

OpenGL[®] SuperBible



Sixth Edition

Comprehensive Tutorial and Reference



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

FREE SAMPLE CHAPTER



SHARE WITH OTHERS

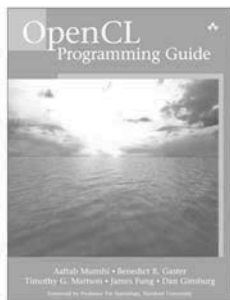
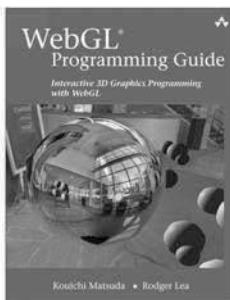
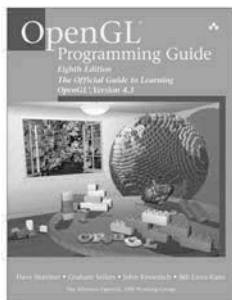
OpenGL[®]

SuperBible

Sixth Edition

OpenGL Series

from Addison-Wesley



◆ 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

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

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

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Wright, Richard S., Jr., 1965- author.

OpenGL superBible : comprehensive tutorial and reference.—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

Copyright © 2014 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

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	xlvi

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

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 <i>Unreal</i>	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 <code>glReadPixels()</code> into system memory	599
Figure 13.10	GPUView showing the effect of <code>glReadPixels()</code> 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 <code>NSView</code> 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 unsynched 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 <code>gl_in[]</code>	312
Listing 8.19	The definition of <code>gl_in[]</code>	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	Equiangular 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 <code>wglChoosePixelFormatARB()</code>	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 <code>GLCoreProfileView</code> 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 <code>GLSurfaceView</code>	732
Listing 14.19	Setting up and rendering	733
Listing 14.20	Construction and initialization of the <code>GLKView</code>	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

Chapter 8

Primitive Processing

WHAT YOU'LL LEARN IN THIS CHAPTER

- How to use tessellation to add geometric detail to your scenes
- How to use geometry shaders to process whole primitives and create geometry on the fly

In the previous chapters, you've read about the OpenGL pipeline and have been at least briefly introduced to the functions of each of its stages. We've covered the vertex shader stage in some detail, including how its inputs are formed and where its outputs go. A vertex shader runs once on each of the vertices you send OpenGL and produces one set of outputs for each. The next few stages of the pipeline seem similar to vertex shaders at first, but can actually be considered *primitive processing* stages. First, the two tessellation shader stages and the fixed-function tessellator that they flank together process *patches*. Next, the geometry shader processes entire primitives (points, lines, and triangles) and runs once for each. In this chapter, we'll cover both tessellation and geometry shading, and investigate some of the OpenGL features that they unlock.

Tessellation

As introduced in the section “Tessellation” in Chapter 3, tessellation is the process of breaking a large primitive referred to as a *patch* into many smaller primitives before rendering them. There are many uses for tessellation, but the most common application is to add geometric detail to otherwise lower fidelity meshes. In OpenGL, tessellation is produced using three distinct stages of the pipeline — the tessellation control shader (TCS), the fixed-function tessellation engine, and the tessellation evaluation shader (TES). Logically, these three stages fit between the vertex shader and the geometry shader stage. When tessellation is active, incoming vertex data is first processed as normal by the vertex shader and then passed, in groups, to the tessellation control shader.

The tessellation control shader operates on groups of up to 32 vertices¹ at a time, collectively known as a patch. In the context of tessellation, the input vertices are often referred to as *control points*. The tessellation control shader is responsible for generating three things:

- The per-patch inner and outer tessellation factors
- The position and other attributes for each output control point
- Per-patch user-defined varyings

