, Richard S. Wright, Jr., Nicholas Haemel.
pages cm
Includes bibliographical references and index.
ISBN-13: 978-0-321-90294-8 (pbk. : alk. paper)
ISBN-10: 0-321-90294-7 (pbk. : alk. paper)
1. Computer graphics. 2. OpenGL. I. Sellers, Graham, author. II. Haemel, Nicholas, author. III. Title.
T385.W728 2013
006.6'8—dc23

2013016852

ISBN-13: 978-0-321-90294-8
ISBN-10: 0-321-90294-7
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, July 2013

Editor-in-Chief
Mark L. Taub

Executive Editor
Laura Lewin

Development Editor
Sheri Cain

Managing Editor
John Fuller

Full-Service Production Manager
Julie B. Nahil

Copy Editor
Teresa D. Wilson

Indexer
Larry Sweazey

Proofreader
Andrea Fox

Technical Reviewers
Piers Daniell
Daniel Koch
Daniel Rakos

Editorial Assistant
Olivia Basegio

Compositor
LaurelTech

For my family and my friends.
For those from whom I have learned.
For people who love to learn.
—Graham Sellers

For my wife LeeAnne,
for not killing me in my sleep
(when I deserved it).
To the memory of Richard S. Wright, Sr.
Thanks, Dad, for just letting me be a nerd.
—Richard S. Wright, Jr.

For my wife, Anna,
who has put up with all my engineering nonsense all these
years and provided undying love and support.
And to my parents for providing me with encouragement and
more LEGOs than I could get both arms around.
—Nicholas Haemel

This page intentionally left blank

Contents

Figures **xv**
Tables **xxiii**
Listings **xxv**
Foreword **xxxiii**
Preface **xxxv**
About This Book xxxv
The Architecture of the Book xxxvi
What's New in This Edition xxxviii
How to Build the Samples xxxix
Errata xl
Acknowledgments **xli**
About the Authors **xlv**

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 16
Drawing Our First Triangle 24
Summary 25

3 Following the Pipeline **27**
Passing Data to the Vertex Shader 28
Vertex Attributes 28
Passing Data from Stage to Stage 29
Interface Blocks 31
Tessellation 32
Tessellation Control Shaders 33
The Tessellation Engine 34
Tessellation Evaluation Shaders 34
Geometry Shaders 36
Primitive Assembly, Clipping, and Rasterization 38
Clipping 38
Viewport Transformation 39
Culling 40
Rasterization 41
Fragment Shaders 42
Framebuffer Operations 45
Pixel Operations 45
Compute Shaders 47
Summary 48

4 Math for 3D Graphics **49**
Is This the Dreaded Math Chapter? 50
A Crash Course in 3D Graphics Math 51
Vectors, or Which Way Is Which? 51
Common Vector Operators 54
Matrices 58
Matrix Construction and Operators 60
Understanding Transformations 63
Coordinate Spaces in OpenGL 63
Coordinate Transformations 66
Concatenating Transformations 73
Quaternions 75
The Model-View Transform 76
Projection Transformations 79

Interpolation, Lines, Curves, and Splines 82
Curves 83
Splines 87
Summary 90

5 Data **91**
Buffers 92
Allocating Memory using Buffers 92
Filling and Copying Data in Buffers 95
Feeding Vertex Shaders from Buffers 97
Uniforms 103
Default Block Uniforms 103
Uniform Blocks 108
Using Uniforms to Transform Geometry 121
Shader Storage Blocks 126
Synchronizing Access to Memory 129
Atomic Counters 133
Synchronizing Access to Atomic Counters 137
Textures 137
Creating and Initializing Textures 138
Texture Targets and Types 139
Reading from Textures in Shaders 141
Loading Textures from Files 144
Controlling How Texture Data Is Read 148
Array Textures 160
Writing to Textures in Shaders 165
Synchronizing Access to Images 176
Texture Compression 177
Texture Views 181
Summary 185

6 Shaders and Programs **187**
Language Overview 188
Data Types 188
Built-In Functions 194
Compiling, Linking, and Examining Programs 201
Getting Information from the Compiler 201
Getting Information from the Linker 204
Separate Programs 206
Shader Subroutines 213
Program Binaries 216
Summary 219

II In Depth 221

7 Vertex Processing and Drawing Commands 223
Vertex Processing . . . 224
Vertex Shader Inputs . . . 224
Vertex Shader Outputs . . . 229
Drawing Commands . . . 231
Indexed Drawing Commands . . . 231
Instancing . . . 237
Indirect Draws . . . 250
Storing Transformed Vertices . . . 259
Using Transform Feedback . . . 260
Starting, Pausing, and Stopping Transform Feedback . . . 264
Ending the Pipeline with Transform Feedback . . . 266
Transform Feedback Example — Physical Simulation . . . 266
Clipping . . . 276
User-Defined Clipping . . . 279
Summary . . . 282

8 Primitive Processing 283
Tessellation . . . 284
Tessellation Primitive Modes . . . 285
Tessellation Subdivision Modes . . . 294
Passing Data between Tessellation Shaders . . . 296
Communication between Shader Invocations . . . 299
Tessellation Example — Terrain Rendering . . . 300
Tessellation Example — Cubic Bézier Patches . . . 304
Geometry Shaders . . . 310
The Pass-Through Geometry Shader . . . 311
Using Geometry Shaders in an Application . . . 313
Discarding Geometry in the Geometry Shader . . . 317
Modifying Geometry in the Geometry Shader . . . 320
Generating Geometry in the Geometry Shader . . . 322
Changing the Primitive Type in the Geometry Shader . . . 325
Multiple Streams of Storage . . . 328
New Primitive Types Introduced by the Geometry Shader . . . 329
Multiple Viewport Transformations . . . 336
Summary . . . 340

9 Fragment Processing and the Framebuffer 341
Fragment Shaders . . . 342
Interpolation and Storage Qualifiers . . . 342
Per-Fragment Tests . . . 345
Scissor Testing . . . 345
Stencil Testing . . . 348
Depth Testing . . . 351
Early Testing . . . 355
Color Output . . . 357
Blending . . . 357
Logical Operations . . . 362
Color Masking . . . 363
Off-Screen Rendering . . . 364
Multiple Framebuffer Attachments . . . 368
Layered Rendering . . . 370
Framebuffer Completeness . . . 376
Rendering in Stereo . . . 379
Antialiasing . . . 384
Antialiasing by Filtering . . . 385
Multi-sample Antialiasing . . . 387
Multi-sample Textures . . . 389
Sample Rate Shading . . . 393
Centroid Sampling . . . 395
Advanced Framebuffer Formats . . . 399
Rendering with No Attachments . . . 399
Floating-Point Framebuffers . . . 401
Integer Framebuffers . . . 415
The sRGB Color Space . . . 416
Point Sprites . . . 419
Texturing Points . . . 420
Rendering a Star Field . . . 420
Point Parameters . . . 423
Shaped Points . . . 424
Rotating Points . . . 426
Getting at Your Image . . . 428
Reading from a Framebuffer . . . 429
Copying Data between Framebuffers . . . 431
Reading Back Texture Data . . . 434
Summary . . . 435

10 Compute Shaders 437
Using Compute Shaders . . . 438

Executing Compute Shaders . . . 439
Compute Shader Communication . . . 444
Examples . . . 449
Compute Shader Parallel Prefix Sum . . . 450
Compute Shader Flocking . . . 462
Summary . . . 471

11 Controlling and Monitoring the Pipeline 473
Queries . . . 474
Occlusion Queries . . . 475
Timer Queries . . . 484
Transform Feedback Queries . . . 487
Synchronization in OpenGL . . . 493
Draining the Pipeline . . . 493
Synchronization and Fences . . . 494
Summary . . . 498

III In Practice 501

12 Rendering Techniques 503
Lighting Models . . . 504
The Phong Lighting Model . . . 504
Blinn-Phong Lighting . . . 513
Rim Lighting . . . 515
Normal Mapping . . . 518
Environment Mapping . . . 522
Material Properties . . . 532
Casting Shadows . . . 534
Atmospheric Effects . . . 540
Non-Photo-Realistic Rendering . . . 544
Cell Shading — Texels as Light . . . 545
Alternative Rendering Methods . . . 548
Deferred Shading . . . 548
Screen-Space Techniques . . . 558
Rendering without Triangles . . . 565
Summary . . . 580

13 Debugging and Performance Optimization 581
Debugging Your Applications . . . 582
Debug Contexts . . . 582

Performance Optimization 589
Performance Analysis Tools 589
Tuning Your Application for Speed 597
Summary 616

14 Platform Specifics **617**
Using Extensions in OpenGL 618
Enhancing OpenGL with Extensions 619
OpenGL on Windows 623
OpenGL Implementations on Windows 623
Basic Window Setup 627
The OpenGL Rendering Context 632
Full-Screen Rendering 644
Cleaning Up 646
OpenGL on Mac OS X 647
The Faces of OpenGL on the Mac 648
OpenGL with Cocoa 649
Introducing GLKit 662
Retina Displays 673
Core OpenGL 674
Full-Screen Rendering 675
Sync Frame Rate 677
Multi-threaded OpenGL 679
GLUT 680
OpenGL on Linux 682
The Basics 682
Brief History 682
What Is X? 683
Getting Started 683
Building OpenGL Apps 687
Windows and Render Surfaces 693
GLX Strings 695
Context Management 695
Using Contexts 699
Putting It All Together 701
Going Full Screen on X 704
OpenGL on Mobile Platforms 705
OpenGL on a Diet 705
OpenGL ES 3.0 709
The OpenGL ES Environment 713
EGL: A New Windowing Environment 718
More EGL 727

Negotiating Embedded Environments 728
Android Development Environments 729
iOpenGL 734
Summary 744

A Further Reading **747**

B The SBM File Format **751**

C The SuperBible Tools **759**

Glossary **765**
Index **773**

Figures

Figure 1.1 Simplified graphics pipeline 6
Figure 1.2 Future Crew's 1992 demo *Unreal* 8

Figure 2.1 The output of our first OpenGL application 15
Figure 2.2 Rendering our first point 23
Figure 2.3 Making our first point bigger 23
Figure 2.4 Our very first OpenGL triangle 25

Figure 3.1 Our first tessellated triangle 36
Figure 3.2 Tessellated triangle after adding a geometry shader . . 38
Figure 3.3 Clockwise (left) and counterclockwise (right) winding order 41
Figure 3.4 Result of Listing 3.10 43
Figure 3.5 Result of Listing 3.12 45

Figure 4.1 A point in space is both a vertex and a vector 52
Figure 4.2 The dot product — cosine of the angle between two vectors 55
Figure 4.3 A cross product returns a vector perpendicular to its parameters 56
Figure 4.4 Reflection and refraction 58
Figure 4.5 A 4×4 matrix representing rotation and translation 62

Figure 4.6 Modeling transformations: rotation then translation, and translation then rotation 63
Figure 4.7 Two perspectives of view coordinates 65
Figure 4.8 The modeling transformations 67
Figure 4.9 A cube translated ten units in the positive *y* direction 69
Figure 4.10 A cube rotated about an arbitrary axis 71
Figure 4.11 A non-uniform scaling of a cube 74
Figure 4.12 A side-by-side example of an orthographic versus perspective projection 81
Figure 4.13 Finding a point on a line 83
Figure 4.14 A simple Bézier curve 84
Figure 4.15 A cubic Bézier curve 85
Figure 4.16 A cubic Bézier spline 88

Figure 5.1 Binding buffers and uniform blocks to binding points 118
Figure 5.2 A few frames from the spinning cube application . . . 124
Figure 5.3 Many cubes! 125
Figure 5.4 A simple textured triangle 142
Figure 5.5 A full-screen texture loaded from a `.KTX` file 146
Figure 5.6 An object wrapped in simple textures 148
Figure 5.7 Texture filtering — nearest (left) and linear (right) . . 153
Figure 5.8 A series of mipmapped images 155
Figure 5.9 A tunnel rendered with three textures and mipmapping 158
Figure 5.10 Example of texture coordinate wrapping modes 160
Figure 5.11 Output of the alien rain sample 165
Figure 5.12 Resolved per-fragment linked lists 177