The tessellation factors are sent on to the fixed-function tessellation engine, which uses them to determine the way that it will break up the patch into smaller primitives. Besides the tessellation factors, the output of a tessellation control shader is a new patch (i.e., a new collection of vertices) that is passed to the tessellation evaluation shader after the patch has been tessellated by the tessellation engine. If some of the data is common to all output vertices (such as the color of the patch), then that data may be marked as *per patch*. When the fixed-function tessellator runs, it generates a new set of vertices spaced across the patch as determined by the tessellation factors and the tessellation mode, which is determined using a layout declaration in the tessellation evaluation shader. The only input to the tessellation evaluation shader generated by OpenGL is a set of coordinates indicating where in the patch the vertex lies. When the tessellator is generating triangles, those coordinates are *barycentric*

1. The minimum number of vertices per patch required to be supported by the OpenGL specification is 32. However, the upper limit is not fixed and may be determined by retrieving the value of `GL_MAX_PATCH_VERTICES`.

coordinates. When the tessellation engine is generating lines or triangles, those coordinates are simply a pair of normalized values indicating the relative position of the vertex. This is stored in the `gl_TessCoord` input variable. This setup is shown in the schematic of Figure 8.1.

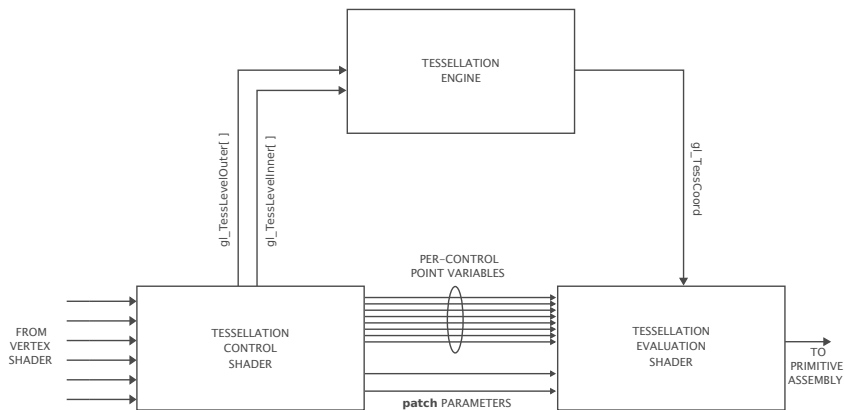


Figure 8.1: Schematic of OpenGL tessellation

Tessellation Primitive Modes

The tessellation mode is used to determine how OpenGL breaks up patches into primitives before passing them on to rasterization. This mode is set using an input layout qualifier in the tessellation evaluation shader and may be one of **quads**, **triangles**, or **isolines**. This primitive mode not only controls the form of the primitives produced by the tessellator, but also the interpretation of the `gl_TessCoord` input variable in the tessellation evaluation shader.

Tessellation Using Quads

When the chosen tessellation mode is set to quads, the tessellation engine will generate a quadrilateral (or quad) and break it up into a set of triangles. The two elements of the `gl_TessLevelInner[]` array should be written by the tessellation control shader and control the level of tessellation applied to the innermost region within the quad. The first element sets the tessellation in the horizontal (*u*) direction, and the second element sets the tessellation level applied in the vertical (*v*) direction. Also, all four elements of the `gl_TessLevelOuter[]` array should be written by the tessellation control shader and are used to determine the level of tessellation applied to the outer edges of the quad. This is shown in Figure 8.2.

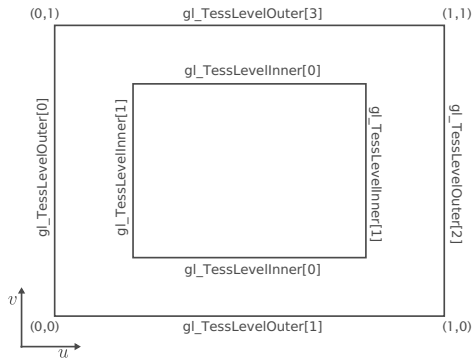


Figure 8.2: Tessellation factors for quad tessellation

When the quad is tessellated, the tessellation engine generates vertices across a two-dimensional domain normalized within the quad. The value stored in the `gl_TessCoord` input variable sent to the tessellation evaluation shader is then a two-dimensional vector (that is, only the x and y components of `gl_TessCoord` are valid) containing the normalized coordinate of the vertex within the quad. The tessellation evaluation shader can use these coordinates to generate its outputs from the inputs passed by the tessellation control shader. An example of quad tessellation produced by the `tessmodes` sample application is shown in Figure 8.3.

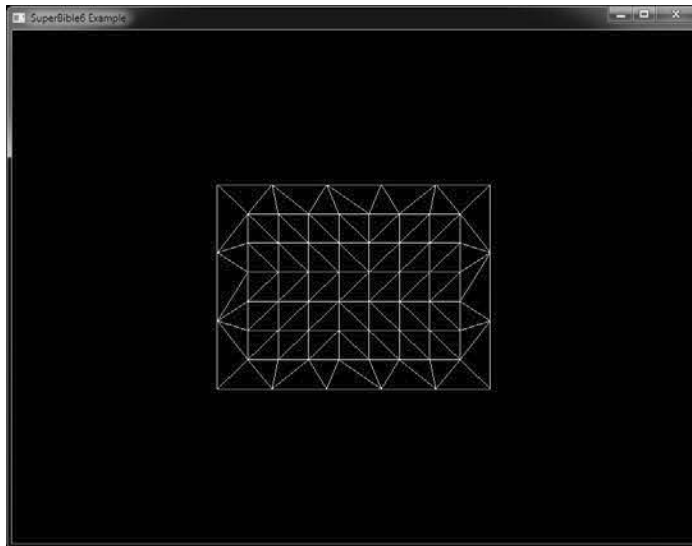


Figure 8.3: Quad tessellation example

In Figure 8.3, the inner tessellation factors in the u and v directions were set to 9.0 and 7.0, respectively. The outer tessellation factors were set to 3.0 and 5.0 in the u and v directions. This was accomplished using the very simple tessellation control shader shown in Listing 8.1.

```
#version 430 core

layout (vertices = 4) out;

void main(void)
{
    if (gl_InvocationID == 0)
    {
        gl_TessLevelInner[0] = 9.0;
        gl_TessLevelInner[1] = 7.0;
        gl_TessLevelOuter[0] = 3.0;
        gl_TessLevelOuter[1] = 5.0;
        gl_TessLevelOuter[2] = 3.0;
        gl_TessLevelOuter[3] = 5.0;
    }

    gl_out[gl_InvocationID].gl_Position =
        gl_in[gl_InvocationID].gl_Position;
}
```

Listing 8.1: Simple quad tessellation control shader example

The result of setting the tessellation factors in this way is visible in Figure 8.3. If you look closely, you will see that along the horizontal outer edges there are five divisions and along the vertical ones there are three divisions. On the interior, you can see that there are 9 divisions along the horizontal axis and 7 along the vertical.

The tessellation evaluation shader that generated Figure 8.3 is shown in Listing 8.2. Notice that the tessellation mode is set using the **quads** input layout qualifier near the front of the tessellation evaluation shader. The shader then uses the x and y components of `gl_TessCoordinate` to perform its own interpolation of the vertex position. In this case, the `gl_in[]` array is four elements long (as specified in the control shader shown in Listing 8.1).

```
#version 430 core

layout (quads) in;

void main(void)
{
    // Interpolate along bottom edge using x component of the
    // tessellation coordinate
    vec4 p1 = mix(gl_in[0].gl_Position,
                  gl_in[1].gl_Position,
                  gl_TessCoord.x);
    // Interpolate along top edge using x component of the
    // tessellation coordinate
    vec4 p2 = mix(gl_in[2].gl_Position,
```

```

        gl_in[3].gl_Position,
        gl_TessCoord.x);
    // Now interpolate those two results using the y component
    // of tessellation coordinate
    gl_Position = mix(p1, p2, gl_TessCoord.y);
}

```

Listing 8.2: Simple quad tessellation evaluation shader example

Tessellation Using Triangles

When the tessellation mode is set to triangles (again, using an input layout qualifier in the tessellation control shader), the tessellation engine produces a triangle that is then broken into many smaller triangles. Only the first element of the `gl_TessLevelInner[]` array is used, and this level is applied to the entirety of the inner area of the tessellated triangle. The first three elements of the `gl_TessLevelOuter[]` array are used to set the tessellation factors for the three edges of the triangle. This is shown in Figure 8.4.