Figure 6.1 Shape of a Hermite curve 198

Figure 7.1 Indices used in an indexed draw 232
Figure 7.2 Base vertex used in an indexed draw 235
Figure 7.3 Triangle strips with and without primitive restart . . . 237
Figure 7.4 First attempt at an instanced field of grass 241
Figure 7.5 Slightly perturbed blades of grass 242
Figure 7.6 Control over the length and orientation of our grass 243
Figure 7.7 The final field of grass 244
Figure 7.8 Result of instanced rendering 249
Figure 7.9 Result of asteroid rendering program 258
Figure 7.10 Relationship of transform feedback binding points . . 262
Figure 7.11 Connections of vertices in the spring-mass system . . 267

Figure 7.12 Simulation of points connected by springs 273
Figure 7.13 Visualizing springs in the spring-mass system 275
Figure 7.14 Clipping lines 276
Figure 7.15 Clipping triangles 277
Figure 7.16 Clipping triangles using a guard band 278
Figure 7.17 Rendering with user clip distances 282

Figure 8.1 Schematic of OpenGL tessellation 285
Figure 8.2 Tessellation factors for quad tessellation 286
Figure 8.3 Quad tessellation example 286
Figure 8.4 Tessellation factors for triangle tessellation 288
Figure 8.5 Triangle tessellation example 289
Figure 8.6 Tessellation factors for isoline tessellation 290
Figure 8.7 Isoline tessellation example 292
Figure 8.8 Tessellated isoline spirals example 293
Figure 8.9 Triangle tessellated using point mode 294
Figure 8.10 Tessellation using different subdivision modes 295
Figure 8.11 Displacement map used in terrain sample 300
Figure 8.12 Terrain rendered using tessellation 305
Figure 8.13 Tessellated terrain in wireframe 305
Figure 8.14 Final rendering of a cubic Bézier patch 309
Figure 8.15 A Bézier patch and its control cage 310
Figure 8.16 Geometry culled from different viewpoints 320
Figure 8.17 Exploding a model using the geometry shader 322
Figure 8.18 Basic tessellation using the geometry shader 325
Figure 8.19 Displaying the normals of a model using a geometry shader 328
Figure 8.20 Lines produced using lines with adjacency primitives 331
Figure 8.21 Triangles produced using `GL_TRIANGLES_ADJACENCY` 331
Figure 8.22 Triangles produced using `GL_TRIANGLE_STRIP_ADJACENCY` 332
Figure 8.23 Ordering of vertices for `GL_TRIANGLE_STRIP_ADJACENCY` 332
Figure 8.24 Rendering a quad using a pair of triangles 333
Figure 8.25 Parameterization of a quad 334
Figure 8.26 Quad rendered using a geometry shader 337
Figure 8.27 Result of rendering to multiple viewports 339

Figure 9.1 Contrasting perspective-correct and linear interpolation 345
Figure 9.2 Rendering with four different scissor rectangles 347

Figure 9.3 Effect of depth clamping at the near plane 354
Figure 9.4 A clipped object with and without depth clamping . . 355
Figure 9.5 All possible combinations of blending functions . . . 360
Figure 9.6 Result of rendering into a texture 369
Figure 9.7 Result of the layered rendering example 374
Figure 9.8 Result of stereo rendering to a stereo display 384
Figure 9.9 Antialiasing using line smoothing 385
Figure 9.10 Antialiasing using polygon smoothing 386
Figure 9.11 Antialiasing sample positions 387
Figure 9.12 No antialiasing (left) and 8-sample antialiasing (center and right) 388
Figure 9.13 Antialiasing of high-frequency shader output 394
Figure 9.14 Partially covered multi-sampled pixels 396
Figure 9.15 Different views of an HDR image 404
Figure 9.16 Histogram of levels for treelights.ktx 405
Figure 9.17 Naïve tone mapping by clamping 406
Figure 9.18 Transfer curve for adaptive tone mapping 407
Figure 9.19 Result of adaptive tone mapping program 409
Figure 9.20 The effect of light bloom on an image 409
Figure 9.21 Original and thresholded output for bloom example 412
Figure 9.22 Blurred thresholded bloom colors 413
Figure 9.23 Result of the bloom program 414
Figure 9.24 Gamma curves for sRGB and simple powers 418
Figure 9.25 A particle effect in the flurry screen saver 419
Figure 9.26 The star texture map 421
Figure 9.27 Flying through space with point sprites 423
Figure 9.28 Two potential orientations of textures on a point sprite 424
Figure 9.29 Analytically generated point sprite shapes 425

Figure 10.1 Global and local compute work group dimensions . . 443
Figure 10.2 Effect of race conditions in a compute shader 448
Figure 10.3 Effect of `barrier()` on race conditions 449
Figure 10.4 Sample input and output of a prefix sum operation . . 450
Figure 10.5 Breaking a prefix sum into smaller chunks 452
Figure 10.6 A 2D prefix sum 454
Figure 10.7 Computing the sum of a rectangle in a summed area table 456
Figure 10.8 Variable filtering applied to an image 457
Figure 10.9 Depth of field in a photograph 458
Figure 10.10 Applying depth of field to an image 461
Figure 10.11 Effects achievable with depth of field 461

Figure 10.12 Stages in the iterative flocking algorithm 463
Figure 10.13 Output of compute shader flocking program 471

Figure 12.1 Vectors used in Phong lighting 506
Figure 12.2 Per-vertex lighting (Gouraud shading) 509
Figure 12.3 Per-fragment lighting (Phong shading) 510
Figure 12.4 Varying specular parameters of a material 513
Figure 12.5 Phong lighting (left) vs. Blinn-Phong lighting (right) 515
Figure 12.6 Rim lighting vectors 516
Figure 12.7 Result of rim lighting example 517
Figure 12.8 Example normal map 518
Figure 12.9 Result of normal mapping example 522
Figure 12.10 A selection of spherical environment maps 523
Figure 12.11 Result of rendering with spherical environment mapping 525
Figure 12.12 Example equirectangular environment map 526
Figure 12.13 Rendering result of equirectangular environment map 527
Figure 12.14 The layout of six cube faces in the Cubemap sample program 528
Figure 12.15 Cube map environment rendering with a sky box . . 532
Figure 12.16 Pre-filtered environment maps and gloss map 533
Figure 12.17 Result of per-pixel gloss example 535
Figure 12.18 Depth as seen from a light 537
Figure 12.19 Results of rendering with shadow maps 540
Figure 12.20 Graphs of exponential decay 543
Figure 12.21 Applying fog to tessellated landscape 544
Figure 12.22 A one-dimensional color lookup table 545
Figure 12.23 A toon-shaded torus 547
Figure 12.24 Visualizing components of a G-buffer 553
Figure 12.25 Final rendering using deferred shading 554
Figure 12.26 Deferred shading with and without normal maps . . . 556
Figure 12.27 Bumpy surface occluding points 559
Figure 12.28 Selection of random vector in an oriented hemisphere 561
Figure 12.29 Effect of increasing direction count on ambient occlusion 562
Figure 12.30 Effect of introducing noise in ambient occlusion . . . 562
Figure 12.31 Ambient occlusion applied to a rendered scene 563
Figure 12.32 A few frames from the Julia set animation 568
Figure 12.33 Simplified 2D illustration of ray tracing 570
Figure 12.34 Our first ray-traced sphere 573
Figure 12.35 Our first lit ray-traced sphere 574

Figure 12.36 Implementing a stack using framebuffer objects 575
Figure 12.37 Ray-traced spheres with increasing ray bounces 576
Figure 12.38 Adding a ray-traced plane 578
Figure 12.39 Ray-traced spheres in a box 579

Figure 13.1 GPUView in action 591
Figure 13.2 VSync seen in GPUView 592
Figure 13.3 A packet dialog in GPUVIew 593
Figure 13.4 GPU PerfStudio 2 running the displacement mapping example 594
Figure 13.5 GPU PerfStudio 2 frame debugger 595
Figure 13.6 GPU PerfStudio 2 HUD control window 596
Figure 13.7 GPU PerfStudio 2 overlaying information 596
Figure 13.8 GPU PerfStudio 2 showing AMD performance counters 597
Figure 13.9 GPUView showing the effect of **`glReadPixels()`** into system memory 599
Figure 13.10 GPUView showing the effect of **`glReadPixels()`** into a buffer 600

Figure 14.1 Realtech VR's OpenGL Extensions Viewer 619
Figure 14.2 AMD and NVIDIA OpenGL drivers 625
Figure 14.3 The OpenGL Extensions Viewer is free on the Mac App Store 650
Figure 14.4 The initial `CocoaGL` project 651
Figure 14.5 Interface Builder is ready to build your OpenGL app . 651
Figure 14.6 The OpenGL window ready to go... or is it? 652
Figure 14.7 Creating the basic NSView view class 653
Figure 14.8 Turn off the One Shot memory attribute 659
Figure 14.9 This chapter's demo rendering in a Cocoa view 664
Figure 14.10 The Cocoa sample with the supporting files 670
Figure 14.11 Tearing caused by an unsynced buffer swap 678
Figure 14.12 Here's looking at you! 703
Figure 14.13 OpenGL ES rendering on a cell phone 714
Figure 14.14 A typical embedded system diagram 719
Figure 14.15 StonehengeES rendered on an Android phone 731
Figure 14.16 The Xcode welcome screen 735
Figure 14.17 Selecting an OpenGL-ES-based game (application) template 735
Figure 14.18 The starter OpenGL ES application 736
Figure 14.19 The "dancing cubes" default OpenGL ES code 736

Figure 14.20 The Xcode project with the Stonehenge model code added 739
Figure 14.21 The completed Stonehenge model on an iOS device 743

Figure B.1 Dump of example SBM file 757

This page intentionally left blank

Tables

Table 1.1 OpenGL Versions and Publication Dates 7

Table 4.1 Common Coordinate Spaces Used in 3D Graphics . . . 64

Table 5.1 Buffer Object Usage Models 93
Table 5.2 Basic OpenGL Type Tokens and Their Corresponding C Types 96
Table 5.3 Uniform Parameter Queries via `glGetActiveUniformsiv()` 114
Table 5.4 Atomic Operations on Shader Storage Blocks 130
Table 5.5 Texture Targets and Description 139
Table 5.6 Basic Texture Targets and Sampler Types 142
Table 5.7 Texture Filters, Including Mipmapped Filters 156
Table 5.8 Image Types 166
Table 5.9 Image Data Format Classes 168
Table 5.10 Image Data Format Classes 169
Table 5.11 Atomic Operations on Images 172
Table 5.12 Native OpenGL Texture Compression Formats 178
Table 5.13 Texture View Target Compatibility 183
Table 5.14 Texture View Format Compatibility 184

Table 6.1 Scalar Types in GLSL 188
Table 6.2 Vector and Matrix Types in GLSL 190

Table 7.1 Vertex Attribute Types 226
Table 7.2 Draw Type Matrix 232
Table 7.3 Values for primitiveMode 265

Table 8.1 Allowed Draw Modes for Geometry Shader Input Modes 313
Table 8.2 Sizes of Input Arrays to Geometry Shaders 315

Table 9.1 Stencil Functions 349
Table 9.2 Stencil Operations 350
Table 9.3 Depth Comparison Functions 353
Table 9.4 Blend Functions 359
Table 9.5 Blend Equations 362
Table 9.6 Logic Operations 363
Table 9.7 Framebuffer Completeness Return Values 378
Table 9.8 Floating-Point Texture Formats 402