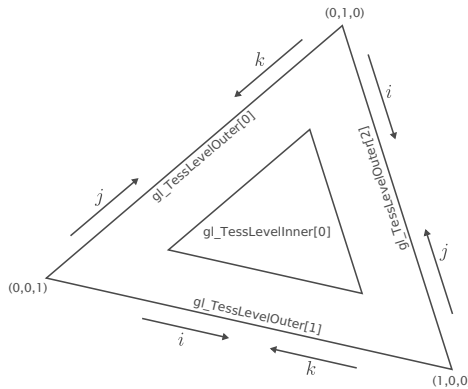


Figure 8.4: Tessellation factors for triangle tessellation

As the tessellation engine generates the vertices corresponding to the tessellated triangles, each vertex is assigned a three-dimensional coordinate called a *barycentric coordinate*. The three components of a barycentric coordinate can be used to form a weighted sum of three inputs representing the corners of a triangle and arrive at a value that is linearly interpolated across that triangle. An example of triangle tessellation is shown in Figure 8.5.

The tessellation control shader used to generate Figure 8.5 is shown in Listing 8.3. Notice how similar it is to Listing 8.1 in that all it does is write

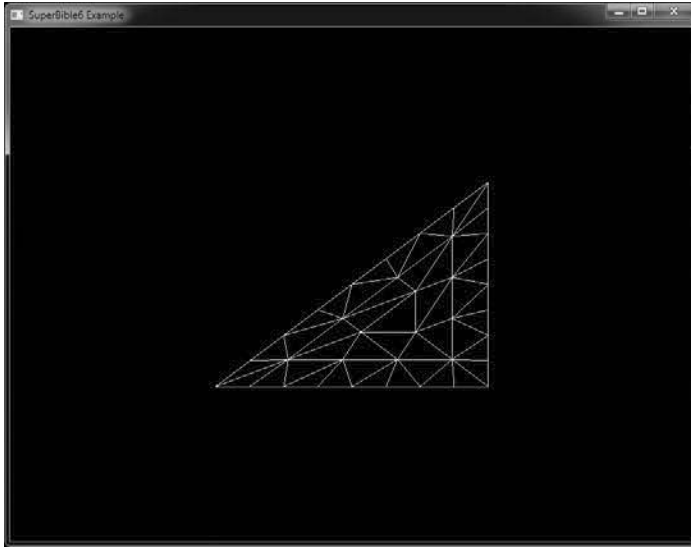


Figure 8.5: Triangle tessellation example

constants into the inner and outer tessellation levels and pass through the control point positions unmodified.

```
#version 430 core
layout (vertices = 3) out;

void main(void)
{
    if (gl_InvocationID == 0)
    {
        gl_TessLevelInner[0] = 5.0;
        gl_TessLevelOuter[0] = 8.0;
        gl_TessLevelOuter[1] = 8.0;
        gl_TessLevelOuter[2] = 8.0;
    }

    gl_out[gl_InvocationID].gl_Position =
        gl_in[gl_InvocationID].gl_Position;
}
```