Table 11.1 Possible Return Values for **`glClientWaitSync()`** 496

Table 13.1 Map Buffer Access Types 601

Table 14.1 Pixel Format Attributes 636
Table 14.2 Buffer Swap Values for `WGL_SWAP_METHOD_ARB` 637
Table 14.3 OpenGL Technologies in OS X 648
Table 14.4 Cocoa Pixel Format Attributes 655
Table 14.5 Read-Only Properties of the `GLKTextureInfo` Class . . . 663
Table 14.6 GLX Config Attribute List 690
Table 14.7 Base OpenGL Versions for OpenGL ES 708
Table 14.8 EGL Config Attribute List 721
Table 14.9 EGL Config Attribute List 723
Table 14.10 Configuration Members and Flags for `GLKView` 738

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 24
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 35
Listing 3.9 Our first geometry shader 37
Listing 3.10 Deriving a fragment's color from its position 43
Listing 3.11 Vertex shader with an output 44
Listing 3.12 Deriving a fragment's color from its position 44
Listing 3.13 Simple do-nothing compute shader 47

Listing 5.1 Generating, binding, and initializing a buffer 94
Listing 5.2 Updating the content of a buffer with **glBufferSubData()** 94
Listing 5.3 Mapping a buffer's data store with **glMapBuffer()** . . 95
Listing 5.4 Setting up a vertex attribute 99
Listing 5.5 Using an attribute in a vertex shader 99
Listing 5.6 Declaring two inputs to a vertex shader 100
Listing 5.7 Multiple separate vertex attributes 101
Listing 5.8 Multiple interleaved vertex attributes 102
Listing 5.9 Example uniform block declaration 109
Listing 5.10 Declaring a uniform block with the **std140** layout . . 110
Listing 5.11 Example of a uniform block with offsets 111
Listing 5.12 Retrieving the indices of uniform block members . . 112
Listing 5.13 Retrieving the information about uniform block members 113
Listing 5.14 Setting a single float in a uniform block 114
Listing 5.15 Retrieving the indices of uniform block members . . 115
Listing 5.16 Specifying the data for an array in a uniform block 115
Listing 5.17 Setting up a matrix in a uniform block 116
Listing 5.18 Specifying bindings for uniform blocks 119
Listing 5.19 Uniform blocks binding layout qualifiers 119
Listing 5.20 Setting up cube geometry 121
Listing 5.21 Building the model-view matrix for a spinning cube 122
Listing 5.22 Updating the projection matrix for the spinning cube 123
Listing 5.23 Rendering loop for the spinning cube 123
Listing 5.24 Spinning cube vertex shader 123
Listing 5.25 Spinning cube fragment shader 124
Listing 5.26 Rendering loop for the spinning cube 125
Listing 5.27 Example shader storage block declaration 126
Listing 5.28 Using a shader storage block in place of vertex attributes 127
Listing 5.29 Setting up an atomic counter buffer 134
Listing 5.30 Setting up an atomic counter buffer 134
Listing 5.31 Counting area using an atomic counter 135
Listing 5.32 Using the result of an atomic counter in a uniform block 136
Listing 5.33 Generating, binding, and initializing a texture 138
Listing 5.34 Updating texture data with **glTexSubImage2D()** 138
Listing 5.35 Reading from a texture in GLSL 141
Listing 5.36 The header of a .KTX file 144
Listing 5.37 Loading a .KTX file 145

Listing 5.38 Vertex shader with single texture coordinate 147
Listing 5.39 Fragment shader with single texture coordinate . . . 147
Listing 5.40 Initializing an array texture 161
Listing 5.41 Vertex shader for the alien rain sample 162
Listing 5.42 Fragment shader for the alien rain sample 163
Listing 5.43 Rendering loop for the alien rain sample 164
Listing 5.44 Fragment shader performing image loads and stores 171
Listing 5.45 Filling a linked list in a fragment shader 174
Listing 5.46 Traversing a linked list in a fragment shader 175

Listing 6.1 Retrieving the compiler log from a shader 202
Listing 6.2 Fragment shader with external function declaration 206
Listing 6.3 Configuring a separable program pipeline 208
Listing 6.4 Printing interface information 212
Listing 6.5 Example subroutine uniform declaration 213
Listing 6.6 Setting values of subroutine uniforms 216
Listing 6.7 Retrieving a program binary 217

Listing 7.1 Declaration of a Multiple Vertex Attributes 225
Listing 7.2 Setting up indexed cube geometry 233
Listing 7.3 Drawing indexed cube geometry 234
Listing 7.4 Drawing the same geometry many times 238
Listing 7.5 Pseudo-code for **`glDrawArraysInstanced()`** 240
Listing 7.6 Pseudo-code for **`glDrawElementsInstanced()`** 240
Listing 7.7 Simple vertex shader with per-vertex color 246
Listing 7.8 Simple instanced vertex shader 247
Listing 7.9 Getting ready for instanced rendering 248
Listing 7.10 Example use of an indirect draw command 253
Listing 7.11 Setting up the indirect draw buffer for asteroids . . . 254
Listing 7.12 Vertex shader inputs for asteroids 255
Listing 7.13 Per-indirect draw attribute setup 255
Listing 7.14 Asteroid field vertex shader 255
Listing 7.15 Drawing asteroids 257
Listing 7.16 Spring-mass system vertex setup 268
Listing 7.17 Spring-mass system vertex shader 271
Listing 7.18 Spring-mass system iteration loop 274
Listing 7.19 Spring-mass system rendering loop 274
Listing 7.20 Clipping an object against a plane and a sphere . . . 281

Listing 8.1 Simple quad tessellation control shader example . . . 287
Listing 8.2 Simple quad tessellation evaluation shader example . . . 287
Listing 8.3 Simple triangle tessellation control shader example . . . 289
Listing 8.4 Simple triangle tessellation evaluation shader example . . . 290
Listing 8.5 Simple isoline tessellation control shader example . . . 291
Listing 8.6 Simple isoline tessellation evaluation shader example . . . 291
Listing 8.7 Isoline spirals tessellation evaluation shader . . . 292
Listing 8.8 Vertex shader for terrain rendering . . . 301
Listing 8.9 Tessellation control shader for terrain rendering . . . 302
Listing 8.10 Tessellation evaluation shader for terrain rendering . . . 303
Listing 8.11 Fragment shader for terrain rendering . . . 304
Listing 8.12 Cubic Bézier patch vertex shader . . . 306
Listing 8.13 Cubic Bézier patch tessellation control shader . . . 307
Listing 8.14 Cubic Bézier patch tessellation evaluation shader . . . 307
Listing 8.15 Cubic Bézier patch fragment shader . . . 309
Listing 8.16 Source code for a simple geometry shader . . . 311
Listing 8.17 Geometry shader layout qualifiers . . . 311
Listing 8.18 Iterating over the elements of `gl_in[]` . . . 312
Listing 8.19 The definition of `gl_in[]` . . . 314
Listing 8.20 Configuring the custom culling geometry shader . . . 318
Listing 8.21 Finding a face normal in a geometry shader . . . 318
Listing 8.22 Conditionally emitting geometry in a geometry shader . . . 319
Listing 8.23 Setting up the “explode” geometry shader . . . 321
Listing 8.24 Pushing a face out along its normal . . . 321
Listing 8.25 Pass-through vertex shader . . . 323
Listing 8.26 Setting up the “tessellator” geometry shader . . . 323
Listing 8.27 Generating new vertices in a geometry shader . . . 323
Listing 8.28 Emitting a single triangle from a geometry shader . . . 324
Listing 8.29 Using a function to produce faces in a geometry shader . . . 324
Listing 8.30 A pass-through vertex shader that includes normals . . . 326
Listing 8.31 Setting up the “normal visualizer” geometry shader . . . 326
Listing 8.32 Producing lines from normals in the geometry shader . . . 327

Listing 8.33 Drawing a face normal in the geometry shader 327
Listing 8.34 Geometry shader for rendering quads 335
Listing 8.35 Fragment shader for rendering quads 336
Listing 8.36 Rendering to multiple viewports in a geometry shader 338

Listing 9.1 Setting up scissor rectangle arrays 346
Listing 9.2 Example stencil buffer usage, border decorations . . . 350
Listing 9.3 Rendering with all blending functions 359
Listing 9.4 Setting up a simple framebuffer object 367
Listing 9.5 Rendering to a texture 367
Listing 9.6 Setting up an FBO with multiple attachments 369
Listing 9.7 Declaring multiple outputs in a fragment shader . . . 370
Listing 9.8 Setting up a layered framebuffer 371
Listing 9.9 Layered rendering using a geometry shader 372
Listing 9.10 Displaying an array texture — vertex shader 373
Listing 9.11 Displaying an array texture — fragment shader 373
Listing 9.12 Attaching texture layers to a framebuffer 375
Listing 9.13 Checking completeness of a framebuffer object 378
Listing 9.14 Creating a stereo window 380
Listing 9.15 Drawing into a stereo window 381
Listing 9.16 Rendering to two layers with a geometry shader . . . 382
Listing 9.17 Copying from an array texture to a stereo back buffer 383
Listing 9.18 Turning on line smoothing 386
Listing 9.19 Choosing 8-sample antialiasing 388
Listing 9.20 Setting up a multi-sample framebuffer attachment . . 390
Listing 9.21 Simple multi-sample “maximum” resolve 391
Listing 9.22 Fragment shader producing high-frequency output 393
Listing 9.23 A 100-megapixel virtual framebuffer 401
Listing 9.24 Applying simple exposure coefficient to an HDR image 406
Listing 9.25 Adaptive HDR to LDR conversion fragment shader 407
Listing 9.26 Bloom fragment shader; output bright data to a separate buffer 410
Listing 9.27 Blur fragment shader 412
Listing 9.28 Adding bloom effect to scene 414
Listing 9.29 Creating integer framebuffer attachments 415
Listing 9.30 Texturing a point sprite in the fragment shader 420
Listing 9.31 Vertex shader for the star field effect 422

Listing 9.32 Fragment shader for the star field effect 423
Listing 9.33 Fragment shader for generating shaped points 425
Listing 9.34 Naïve rotated point sprite fragment shader 427
Listing 9.35 Rotated point sprite vertex shader 427
Listing 9.36 Rotated point sprite fragment shader 427
Listing 9.37 Taking a screenshot with **`glReadPixels()`** 430

Listing 10.1 Creating and compiling a compute shader 438
Listing 10.2 Compute shader image inversion 444
Listing 10.3 Dispatching the image copy compute shader 444
Listing 10.4 Compute shader with race conditions 447
Listing 10.5 Simple prefix sum implementation in C++ 450
Listing 10.6 Prefix sum implementation using a compute shader 453
Listing 10.7 Compute shader to generate a 2D prefix sum 455
Listing 10.8 Depth of field using summed area tables 459
Listing 10.9 Initializing shader storage buffers for flocking 464
Listing 10.10 The rendering loop for the flocking example 465
Listing 10.11 Compute shader for updates in flocking example . . . 466
Listing 10.12 The first rule of flocking 467
Listing 10.13 The second rule of flocking 467
Listing 10.14 Main body of the flocking update compute shader . . 468
Listing 10.15 Inputs to the flock rendering vertex shader 469
Listing 10.16 Flocking vertex shader body 470

Listing 11.1 Getting the result from a query object 478
Listing 11.2 Figuring out if occlusion query results are ready . . . 478
Listing 11.3 Simple, application-side conditional rendering 479
Listing 11.4 Rendering when query results aren't available 480
Listing 11.5 Basic conditional rendering example 481
Listing 11.6 A more complete conditional rendering example . . . 482
Listing 11.7 Timing operations using timer queries 484
Listing 11.8 Timing operations using **`glQueryCounter()`** 485
Listing 11.9 Drawing data written to a transform feedback buffer 491
Listing 11.10 Working while waiting for a sync object 495

Listing 12.1 The Gouraud shading vertex shader 507
Listing 12.2 The Gouraud shading fragment shader 508
Listing 12.3 The Phong shading vertex shader 510
Listing 12.4 The Phong shading fragment shader 511

Listing 12.5 Blinn-Phong fragment shader 514
Listing 12.6 Rim lighting shader function 516
Listing 12.7 Vertex shader for normal mapping 520
Listing 12.8 Fragment shader for normal mapping 521
Listing 12.9 Spherical environment mapping vertex shader 523
Listing 12.10 Spherical environment mapping fragment shader . . 524
Listing 12.11 Equirectangular environment mapping fragment shader 526
Listing 12.12 Loading a cube map texture 528
Listing 12.13 Vertex shader for sky box rendering 530
Listing 12.14 Fragment shader for sky box rendering 530
Listing 12.15 Vertex shader for cube map environment rendering 531
Listing 12.16 Fragment shader for cube map environment rendering 531
Listing 12.17 Fragment shader for per-fragment shininess 534
Listing 12.18 Getting ready for shadow mapping 536
Listing 12.19 Setting up matrices for shadow mapping 536
Listing 12.20 Setting up a shadow matrix 538
Listing 12.21 Simplified vertex shader for shadow mapping 538
Listing 12.22 Simplified fragment shader for shadow mapping . . . 539
Listing 12.23 Displacement map tessellation evaluation shader . . 541
Listing 12.24 Application of fog in a fragment shader 543
Listing 12.25 The toon vertex shader 546
Listing 12.26 The toon fragment shader 546
Listing 12.27 Initializing a G-buffer 550
Listing 12.28 Writing to a G-buffer 551
Listing 12.29 Unpacking data from a G-buffer 552
Listing 12.30 Lighting a fragment using data from a G-buffer 553
Listing 12.31 Deferred shading with normal mapping (fragment shader) 555
Listing 12.32 Ambient occlusion fragment shader 564
Listing 12.33 Setting up the Julia set renderer 567
Listing 12.34 Inner loop of the Julia renderer 567
Listing 12.35 Using a gradient texture to color the Julia set 568
Listing 12.36 Ray-sphere intersection test 571
Listing 12.37 Determining closest intersection point 572
Listing 12.38 Ray-plane intersection test 578

Listing 13.1 Creating a debug context with the `sb6` framework . . 582
Listing 13.2 Setting the debug callback function 583

Listing 14.1 Registering a window class 628
Listing 14.2 Creating a simple window 629
Listing 14.3 Declaration of `PIXELFORMATDESCRIPTOR` 631
Listing 14.4 Choosing and setting a pixel format 632
Listing 14.5 Windows main message loop 633
Listing 14.6 Finding a pixel format with **`wglChoosePixelFormatARB()`** 639
Listing 14.7 Enumerating pixel formats on Windows 640
Listing 14.8 Creating shared contexts on Windows 643
Listing 14.9 Setting up a full-screen window 645
Listing 14.10 Definition of the Objective-C `GLCoreProfileView` class 653
Listing 14.11 Initialization of our core context OpenGL view 654
Listing 14.12 Outputting information about the OpenGL context 660
Listing 14.13 Code called whenever the view changes size 660
Listing 14.14 Code called whenever the view changes size 661
Listing 14.15 Controlling movement smoothly with keyboard bit flags and a timer 672
Listing 14.16 Creating and initializing the full-screen window . . . 676
Listing 14.17 GLUT `main` function to set up OpenGL 681
Listing 14.18 Extending `GLSurfaceView` 732
Listing 14.19 Setting up and rendering 733
Listing 14.20 Construction and initialization of the `GLKView` 738
Listing 14.21 Redirecting the current folder to point our resources 742

Foreword

OpenGL® SuperBible has long been an essential reference for 3D graphics developers, and this new edition is more relevant than ever, particularly given the increasing importance of multi-platform deployment. In our line of work, we spend a lot of time at the interface between high-level rendering algorithms and fast-moving GPU and API targets. Even though, between us, we have more than thirty-five years of experience with real-time graphics programming, there is always more to learn. This is why we are so excited about this new edition of the *OpenGL® SuperBible*.

Many programmers of our generation used OpenGL back in the nineties before market forces dictated that we ship Windows games using Direct3D, which first shipped in 1995. While Direct3D initially followed in the footsteps of OpenGL, it eventually surpassed OpenGL in its rapid exposure of advanced GPU functionality, particularly in the transition to programmable graphics hardware.

During this transition, Microsoft consistently shipped new versions of Direct3D for a period of eight years, ending in 2002 with DirectX 9. With DirectX 10, however, Microsoft adopted a release strategy that tied new versions of DirectX to new versions of Windows, not only in terms of timing but in terms of legacy support. That is, not only did new versions of DirectX come out less frequently — only two major versions have come out in the last 11 years — but they were not supported on certain older versions of Windows. Naturally, this change in strategy by Microsoft curtailed the GPU vendors' ability to expose their innovations on Windows.

Fortunately, in this same timeframe, the OpenGL Architecture Review Board accelerated development, putting OpenGL back in a position of

leadership. In fact, there has been so much progress in the past five years that OpenGL has reached a tipping point and is again viable for game development, particularly as more and more developers are adopting a multiplatform strategy that includes OS X and Linux.

OpenGL even has advantages to developers primarily targeting Windows, allowing them to access the very latest GPU features on all Windows versions, not just recent ones that have support for DirectX 10 or DirectX 11. In the growing Asian market, for example, Steam customers have the same caliber of PC hardware as their Western counterparts, but far more of them are running Windows XP, where DirectX 10 and DirectX 11 are not available. An application written using OpenGL, rather than Direct3D, can use the advanced features of customers' hardware and not have to maintain a reduced-quality rendering codepath for customers using Windows XP.

This edition of *OpenGL® SuperBible* is an outstanding resource for a wide variety of software developers, from students who may have some of the math and programming fundamentals but need a nudge in the right direction, to seasoned professional developers who need to quickly find out the nitty-gritty details of a particular API feature. In fact, we suspect that many professionals may be coming back to OpenGL after a number of years away, and this book is an excellent resource for doing just that.

Specifically, this edition of *OpenGL® SuperBible* introduces many of the new features of OpenGL 4.3, such as compute shaders, texture views, indirect multi-draw, enhanced API debugging, and more. As readers of previous editions have come to expect, the SuperBible continues to go well beyond the information provided in the API documentation and into the fundamentals of popular application techniques. Just having all of the essential platform-specific API initialization material for Linux, OS X, and Windows in one place is worth the price of admission, not to mention the detailed discussions of modern debugging techniques, shadow mapping, non-photo-realistic rendering, deferred rendering, and more.

We believe that, for newcomers, OpenGL is the right place to start writing 3D graphics code that will run on a wide array of platforms in order to reach the largest possible audience. Likewise, for professionals, there has never been a better time to come back to OpenGL.

Rich Geldreich and Jason Mitchell
Valve

Preface

About This Book

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 is to ensure that there are as few forward references as possible and to require little or no assumed knowledge. The book should be 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 placed to read and learn from more advanced computer graphics research articles and be confident that you could 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, or to mention every function in the specification or every value that can be passed to a command. Rather, the goal is to provide a solid understanding of OpenGL, introduce its 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 breaks down roughly into three major parts. In the first part, we explain what OpenGL is, how it connects to the graphics pipeline, and 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 in the math behind 3D computer graphics, and describe how OpenGL manages the large amounts of data that are required to provide a compelling experience to the users of your applications. We also describe the programming model for *shaders*, which will form a core part of any OpenGL application.

In the second part of the book, we begin to introduce features of OpenGL that require some knowledge of multiple parts of the graphics pipeline and may refer to concepts already introduced. This allows us to introduce 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 at least been briefly introduced to its destination.

In the final part of the book, 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

- Chapter 1, "Introduction," which provides a brief introduction to OpenGL, its origins, history, and current state.
- Chapter 2, "Our First OpenGL Program," which 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 will be 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 will start to become a little more complex and interesting. In this part, you will find

- Chapter 7, “Vertex Processing and Drawing Commands,” which 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, “Controlling and Monitoring the Pipeline,” shows you how you can get a glimpse of how OpenGL executes the commands you give it — how long they take to execute, and the amount of data that they produce.

In Part III, we build on the knowledge that you will have gained in reading the first two-thirds 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

- Chapter 12, "Rendering Techniques," covers several applications of OpenGL for graphics rendering, from simulation of light to artistic methods and even some non-traditional techniques.
- Chapter 13, "Debugging and Performance Optimization," provides advice and tips on how to get your applications running without errors, and how to get them going fast.
- Chapter 14, "Platform Specifics," covers issues that may be particular to certain platforms, including Windows, Mac, Linux, and mobile devices.

Finally, several appendices are provided that describe the tools and file formats used in this book, and give pointers to more useful OpenGL resources.

What's New in This Edition

This edition of the book differs somewhat from previous editions. This is the sixth edition of the book. The first edition of the book was published in 1996, more than fifteen years ago. Over time, OpenGL has evolved and so has the book's audience. Even since the fifth edition, which was published in 2010, a lot has changed. In some ways, OpenGL has become more complex, with more bells and whistles, more features, and more that you have to do to make something — really anything — show up on the screen. This has raised the barrier to entry for students, and in the fifth edition, we tried to lower that barrier again by glossing over a lot of details or hiding them in utility classes, functions, wrappers, and libraries.

In this edition, we do not hide anything from the reader. What this means is that it might take a while to draw something really impressive, but the extra effort will give you a deeper understanding of what OpenGL is and how it interacts with the underlying graphics hardware. Only the most basic of application frameworks are provided, and our first few programs will be thoroughly underwhelming. However, we're working on the assumption that you'll read the whole book and that by the end of it, you'll have something to show your friends, colleagues, or potential employers that you can be proud of.

In this edition, the printed copy of the OpenGL reference pages, or "man" pages, is gone. The reference pages are available online at `http://www.opengl.org/sdk/docs/man4/` and as a live document are kept up to date. A printed copy of those pages is somewhat redundant and leads to errors — several were found in the reference pages after the fifth edition went to print with no reasonable means of distributing an errata. Further, the reference pages consumed hundreds of printed pages of the book, adding to its cost and size. We'd rather fill a bunch of those pages with more content and save a few trees with the rest.

We've also changed the structure of the book somewhat and make several passes over OpenGL. Rather than having a whole chapter dedicated to a single topic, for example, we introduce as much as possible as early as possible using worked, minimal examples, and then bring in features that touch multiple aspects of OpenGL. This should greatly reduce the number of forward or circular references, and reduce the number of times we need to tell you *don't worry about this, we'll explain it later*.

We hope you enjoy it.

How to Build the Samples

Retrieve the sample code from the book's 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 XP or later is required), Linux (several major distributions), and Mac OS X. 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 4.3 — the most recent version at time of writing. The samples printed in the book are written assuming that OpenGL 4.3 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, and so,

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

- There were several months 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 has those fixes and tweaks applied and therefore deviates 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 sample applications in the Web package. Some sample 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 sample application.

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`

Acknowledgments

First and foremost, I would like to thank my wife, Chris, and my two wonderful kids, Jeremy and Emily. For the never-ending evenings, weekends, and holidays that I spent holed up in my office or curled up with a laptop instead of hanging out with you guys.... I appreciate your patience. I'd like to extend a huge thank you to our tech reviewers, Piers Daniell, Daniel Koch, and Daniel Rákos. You guys did a fantastic job, finding my mistakes and helping to make this book as good as it could be. Your feedback was particularly thorough, and the book grew by at least one hundred pages after I received your reviews. Thanks also to my co-authors Nick Haemel and Richard Wright, Jr. In particular to Richard, thanks for trusting me with taking the lead on this edition. I can only hope that this one turned out as well as the five that preceded it. Thanks to Laura Lewin, Olivia Basegio, Sheri Cain, and the rest of the staff at Addison-Wesley for putting up with me delivering whatever I felt, whenever I felt, and pretty much ignoring schedules and processes. Finally, thanks to you, our readers. Without you, there'd be no book.