Listing 8.3: Simple triangle tessellation control shader example

Listing 8.3 sets the inner tessellation level to 5.0 and all three outer tessellation levels to 8.0. Again, looking closely at Figure 8.5, you can see that each of the outer edges of the tessellated triangle has 8 divisions and the inner edges have 5 divisions. The tessellation evaluation shader that produced Figure 8.5 is shown in Listing 8.4.

```
#version 430 core

layout (triangles) 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 8.4: Simple triangle tessellation evaluation shader example

Again, to produce a position for each vertex generated by the tessellation engine, we simply calculate a weighted sum of the input vertices. This time, all three components of `gl_TessCoord` are used and represent the relative weights of the three vertices making up the outermost tessellated triangle. Of course, we're free to do anything we wish with the barycentric coordinates, the inputs from the tessellation control shader, and any other data we have access to in the evaluation shader.

Tessellation Using Isolines

Isoline tessellation is a mode of the tessellation engine where, rather than producing triangles, it produces real line primitives running along lines of equal v coordinate in the tessellation domain. Each line is broken up into segments along the u direction. The two outer tessellation factors stored in the first two components of `gl_TessLevelOuter[]` are used to specify the number of lines and the number of segments per line, respectively, and the inner tessellation factors (`gl_TessLevelInner[]`) are not used at all. This is shown in Figure 8.6.

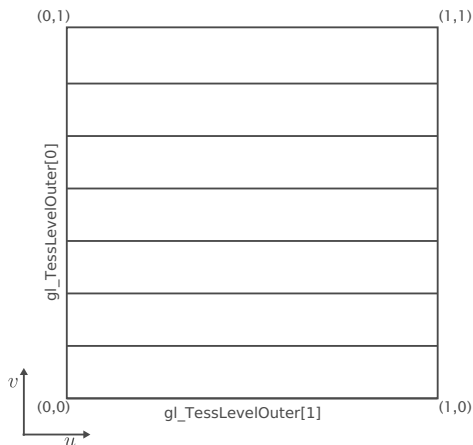


Figure 8.6: Tessellation factors for isoline tessellation

The tessellation control shader shown in Listing 8.5 simply set both the outer tessellation levels to 5.0 and doesn't write to the inner tessellation levels. The corresponding tessellation evaluation shader is shown in Listing 8.6.

```
#version 430 core

layout (vertices = 4) out;

void main(void)
{
    if (gl_InvocationID == 0)
    {
        gl_TessLevelOuter[0] = 5.0;
        gl_TessLevelOuter[1] = 5.0;
    }

    gl_out[gl_InvocationID].gl_Position =
        gl_in[gl_InvocationID].gl_Position;
}
```

Listing 8.5: Simple isoline tessellation control shader example

Notice that Listing 8.6 is virtually identical to Listing 8.2 except that the input primitive mode is set to **isolines**.

```
#version 430 core

layout (isolines) in;

void main(void)
{
    // Interpolate along bottom edge using x component of the
    // tessellation coordinate
    vec4 p1 = mix(gl_in[0].gl_Position,
                  gl_in[1].gl_Position,
                  gl_TessCoord.x);
    // Interpolate along top edge using x component of the
    // tessellation coordinate
    vec4 p2 = mix(gl_in[2].gl_Position,
                  gl_in[3].gl_Position,
                  gl_TessCoord.x);
    // Now interpolate those two results using the y component
    // of tessellation coordinate
    gl_Position = mix(p1, p2, gl_TessCoord.y);
}
```

Listing 8.6: Simple isoline tessellation evaluation shader example

The result of our extremely simple isoline tessellation example is shown in Figure 8.7.

Figure 8.7 doesn't really seem all that interesting. It's also difficult to see that each of the horizontal lines is actually made up of several segments.

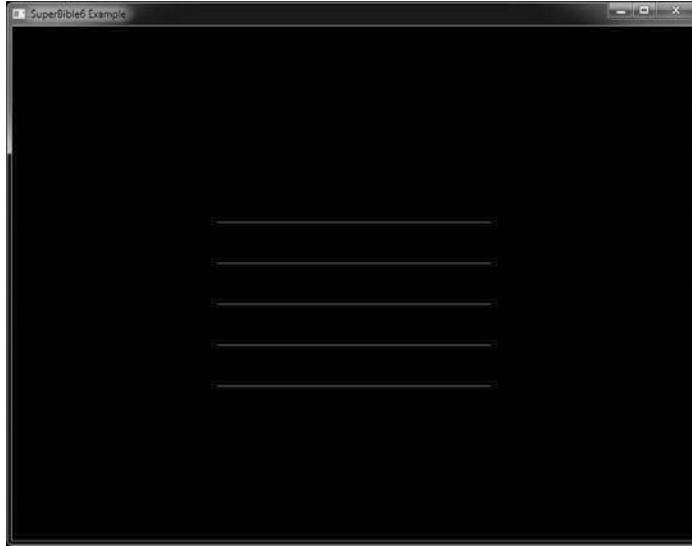


Figure 8.7: Isoline tessellation example

If, however, we change the tessellation evaluation shader to that shown in Listing 8.7, we can generate the image shown in Figure 8.8.