Graham Sellers

Thanks to Nick and Graham, my very qualified co-authors. Especially thanks to Graham for taking over the role of lead author for the sixth edition of this book. I fear this revision simply would not have happened without him taking over both the management and the majority of the rewrite for this edition. Two editions ago, Addison-Wesley added this book to its "OpenGL Library" lineup, and I continue to be grateful for that

move years later. For more than fifteen years, countless editors, reviewers, and publishers have made me look good and smarter than I am. There are too many to name, but I have to single out Debra Williams-Cauley for braving more than half this book's lifetime, and, yes, thank you Laura Lewin for taking over for Debra.... You are a brave soul!

Thanks to Full Sail University for letting me teach OpenGL for more than ten years now, while still continuing my "day job." Especially Rob Catto for looking the other way more than once, and running interference when things get in my way on a regular basis. My very good friends and associates in the graphics department there, particularly my department chair, Johnathan Burnside, who simply tolerates my schedule. To Wendy "Kitty" Jones, thanks for all the Thai food! Very special thanks also to my muse, Callisto, for your continuing inspiration and support, not to mention listening to me complain all the time. Special thanks to Software Bisque (Steve, Tom, Daniel, and Matt) for giving me something "real" to do with OpenGL every day, and providing me with possibly the coolest day (and night) job anybody could ever ask for. I also have to thank my family, LeeAnne, Sara, Stephen, and Alex. You've all put up with a lot of mood swings, rapidly changing priorities, and an unpredictable work schedule, and you've provided a good measure of motivation when I really needed it over the years.

Richard S. Wright, Jr.

Thanks to Richard and Graham for collaborating on one more project supporting OpenGL through creating great instructional content. Without your dedication and commitment, computer graphics students would not have the necessary tools to learn 3D graphics. It has been a pleasure working with you over the years to help support 3D graphics and OpenGL specifically. Thanks to Addison-Wesley and Laura Lewin for supporting our project.

I'd also like to thank NVIDIA for the great experiences that have expanded my 3D horizons. It has been great having opportunities to break new ground squeezing OpenGL into incredibly small products. I can't wait to ship all of the exciting things we have been working on! Thanks to Barthold Lichtenbelt for pulling me back into graphics and giving me an opportunity to work on some of the most exciting technology I've seen to date. Thanks to Piers Daniell for your vigilance and help in keeping us all

on track and making sure we get all the details right. Special thanks to Xi Chen at NVIDIA for all your help on Android sample code.

And of course, I couldn't have completed yet another project without the support of my family and friends. To my wife, Anna: You have put up with all of my techno mumbo jumbo all these years while at the same time saving lives and making a significant difference in medicine in your own right. Thanks for your patience and support — I could never be successful without you.

Nicholas Haemel

This page intentionally left blank

About the Authors

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 nineties, and then through Glide, and finally OpenGL in the late nineties. He holds a master's degree in engineering from the University of Southampton, England.

Currently, Graham is a senior manager and software architect on the OpenGL driver team at AMD. He represents AMD at the 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.

Richard S. Wright, Jr., has been using OpenGL for more than eighteen years, since version 1.1, and has taught OpenGL programming in the game design degree program at Full Sail University near Orlando, Florida, for more than a decade. Currently, Richard is a senior engineer at Software Bisque, where he is the technical lead and product manager for a 3D solar

system simulator and their full-dome theater planetarium products, and works on their mobile products and scientific imaging applications.

Previously with Real 3D/Lockheed Martin, Richard was a regular OpenGL ARB attendee and contributed to the OpenGL 1.2 specification and conformance tests back when mammoths still walked the earth. Since then, Richard has worked in multi-dimensional database visualization, game development, medical diagnostic visualization, and astronomical space simulation on Windows, Linux, Mac OS X, and various handheld platforms.

Richard first learned to program in the eighth grade in 1978 on a paper terminal. At age 16, his parents let him buy a computer instead of a car with his grass-cutting money, and he sold his first computer program less than a year later (and it was a graphics program!). When he graduated from high school, his first job was teaching programming and computer literacy for a local consumer education company. He studied electrical engineering and computer science at the University of Louisville's Speed Scientific School and made it halfway through his senior year before his career got the best of him and took him to Florida. A native of Louisville, Kentucky, he now lives in Lake Mary, Florida. When not programming or dodging hurricanes, Richard is an avid amateur astronomer and photography buff. Richard is also, proudly, a Mac.

Nicholas Haemel has been involved with OpenGL for more than fifteen years, since soon after its wide acceptance. He graduated from the Milwaukee School of Engineering with a degree in computer engineering and a love for embedded systems, computer hardware, and making things work. Soon after graduation he put these skills to work for the 3D drivers group at ATI, developing graphics drivers and working on new GPUs.

Nick is now a senior manager of Tegra OpenGL Driver Development at NVIDIA. He leads a team of software developers working on NVIDIA mobile graphics drivers, represents NVIDIA at the Khronos Standards Body, has authored many OpenGL extensions, and contributed to all OpenGL specifications since version 3.0 and to the OpenGL ES 3.0 specification.

Nick's graphics career began at age nine when he first learned to program 2D graphics using Logo Writer. After convincing his parents to purchase a state-of-the-art 286 IBM-compatible PC, it immediately became the central

control unit for robotic arms and other remotely programmable devices. Fast-forward twenty-five years and the devices being controlled are GPUs and SoCs smaller than the size of a fingernail but with more than eight billion transistors. Nick's interests also extend to business leadership and management, bolstered by an MBA from the University of Wisconsin–Madison. Nick currently resides in the Bay Area in California. When not working on accelerating the future of graphics, Nick enjoys the outdoors as a competitive sailor, mountaineer, ex-downhill ski racer, road biker, and photographer.

This page intentionally left blank

Part I

Foundations

This page intentionally left blank

Chapter 1

Introduction

WHAT YOU'LL LEARN IN THIS CHAPTER

- What the graphics pipeline is and how OpenGL relates to it
- The origins of OpenGL and how it came to be the way that it is today
- Some of the fundamental concepts that we'll be building on throughout the book

This book is about OpenGL. OpenGL is an interface that your application can use to access and control the graphics subsystem of the device upon which it runs. This could be anything from a high-end graphics workstation to a commodity desktop computer, a video game console, or even a mobile phone. Standardizing the interface to a subsystem increases portability and allows software developers to concentrate on creating quality products, on producing interesting content, and on the overall performance of their applications, rather than worrying about the specifics of the platforms they want them to run on. These standard interfaces are called Application Programming Interfaces (or APIs), of which OpenGL is one. This chapter introduces OpenGL, describes how it relates to the underlying graphics subsystem, and provides some history on the origin and evolution of OpenGL.

OpenGL and the Graphics Pipeline

Generating a product at high efficiency and volume generally requires two things: scalability and parallelism. In factories, this is achieved by using production lines. While one worker installs the engine in a car, another can be installing the doors and yet another can be installing the wheels. By overlapping the phases of production of the product, with each phase being executed by a skilled technician that concentrates their energy on that single task, each phase becomes more efficient and overall productivity goes up. Also, by making many cars at the same time, a factory can have multiple workers installing multiple engines or wheels or doors, and many cars can be on the production line at the same time, each at different stages of completion.

The same is true in computer graphics. The commands from your program are taken by OpenGL and sent to the underlying graphics hardware, which works on them in an efficient manner to produce the desired result as quickly and efficiently as possible. There could be many commands lined up to execute on the hardware (a term referred to as *in flight*), and some may even be partially completed. This allows their execution to be overlapped such that a later stage of one command might run concurrently with an earlier stage of another command. Furthermore, computer graphics generally consist of many repetitions of very similar tasks (such as figuring out what color a pixel should be), and these tasks are usually independent of one another — that is, the result of coloring one pixel doesn't depend on any other. Just as a car plant can build multiple cars simultaneously, so can OpenGL break up the work you give it and work on its fundamental elements *in parallel*. Through a combination of *pipelining* and *parallelism*, incredible performance of modern graphics processors is realized.

The goal of OpenGL is to provide an *abstraction layer* between your application and the underlying graphics subsystem, which is often a hardware accelerator made up of one or more custom, high performance processors with dedicated memory, display outputs, and so on. This abstraction layer allows your application to not need to know who made the graphics processor (or GPU — graphics processing unit), how it works, or how well it performs. Certainly it is possible to determine this information, but the point is that applications don't need to.

As a design principle, OpenGL must strike a balance between too high and too low an abstraction level. On the one hand, it must hide differences between various manufacturers' products (or between the various products of a single manufacturer) and system-specific traits such as screen

resolution, processor architecture, installed operating system, and so on. On the other hand, the level of abstraction must be low enough that programmers can gain access to the underlying hardware and make best use of it. If OpenGL presented too high of an abstraction level, then it would be easy to create programs that fit the model, but very hard to use advanced features of the graphics hardware that weren't included. This is the type of model followed by software such as game engines — new features of the graphics hardware generally require pretty large changes in the engine in order for games built on top of it to gain access to them. If the abstraction level is too low, applications need to start worrying about architectural peculiarities of the system they're running on. Low levels of abstraction are common in video game consoles, for example, but don't fit well into a graphics library that spans in support from mobile phones through gaming PCs to high power professional graphics workstations.

As technology advances, more and more research is conducted into computer graphics, best practices are developed, and bottlenecks and requirements move, and so OpenGL must move to keep up.

The current state-of-the-art in graphics processing units, which most OpenGL implementations are based on, are capable of many teraflops of computing power, have gigabytes of memory that can be accessed at hundreds of gigabytes per second, and can drive multiple, multi-megapixel displays at high refresh rates. GPUs are also extremely flexible, and are able to work on tasks that might not be considered graphics at all such as physical simulations, artificial intelligence, and even audio processing.

Current GPUs consist of large number of small programmable processors called *shader cores* which run mini-programs called *shaders*. Each core has a relatively low throughput, processing a single instruction of the shader in one or more clock cycles and normally lacking advanced features such as out-of-order execution, branch prediction, super-scalar issue, and so on. However, each GPU might contain anywhere from a few tens to a few thousand of these cores, and together they can perform an immense amount of work. The graphics system is broken into a number *stages*, each represented either by a shader or by a fixed-function, possibly configurable processing block. Figure 1.1 shows a simplified schematic of the graphics pipeline.

In Figure 1.1, the boxes with rounded corners are considered *fixed-function* stages whereas the boxes with square corners are programmable, which means that they execute shaders that you supply. In practice, some or all of the fixed-function stages may really be implemented in shader code too — it's just that you don't supply that code, but rather the GPU

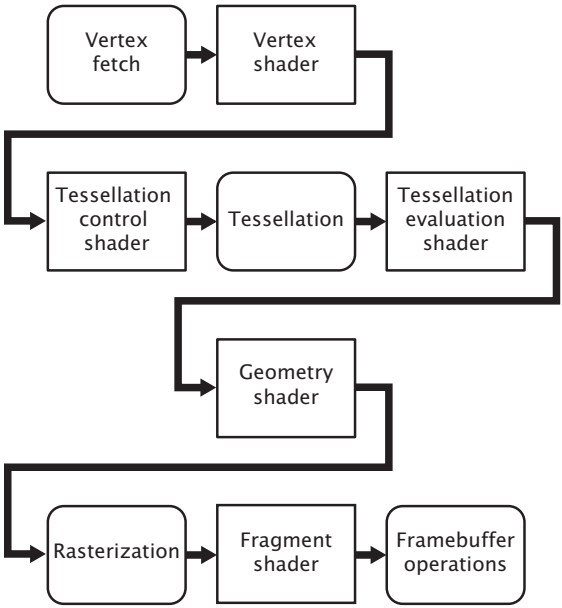


Figure 1.1: Simplified graphics pipeline

manufacturer would generally supply it as part of a driver, firmware, or other system software.

The Origins and Evolution of OpenGL

OpenGL has its origins at Silicon Graphics, Inc., (SGI) and their IRIS GL. GL stood for (and still stands for) "Graphics Library" and in much of the modern OpenGL documentation you will see the term "the GL," meaning "the graphics library," originating from this era. Silicon Graphics was[1] a manufacturer of high-end graphics workstations. These were extremely expensive, and using a proprietary API for graphics wasn't helping. Other manufacturers were producing much more inexpensive solutions running on competing APIs that were often compatible with each other. In the early nineties, SGI realized that portability was important and so decided to clean up IRIS GL, remove system-specific parts of the API and release it as an open standard that could be implemented, royalty free by anyone. The very first version of OpenGL was released in June of 1992 and was marked as OpenGL 1.0.

1. Silicon Graphics, or more accurately SGI, still exists today, but went bankrupt in 2009, with its assets and brands acquired by Rackable Systems, who assumed the moniker SGI, but do not operate in the high-end graphics market.

That year, SGI was also instrumental in establishing the OpenGL Architectural Review Board (ARB), the original members of which included companies such as Compaq, DEC, IBM, Intel, and Microsoft. Soon, other companies such as Hewlett Packard, Sun Microsystems, Evans & Sutherland, and Intergraph joined the group. The OpenGL ARB is the standards body that designs, governs, and produces the OpenGL specification and is now a part of Khronos Group, which is a larger consortium of companies that oversees the development of many open standards. Some of these original members either no longer exist (perhaps having gone out of business or having been acquired by or merged with other companies) or are no longer members of the ARB, having left the graphics business or otherwise gone their own ways. However, some still exist, either under new names or as the entity that was involved in the development of that very first version of OpenGL more than 20 years ago.

At time of writing, there have been 17 editions of the OpenGL specification. Their version numbers and dates of publication are shown in Table 1.1. This book covers version 4.3 of the OpenGL specification.

Table 1.1: OpenGL Versions and Publication Dates

Version	Publication Date
OpenGL 1.0	January 1992
OpenGL 1.1	January 1997
OpenGL 1.2	March 1998
OpenGL 1.2.1	October 1998
OpenGL 1.3	August 2001
OpenGL 1.4	July 2002
OpenGL 1.5	July 2003
OpenGL 2.0	September 2004
OpenGL 2.1	July 2006
OpenGL 3.0	August 2008
OpenGL 3.1	March 2009
OpenGL 3.2	August 2009
OpenGL 3.3	March 2010
OpenGL 4.0	March 2010[2]
OpenGL 4.1	July 2010
OpenGL 4.2	August 2011
OpenGL 4.3	August 2012

2. Yes, two versions at the same time!

Core Profile OpenGL

Twenty years is a long time in the development of cutting edge technology. In 1992, the top-of-the-line Intel CPU was the 80486, math co-processors were still optional, and the Pentium had not yet been invented (or at least released). Apple computers were still using Motorola 68K derived processors, and the PowerPC processors to which they would later switch would be made available during the second half of 1992. High-performance graphics acceleration was simply not something that was common in commodity home computers. If you didn't have access to a high-performance graphics workstation, you probably would have no hope of using OpenGL for anything. Software rendering ruled the world, and the Future Crew's *Unreal* demo won the Assembly '92 demo party. The best you could hope for in a home computer was some basic filled polygons or sprite rendering capabilities. The state of the art in 1992 home computer 3D graphics is shown in Figure 1.2.

Figure 1.2: Future Crew's 1992 demo *Unreal*

Over time, the price of graphics hardware came down, performance went up, and, partly due to low cost acceleration add-in boards for PCs and partly due to the increased performance of video game consoles, new features and capabilities showed up in affordable graphics processors and were added to OpenGL. Most of these features originated in *extensions* proposed by members of the OpenGL ARB. Some interacted well with each other and with existing features in OpenGL, and some did not. Also, as newer, better ways of squeezing performance out of graphics systems were

invented, they were simply added to OpenGL, resulting in it having multiple ways of doing the same thing.

For many years, the ARB held a strong position on backwards compatibility, as it still does today. However, this backwards compatibility comes at a significant cost. Best practices have changed — what may have worked well or was not really a significant bottleneck on mid-1990s graphics hardware doesn't fit modern graphics processor architecture well. Specifying how new features interact with the older legacy features isn't easy and, in many cases, can make it almost impossible to cleanly introduce a new feature to OpenGL. As for implementing OpenGL, this has become such a difficult task that drivers tend to have more bugs than they really should, and graphics vendors need to spend considerable amounts of energy maintaining support for all kinds of legacy features that don't contribute to the advancement of or innovation in graphics.

For these reasons, in 2008, the ARB decided it would "fork" the OpenGL specification into two *profiles*. The first is the modern, *core* profile, which removes a number of legacy features leaving only those that are truly accelerated by current graphics hardware. This specification is several hundred pages shorter[3] than the other version of the specification, the *compatibility* profile. The compatibility profile maintains backwards compatibility with all revisions of OpenGL back to version 1.0. That means that software written in 1992 should compile and run on a modern graphics card with a thousand times higher performance today than when that program was first produced.

However, the compatibility profile really exists to allow software developers to maintain legacy applications and to add features to them without having to tear out years of work in order to shift to a new API. However, the core profile is strongly recommended by most OpenGL experts to be the profile that should be used for new application development. In particular, on some platforms, newer features are only available if you are using the core profile of OpenGL, and on others, an application written using the core profile of OpenGL will run *faster* than that same application unmodified, except to request the compatibility profile, even if it only uses features that are available in core profile OpenGL. Finally, if a feature's in the compatibility profile but has been removed from the core profile of OpenGL, there's probably a good reason for that, and it's a reasonable indication that you shouldn't be using it.

3. The core profile specification is still pretty hefty at well over 700 pages long.

This book covers only the core profile of OpenGL, and this is the last time we will mention the compatibility profile.

Primitives, Pipelines, and Pixels

As discussed, the model followed by OpenGL is that of a production line, or pipeline. Data flow within this model is generally one way, with data formed from commands called by your programs entering the front of the pipeline and flowing from stage to stage until it reaches the end of the pipeline. Along the way, shaders or other fixed-function blocks within the pipeline may pick up more data from *buffers* or *textures*, which are structures designed to store information that will be used during rendering. Some stages in the pipeline may even save data into these buffers or textures, allowing the application to read or save the data, or even for feedback to occur.

The fundamental unit of rendering in OpenGL is known as the *primitive*. OpenGL supports many types of primitives, but the three basic renderable primitive types are points, lines, and triangles. Everything you see rendered on the screen is a collection of (perhaps cleverly colored) points, lines, and triangles. Applications will normally break complex surfaces into a very large number of triangles and send them to OpenGL where they are rendered using a hardware accelerator called a *rasterizer*. Triangles are, relatively speaking, pretty easy to draw. As polygons, triangles are always *convex*, and therefore filling rules are easy to devise and follow. Concave polygons can always be broken down into two or more triangles, and so hardware natively supports rendering triangles directly and relies on other subsystems[4] to break complex geometry into triangles. The rasterizer is dedicated hardware that converts the three-dimensional representation of a triangle into a series of pixels that need to be drawn onto the screen.

Points, lines, and triangles are formed from collections of one, two, or three vertices, respectively. A *vertex* is simply a point within a coordinate space. In our case, we primarily consider a three-dimensional coordinate system. The graphics pipeline is broken down into two major parts. The first part, often known as the *front end*, processes vertices and primitives, eventually forming them into the points, lines, and triangles that will be handed off to the rasterizer. This is known as *primitive assembly*. After the rasterizer, the geometry has been converted from what

4. Sometimes, these subsystems are more hardware modules, and sometimes they are functions of drivers implemented in software.

is essentially a vector representation into a large number of independent pixels. These are handed off to the *back end*, which includes depth and stencil testing, fragment shading, blending, and updating the output image.

As you progress through this book, you will see how to tell OpenGL to start working for you. We'll go over how to create buffers and textures and hook them up to your programs. We'll also see how to write shaders to process your data and how to configure the fixed-function blocks of OpenGL to do what you want. OpenGL is really a large collection of fairly simple concepts, built upon each other. Having a good foundation and *big-picture* view of the system is essential, and over the next few chapters, we hope to provide that to you.

Summary

In this chapter you've been introduced to OpenGL and have read a little about its origins, history, status, and direction. You have seen the OpenGL pipeline and have been told how this book is going to progress. We have mentioned some of the terminology that we'll be using throughout the book. Over the next few chapters, you'll create our first OpenGL program, dig a little deeper into the various stages of the OpenGL pipeline, and then lay some foundations with some of the math that's useful in the world of computer graphics.

This page intentionally left blank

Chapter 2

Our First OpenGL Program

WHAT YOU'LL LEARN IN THIS CHAPTER

- How to create and compile shader code
- How to draw with OpenGL
- How to use the book's application framework to initialize your programs and clean up after yourself

In this chapter, we introduce the simple application framework that is used for almost all of the samples in this book. This shows you how to create the main window with the book's application framework and how to render simple graphics into it. You'll also see what a very simple GLSL shader looks like, how to compile it, and how to use it to render simple points. The chapter concludes with your very first OpenGL triangle.

Creating a Simple Application

To introduce the application framework that'll be used in the remainder of this book, we'll start with an extremely simple example application. The application framework is brought into your application by including `sb6.h` in your source code. This is a C++ header file that defines a namespace called `sb6` that includes the declaration of an application class, `sb6::application`, from which we can derive our examples. The framework also includes a number of utility functions and a simple math library called `vmath` to help you with some of the number crunching involved in OpenGL.

To create an application, we simply include `sb6.h`, derive a class v `sb6::application`, and (in exactly one of our source files) include an instance of the `DECLARE_MAIN` macro. This defines the main entry point of our application, which creates an instance of our class (the type of which is passed as a parameter to the macro) and calls its `run()` method, which implements the application's main loop.

In turn, this performs some initialization by calling the `startup()` method and then calls the `render()` method in a loop. In the default implementation, both methods are virtual functions with empty bodies. We override the `render()` method in our derived class and write our drawing code inside it. The application framework takes care of creating a window, handling input, and displaying the rendered results to the user. The complete source code for our first example is given in Listing 2.1, and its output is shown in Figure 2.1.

```
// Include the "sb6.h" header file
#include "sb6.h"

// Derive my_application from sb6::application
class my_application : public sb6::application
{
public:
    // Our rendering function
    void render(double currentTime)
    {
        // Simply clear the window with red
        static const GLfloat red[] = { 1.0f, 0.0f, 0.0f, 1.0f };
        glClearBufferfv(GL_COLOR, 0, red);
    }
};

// Our one and only instance of DECLARE_MAIN
DECLARE_MAIN(my_application);
```

Listing 2.1: Our first OpenGL application

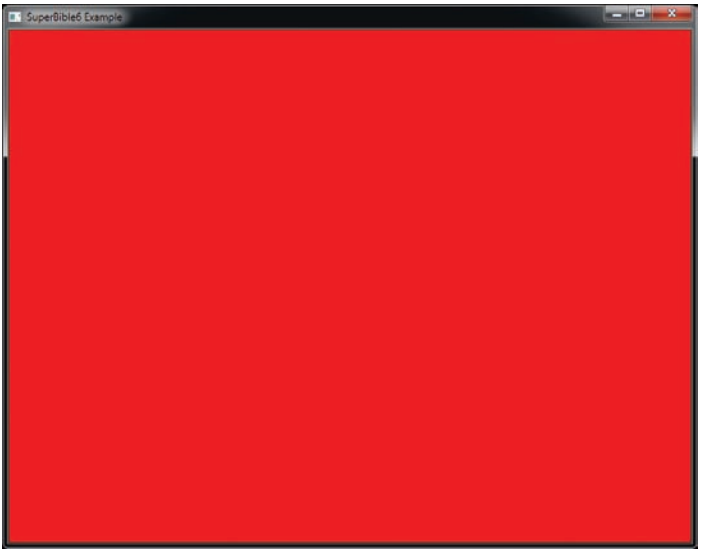


Figure 2.1: The output of our first OpenGL application

The example shown in Listing 2.1 simply clears the whole screen to red. This introduces our first OpenGL function, **`glClearBufferfv()`**. The prototype of **`glClearBufferfv()`** is

```
void glClearBufferfv(GLenum buffer,
                     GLint drawBuffer,
                     const GLfloat * value);
```

All OpenGL functions start with `gl` and follow a number of naming conventions such as encoding some of their parameter types as suffixes on the end of the function names. This allows a limited form of *overloading* even in languages that don't directly support this. In this case, the suffix `fv` means that the function consumes a vector (`v`) of floating-point (`f`) values, where arrays (generally referenced by pointers in languages like C) and vectors are used interchangeably by OpenGL.

The **`glClearBufferfv()`** function tells OpenGL to clear the buffer specified by the first parameter (in this case `GL_COLOR`) to the value specified in its third parameter. The second parameter, `drawBuffer`, is used when there are multiple output buffers that could be cleared. Because we're only using one here and `drawBuffer` is a zero-based index, we'll just set it to zero in this example. Here, that color is stored in the array `red`, which contains four floating-point values — one each for red, green, blue, and alpha, in that order. The red, green, and blue terms should be self-explanatory. Alpha is a

fourth component that is associated with a color and is often used to encode the *opacity* of a fragment. When used this way, setting alpha to zero will make the fragment completely transparent, and setting it to one will make it completely opaque. The alpha value can also be stored in the output image and used in some parts of OpenGL's calculations, even though you can't see it. You can see that we set both the red and alpha values to one and the others to zero. This specifies an opaque red color. The result of running this application is shown in Figure 2.1.

This initial application isn't particularly interesting[1] as all it does is fill the window with a solid red color. You will notice that our `render()` function takes a single parameter — `currentTime`. This contains the number of seconds since the application was started, and we can use it to create a simple animation. In this case, we can use it to change the color that we use to clear the window. Our modified `render()` function[2] is shown in Listing 2.2.

```
// Our rendering function
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);
}
```

Listing 2.2: Animating color over time

Now our window fades from red through yellow, orange, green, and back to red again. Still not that exciting, but at least it does *something*.

Using Shaders

As we mentioned in the introduction to the graphics pipeline in Chapter 1, "Introduction," OpenGL works by connecting a number of mini-programs called shaders together with fixed-function glue. When you draw, the graphics processor executes your shaders and pipes their

1. This sample is especially uninteresting if you are reading this book in black and white!

2. If you're copying this code into your own example, you'll need to include `<math.h>` in order to get the declarations of `sin()` and `cos()`.

inputs and outputs along the pipeline until pixels[3] come out the end. In order to draw anything at all, you'll need to write at least a couple of shaders.

OpenGL shaders are written in a language called the OpenGL Shading Language, or GLSL. This is a language that has its origins in C, but has been modified over time to make it better suited to running on graphics processors. If you are familiar with C, then it shouldn't be hard to pick up GLSL. The compiler for this language is built into OpenGL. The source code for your shader is placed into a *shader object* and compiled, and then multiple shader objects can be linked together to form a *program object*. Each program object can contain shaders for one or more shader stages. The shader stages of OpenGL are vertex shaders, tessellation control and evaluation shaders, geometry shaders, fragment shaders, and compute shaders. The minimal useful pipeline configuration consists only of a vertex shader[4] (or just a compute shader), but if you wish to see any pixels on the screen, you will also need a fragment shader.

Our first couple of shaders are extremely simple. Listing 2.3 shows our first vertex shader. This is about as simple as it gets. In the first line, we have the `#version 430 core` declaration, which tells the shader compiler that we intend to use version 4.3 of the shading language. Notice that we include the keyword `core` to indicate that we only intend to use features from the core profile of OpenGL.

Next, we have the declaration of our `main` function, which is where the shader starts executing. This is exactly the same as in a normal C program, except that the `main` function of a GLSL shader has no parameters. Inside our `main` function, we assign a value to `gl_Position`, which is part of the plumbing that connects the shader to the rest of OpenGL. All variables that start with `gl_` are part of OpenGL and connect shaders to each other or to the various parts of fixed functionality in OpenGL. In the vertex shader, `gl_Position` represents the output position of the vertex. The value we assign (`vec4(0.0, 0.0, 0.5, 1.0)`) places the vertex right in the middle of OpenGL's *clip space*, which is the coordinate system expected by the next stage of the OpenGL pipeline.

3. Actually, there are a number of use cases of OpenGL that create no pixels at all. We will cover those in a while. For now, let's just draw some pictures.

4. If you try to draw anything when your pipeline does not contain a vertex shader, the results will be undefined and almost certainly not what you were hoping for.

```
#version 430 core

void main(void)
{
    gl_Position = vec4(0.0, 0.0, 0.5, 1.0);
}
```

Listing 2.3: Our first vertex shader

Next, our fragment shader is given in Listing 2.4. Again, this is extremely simple. It too starts with a `#version 430 core` declaration. Next, it declares `color` as an output variable using the `out` keyword. In fragment shaders, the value of output variables will be sent to the window or screen. In the `main` function, it assigns a constant to this output. By default, that value goes directly onto the screen and is a vector of four floating-point values, one each for red, green, blue, and alpha, just like in the parameter to **`glClearBufferfv()`**. In this shader, the value we've used is `vec4(0.0, 0.8, 1.0, 1.0)`, which is a cyan color.

```
#version 430 core

out vec4 color;

void main(void)
{
    color = vec4(0.0, 0.8, 1.0, 1.0);
}
```

Listing 2.4: Our first fragment shader

Now that we have both a vertex and a fragment shader, it's time to compile them and link them together into a program that can be run by OpenGL. This is similar to the way that programs written in C++ or other similar languages are compiled and linked to produce executables. The code to link our shaders together into a program object is shown in Listing 2.5.

```
GLuint compile_shaders(void)
{
    GLuint vertex_shader;
    GLuint fragment_shader;
    GLuint program;

    // Source code for vertex shader
    static const GLchar * vertex_shader_source[] =
    {
        "#version 430 core                                  \n"
        "                                                   \n"
        "void main(void)                                    \n"
        "{                                                  \n"
        "    gl_Position = vec4(0.0, 0.0, 0.5, 1.0);        \n"
```

```
        "}                                                      \n"
    };

    // Source code for fragment shader
    static const GLchar * fragment_shader_source[] =
    {
        "#version 430 core                                      \n"
        "                                                       \n"
        "out vec4 color;                                        \n"
        "                                                       \n"
        "void main(void)                                        \n"
        "{                                                      \n"
        "    color = vec4(0.0, 0.8, 1.0, 1.0);                  \n"
        "}                                                      \n"
    };

    // Create and compile vertex shader
    vertex_shader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertex_shader, 1, vertex_shader_source, NULL);
    glCompileShader(vertex_shader);

    // Create and compile fragment shader
    fragment_shader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragment_shader, 1, fragment_shader_source, NULL);
    glCompileShader(fragment_shader);

    // Create program, attach shaders to it, and link it
    program = glCreateProgram();
    glAttachShader(program, vertex_shader);
    glAttachShader(program, fragment_shader);
    glLinkProgram(program);

    // Delete the shaders as the program has them now
    glDeleteShader(vertex_shader);
    glDeleteShader(fragment_shader);

    return program;
}
```

Listing 2.5: Compiling a simple shader

In Listing 2.5, we introduce a handful of new functions:

- **`glCreateShader()`** creates an empty shader object, ready to accept source code and be compiled.
- **`glShaderSource()`** hands shader source code to the shader object so that it can keep a copy of it.
- **`glCompileShader()`** compiles whatever source code is contained in the shader object.
- **`glCreateProgram()`** creates a program object to which you can attach shader objects.
- **`glAttachShader()`** attaches a shader object to a program object.

- **glLinkProgram()** links all of the shader objects attached to a program object together.
- **glDeleteShader()** deletes a shader object. Once a shader has been linked into a program object, the program contains the binary code and the shader is no longer needed.

The shader source code from Listing 2.3 and Listing 2.4 is included in our program as constant strings that are passed to the **glShaderSource()** function, which copies them into the shader objects that we created with **glCreateShader()**. The shader object stores a copy of our source code, and then when we call **glCompileShader()**, it compiles the GLSL shader source code into an intermediate binary representation, which is also stored in the shader object. The program object represents the linked executable that we will use for rendering. We attach our shaders to the program object using **glAttachShader()** and then call **glLinkProgram()**, which links the objects together into code that can be run on the graphics processor. Attaching a shader object to a program object creates a reference to the shader and so we can delete it, knowing that the program object will hold onto the shader's contents as long as it needs it. The `compile_shaders` function in Listing 2.5 returns the newly created program object.

When we call this function, we need to keep the returned program object somewhere so that we can use it to draw things. Also, we really don't want to recompile the whole program every time we want to use it. So, we need a function that is called once when the program starts up. The `sb6` application framework provides just such a function: `application::startup()`, which we can override in our sample application and perform any one-time setup work.

One final thing that we need to do before we can draw anything is to create a *vertex array object* (VAO), which is an object that represents the vertex fetch stage of the OpenGL pipeline and is used to supply input to the vertex shader. As our vertex shader doesn't have any inputs right now, we don't need to do much with the VAO. Nevertheless, we still need to create the VAO so that OpenGL will let us draw. To create the VAO, we call the OpenGL function **glGenVertexArrays()**, and to attach it to our context, we call **glBindVertexArray()**. Their prototypes are

```
void glGenVertexArrays(GLsizei n,
                       GLuint * arrays);

void glBindVertexArray(GLuint array);
```

The vertex array object maintains all of the state related to the input to the OpenGL pipeline. We will add calls to **glGenVertexArrays()** and **glBindVertexArray()** to our `startup()` function.

In Listing 2.6, we have overridden the `startup()` member function of the `sb6::application` class and put our own initialization code in it. Again, as with `render()`, the `startup()` function is defined as an empty virtual function in `sb6::application` and is called automatically by the `run()` function. From `startup()`, we call `compile_shaders` and store the resulting program object in the `rendering_program` member variable in our class. When our application is done running, we should also clean up after ourselves, and so we have also overridden the `shutdown()` function and in it, we delete the program object that we created at start-up. Just as when we were done with our shader objects, we called **glDeleteShader()**, so when we are done with our program objects, we call **glDeleteProgram()**. In our `shutdown()` function, we also delete the vertex array object we created in our `startup()` function.

```
class my_application : public sb6::application
{
public:
    // <snip>

    void startup()
    {
        rendering_program = compile_shaders();
        glGenVertexArrays(1, &vertex_array_object);
        glBindVertexArray(vertex_array_object);
    }

    void shutdown()
    {
        glDeleteVertexArrays(1, &vertex_array_object);
        glDeleteProgram(rendering_program);
        glDeleteVertexArrays(1, &vertex_array_object);
    }

private:
    GLuint  rendering_program;
    GLuint  vertex_array_object;
};
```

Listing 2.6: Creating the program member variable

Now that we have a program, we need to execute the shaders in it and actually get to drawing something on the screen. We modify our `render()` function to call **glUseProgram()** to tell OpenGL to use our program object for rendering and then call our first drawing command, **glDrawArrays()**. The updated listing is shown in Listing 2.7.

```
// Our rendering function
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);

    // Draw one point
    glDrawArrays(GL_POINTS, 0, 1);
}
```

Listing 2.7: Rendering a single point

The **`glDrawArrays()`** function sends vertices into the OpenGL pipeline. Its prototype is

```
void glDrawArrays(GLenum mode,
                  GLint first,
                  GLsizei count);
```

For each vertex, the vertex shader (the one in Listing 2.3) is executed. The first parameter to **`glDrawArrays()`** is the `mode` parameter and tells OpenGL what type of graphics primitive we want to render. In this case, we specified `GL_POINTS` because we want to draw a single point. The second parameter (`first`) is not relevant in this example, and so we've set it to zero. Finally, the last parameter is the number of vertices to render. Each point is represented by a single vertex, and so we tell OpenGL to render only one vertex, resulting in just one point being rendered. The result of running this program is shown in Figure 2.2.

As you can see, there is a tiny point in the middle of the window. For your viewing pleasure, we've zoomed in on the point and shown it in the inset at the bottom right of the image. Congratulations! You've made your very first OpenGL rendering. Although it's not terribly impressive yet, it lays the groundwork for more and more interesting drawing and proves that our application framework and our first, extremely simple shaders are working.

In order to make our point a little more visible, we can ask OpenGL to draw it a little larger than a single pixel. To do this, we'll call the **`glPointSize()`** function, whose prototype is

```
void glPointSize(GLfloat size);
```

This function sets the diameter of the point in pixels to the value you specify in `size`. The maximum value that you can use for points is

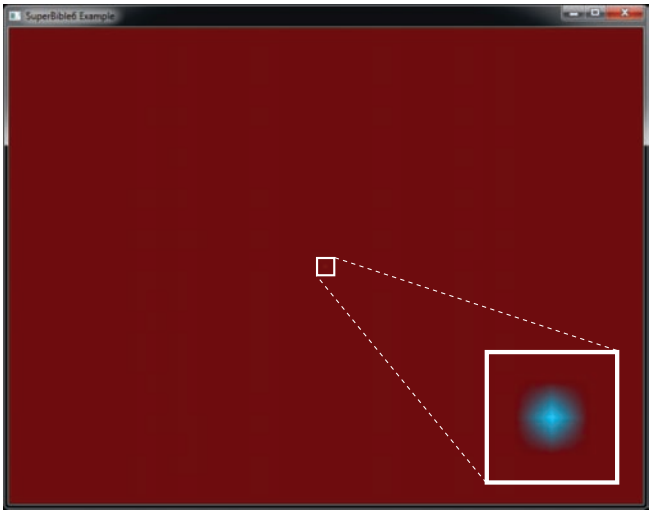


Figure 2.2: Rendering our first point

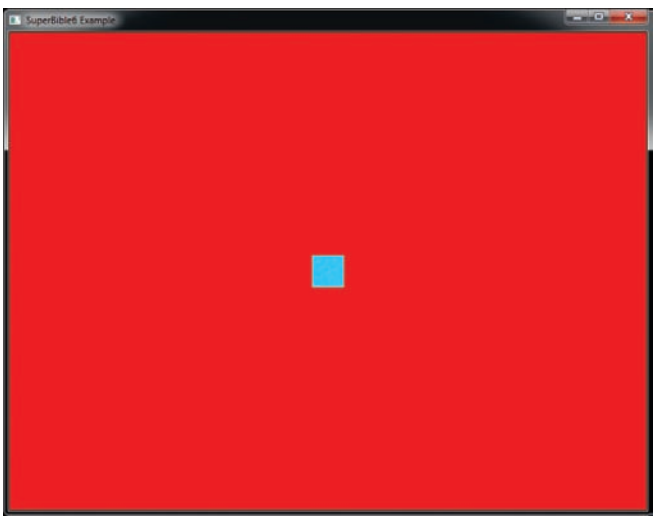


Figure 2.3: Making our first point bigger

implementation defined, but OpenGL guarantees that it's at least 64 pixels. By adding the following line

```
glPointSize(40.0f);
```

to our rendering function in Listing 2.7, we set the diameter of points to 40 pixels, and are presented with the image in Figure 2.3.

Drawing Our First Triangle

Drawing a single point is not really that impressive (even if it is really big!) — we already mentioned that OpenGL supports many different primitive types, and that the most important are points, lines, and triangles. In our toy example, we draw a single point by passing the token `GL_POINTS` to the **`glDrawArrays()`** function. What we really want to do is draw lines or triangles. As you may have guessed, we could also have passed `GL_LINES` or `GL_TRIANGLES` to **`glDrawArrays()`**, but there's one hitch: The vertex shader we showed you in Listing 2.3 places every vertex in the same place, right in the middle of clip space. For points, that's fine as OpenGL assigns area to points for you, but for lines and triangles, having two or more vertices in the exact same place produces a *degenerate primitive*, which is a line with zero length, or a triangle with zero area. If we try to draw anything but points with this shader, we won't get any output at all because all of the primitives will be degenerate. To fix this, we need to modify our vertex shader to assign a different position to each vertex.

Fortunately, GLSL includes a special input to the vertex shader called `gl_VertexID`, which is the index of the vertex that is being processed at the time. The `gl_VertexID` input starts counting from the value given by the `first` parameter of **`glDrawArrays()`** and counts upwards one vertex at a time for `count` vertices (the third parameter of **`glDrawArrays()`**). This input is one of the many *built-in variables* provided by GLSL that represent data that is generated by OpenGL or that you should generate in your shader and give to OpenGL (`gl_Position`, which we just covered, is another example of a built-in variable). We can use this index to assign a different position to each vertex (see Listing 2.8, which does exactly this).

```
#version 430 core

void main(void)
{
    // Declare a hard-coded array of positions
    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));

    // Index into our array using gl_VertexID
    gl_Position = vertices[gl_VertexID];
}
```

Listing 2.8: Producing multiple vertices in a vertex shader

By using the shader of Listing 2.8, we can assign a different position to each of the vertices based on their value of `gl_VertexID`. The points in the

array `vertices` form a triangle, and if we modify our rendering function to pass `GL_TRIANGLES` to **`glDrawArrays()`** instead of `GL_POINTS`, as shown in Listing 2.9, then we obtain the image shown in Figure 2.4.

```
// Our rendering function
void render(double currentTime)
{
    const GLfloat color[] = { 0.0f, 0.2f, 0.0f, 1.0f };
    glClearBufferfv(GL_COLOR, 0, color);

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

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

Listing 2.9: Rendering a single triangle

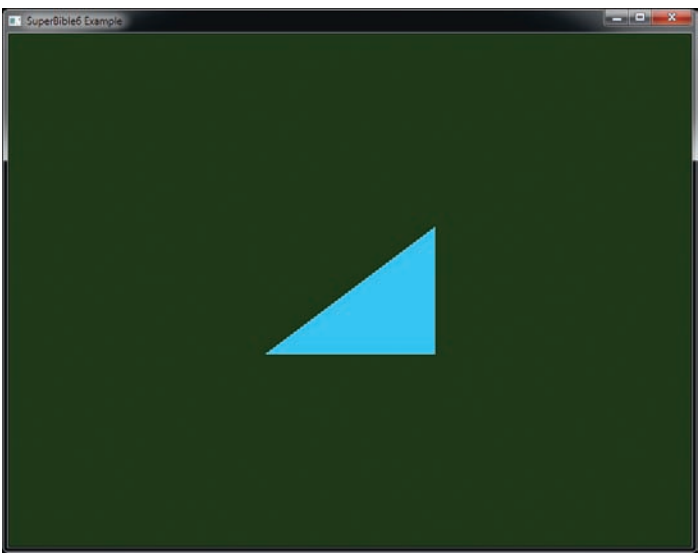


Figure 2.4: Our very first OpenGL triangle

Summary

This concludes the construction of our first OpenGL program. Shortly, we will cover how to get data into your shaders from your application, how to pass your own inputs to the vertex shader, how to pass data from shader stage to shader stage, and more.

In this chapter, you have been briefly introduced to the `sb6` application framework, compiled a shader, cleared the window, and drawn points and triangles. You have seen how to change the size of points using the **`glPointSize()`** function and have seen your first drawing command — **`glDrawArrays()`**.

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 each other, 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 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 back in Chapter 2 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 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, 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 430 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*, and we have used it 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 **glVertexAttrib*()**, it will update the value of the vertex attribute that is passed to the vertex shader. We can use this 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 that it writes its color values to, you can create an output variable in the vertex shader by using the `out` keyword as well. Anything you write to output variables in one shader get sent to similarly named variables 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 430 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, meaning that we can draw different colored triangles!

```
#version 430 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 non-trivial applications, you may wish to communicate a number of different pieces of data between stages, and 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 430 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`, upper case) and an instance name (`vs_out`, lower case). 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 430 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, avoiding confusing things such as having to use `vs_out` in a fragment shader, and 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 (sometimes known as simply the control shader, or abbreviated to TCS). This shader takes its input from the vertex shader and is primarily responsible for two things: the first being the determination of the level of tessellation that will be sent to the tessellation engine, and the second being 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, and so if this is what you want (as in our example application), you don't need to call it at all. When tessellation is active, the vertex shader runs once per control point whilst 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 be different 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:

```
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. The built-in input variable `gl_InvocationID` is used to 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 430 core

layout (vertices = 3) out;

void main(void)
{
    if (gl_InvocationID == 0)
    {
        gl_TessLevelInner[0] = 5.0;
        gl_TessLevelOuter[0] = 5.0;
        gl_TessLevelOuter[1] = 5.0;
        gl_TessLevelOuter[2] = 5.0;
    }
    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 (evaluation shader, or TES for short) runs an invocation for each

vertex produced by the tessellator. When the tessellation levels are high, this means that the tessellation evaluation shader could run an extremely large number of times, and so 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 start of the shader is a layout qualifier that sets the tessellation mode. In this case, we selected that the mode should be triangles. Other qualifiers, `equal_spacing` and `cw`, select 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 section "Tessellation" in Chapter 8.

In the remainder of the shader, you will see that it 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 earlier in Listing 3.7. This shader essentially implements pass-through tessellation. That is, the tessellated output patch is the exact same shape as the original, incoming triangular patch.

```
#version 430 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

In order 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 what type of polygons we want to affect and as 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 example with tessellation enabled and the two shaders of Listing 3.7 and Listing 3.8 is shown in Figure 3.1.

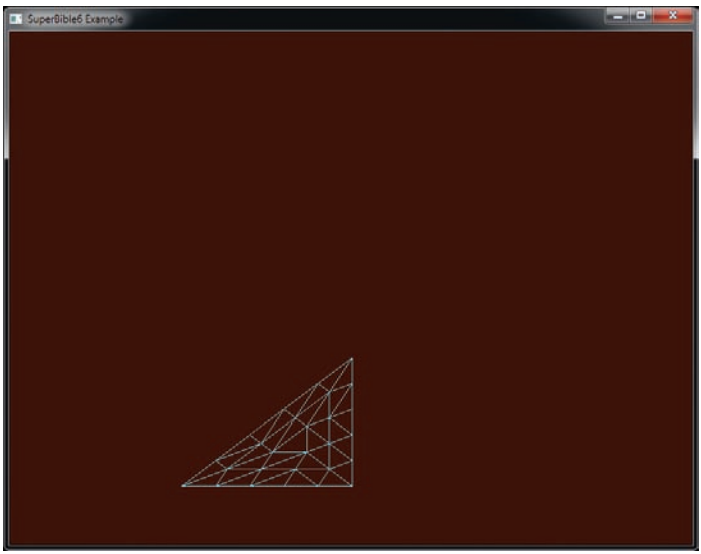


Figure 3.1: Our first tessellated triangle

Geometry Shaders

The geometry shader is logically the last shader stage in the front end, sitting after 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 amongst the shader stages in that it is able to increase or reduce the amount of data flowing in 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, on the other hand, 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 430 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, we have a loop that 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()` for you at the end of your shader, and so calling it explicitly is not necessary in this example. As a result of running this shader, three vertices will be produced and they will be 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.