```
#version 430 core

layout (isolines) in;

void main(void)
{
    float r = (gl_TessCoord.y + gl_TessCoord.x / gl_TessLevelOuter[0]);
    float t = gl_TessCoord.x * 2.0 * 3.14159;
    gl_Position = vec4(sin(t) * r, cos(t) * r, 0.5, 1.0);
}
```

Listing 8.7: Isoline spirals tessellation evaluation shader

The shader in Listing 8.7 converts the incoming tessellation coordinates into polar form, with the radius r calculated as smoothly extending from zero to one, and with the angle t as a scaled version of the x component of the tessellation coordinate to produce a single revolution on each isoline. This produces the spiral pattern shown in Figure 8.8, where the segments of the lines are clearly visible.

Tessellation Point Mode

In addition to being able to render tessellated patches using triangles or lines, it's also possible to render the generated vertices as individual

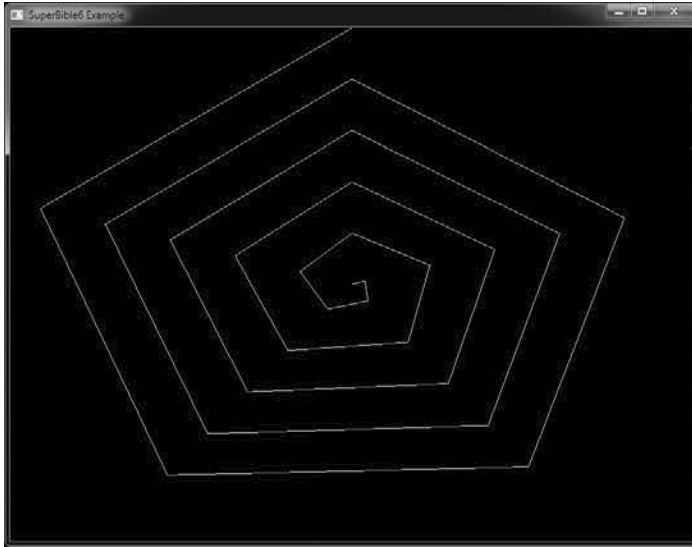


Figure 8.8: Tessellated isoline spirals example

points. This is known as *point mode* and is enabled using the `point_mode` input layout qualifier in the tessellation evaluation shader just like any other tessellation mode. When you specify that point mode should be used, the resulting primitives are points. However, this is somewhat orthogonal to the use of the **quads**, **triangles**, or **isolines** layout qualifiers. That is, you should specify `point_mode` *in addition* to one of the other layout qualifiers. The **quads**, **triangles**, and **isolines** still control the generation of `gl_TessCoord` and the interpretation of the inner and outer tessellation levels. For example, if the tessellation mode is **quads**, then `gl_TessCoord` is a two-dimensional vector, whereas if the tessellation mode is **triangles**, then it is a three-dimensional barycentric coordinate. Likewise, if the tessellation mode is **isolines**, only the outer tessellation levels are used, whereas if it is **triangles** or **quads**, the inner tessellation levels are used as well.

Figure 8.9 shows a version of Figure 8.5 rendered using point mode next to the original image. To produce the figure on the right, we simply change the input layout qualifier of Listing 8.4 to read:

```
layout (triangles, point_mode) in;
```

As you can see, the layout of the vertices is identical in both sides of Figure 8.9, but on the right, each vertex has been rendered as a single point.

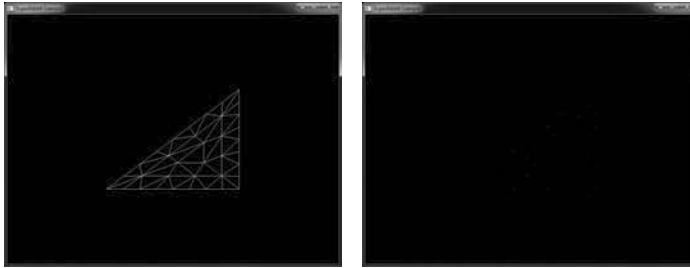


Figure 8.9: Triangle tessellated using point mode

Tessellation Subdivision Modes

The tessellation engine works by generating a triangle or quad primitive and then subdividing its edges into a number of segments determined by the inner and outer tessellation factors produced by the tessellation control shader. It then groups the generated vertices into points, lines, or triangles and sends them on for further processing. In addition to the type of primitives generated by the tessellation engine, you have quite a bit of control about how it subdivides the edges of the generated primitives.

By default, the tessellation engine will subdivide each edge into a number of equal-sized parts where the number of parts is set by the corresponding tessellation factor. This is known as `equal_spacing` mode, and although it is the default, it can be made explicit by including the following layout qualifier in your tessellation evaluation shader:

```
layout (equal_spacing) in;
```

Equal spacing mode is perhaps the easiest mode to comprehend — simply set the tessellation factor to the number segments you wish to subdivide your patch primitive into along each edge, and the tessellation engine takes care of the rest. Although simple, the `equal_spacing` mode comes with a significant disadvantage — as you alter the tessellation factor, it is always rounded up to the next nearest integer and will produce a visible jump from one level to the next as the tessellation factor changes. The two other modes alleviate this problem by allowing the segments to be non-equal in length. These modes are `fractional_even_spacing` and `fractional_odd_spacing`, and again, you can set these modes by using input layout qualifiers as follows:

```
layout (fractional_even_spacing) in;  
// or  
layout (fractional_odd_spacing) in;
```

With fractional even spacing, the tessellation factor is rounded to the next lower even integer and the edge subdivided as if that were the tessellation factor. With fractional odd spacing, the tessellation factor is rounded down to the next lower odd number and the edge subdivided as if that were the tessellation factor. Of course, with either scheme, there is a small remaining segment that doesn't have the same length as the other segments. That last segment is then cut in half, each half having the same length as the other and is therefore a *fractional* segment.

Figure 8.10 shows the same triangle tessellated with `equal_spacing` mode on the left, `fractional_even_spacing` mode in the center, and `fractional_odd_spacing` mode on the right.



Figure 8.10: Tessellation using different subdivision modes

In all three images shown in Figure 8.10, the inner and outer tessellation factors have been set to 5.3. In the leftmost image showing `equal_spacing` mode, you should be able to see that the number of segments along each of the outer edges of the triangle is 6 — the next integer after 5.3. In the center image, which shows `fractional_even_spacing` spacing, there are 4 equal-sized segments (as 4 is the next lower even integer to 5.3) and then two additional smaller segments. Finally, in the rightmost image, which demonstrates `fractional_odd_spacing`, you can see that there are 5 equal-sized segments (5 being the next lower odd integer to 5.3) and there are two very skinny segments that make up the rest.

If the tessellation level is animated, either by being explicitly turned up and down using a uniform, or calculated in the tessellation control shader, the length of the equal-sized segments and the two filler segments will change smoothly and dynamically. Whether you choose `fractional_even_spacing` or `fractional_odd_spacing` really depends on which looks better in your application — there is generally no real advantage to either. However, unless you need a guarantee that tessellated edges have equal-sized segments and you can live with popping if the tessellation level changes, `fractional_even_spacing` or `fractional_odd_spacing` will generally look better in any dynamic application than `equal_spacing`.

Controlling the Winding Order

In Chapter 3, “Following the Pipeline,” we introduced culling and explained how the *winding order* of a primitive affects how OpenGL decides whether to render it. Normally, the winding order of a primitive is determined by the order in which your application presents vertices to OpenGL. However, when tessellation is active, OpenGL generates all the vertices and connectivity information for you. In order to allow you to control the winding order of the resulting primitives, you can specify whether you want the vertices to be generated in clockwise or counterclockwise order. Again, this is specified using an input layout qualifier in the tessellation evaluation shader. To indicate that you want clockwise winding order, use the following layout qualifier:

```
layout (cw) in;
```

To specify that the winding order of the primitives generated by the tessellation engine be counterclockwise, include

```
layout (ccw) in;
```

The `cw` and `ccw` layout qualifiers can be combined with the other input layout qualifiers specified in the tessellation control shader. By default, the winding order is counterclockwise, and so you can omit this layout qualifier if that is what you need. Also, it should be self-evident that winding order only applies to triangles, and so if your application generates isolines or points, then the winding order is ignored — your shader can still include the winding order layout qualifier, but it won’t be used.

Passing Data between Tessellation Shaders

In this section, we have looked at how to set the inner and outer tessellation levels for the quad, triangle, and point primitive modes. However, the resulting images in Figures 8.3 through 8.8 aren’t particularly exciting, in part because we haven’t done anything but compute the positions of the resulting vertices and then just shaded the resulting primitives solid white. In fact, we have rendered all of these images using lines by setting the polygon mode to `GL_LINE` with the `glPolygonMode()` function. To produce something a little more interesting, we’re going to need to pass more data along the pipeline.

Before a tessellation control shader is run, each vertex represents a control point, and the vertex shader runs once for each input control point and produces its output as normal. The vertices (or control points) are then grouped together and passed together to the tessellation control shader.

The tessellation control shader processes this group of control points and produces a new group of control points that may or may not have the same number of elements in it as the original group. The tessellation control shader actually runs once for each control point in the *output* group, but each invocation of the tessellation control shader has access to all of the input control points. For this reason, both the inputs to and outputs from a tessellation control shader are represented as arrays. The input arrays are sized by the number of control points in each patch, which is set by calling

```
glPatchParameteri(GL_PATCH_VERTICES, n);
```

Here, *n* is the number of vertices per patch. By default, the number of vertices per patch is 3. The size of the input arrays in the tessellation control shader is set by this parameter, and their contents come from the vertex shader. The built-in variable `gl_in[]` is always available and is declared as an array of the `gl_PerVertex` structure. This structure is where the built-in outputs go after you write to them in your vertex shader. All other outputs from the vertex shader become arrays in the tessellation control shader as well. In particular, if you use an output block in your vertex shader, the instance of that block becomes an array of instances in the tessellation control shader. So, for example

```
out VS_OUT
{
    vec4    foo;
    vec3    bar;
    int     baz
} vs_out;
```

becomes

```
in VS_OUT
{
    vec4    foo;
    vec3    bar;
    int     baz;
} tcs_in[];
```

in the tessellation evaluation shader.

The output of the tessellation control shader is also an array, but its size is set by the vertices output layout qualifier at the front of the shader. It is quite common to set the input and output vertex count to the same value (as was the case in the samples earlier in this section) and then pass the input directly to the output from the tessellation control shader. However, there's no requirement for this, and the size of the output arrays in the tessellation control shader is limited by the value of the `GL_MAX_PATCH_VERTICES` constant.

As the outputs of the tessellation control shader are arrays, so the inputs to the tessellation evaluation shader are also similarly sized arrays. The tessellation evaluation shader runs once per generated vertex and, like the tessellation control shader, has access to all of the data for all of the vertices in the patch.

In addition to the per-vertex data passed from tessellation control shader to the tessellation evaluation shader in arrays, it's also possible to pass data directly between the stages that is constant across an entire patch. To do this, simply declare the output variable in the tessellation control shader and the corresponding input in the tessellation evaluation shader using the **patch** keyword. In this case the variable does not have to be declared as an array (although you are welcome to use arrays as **patch** qualified variables) as there is only one instance per patch.

Rendering without a Tessellation Control Shader

The purpose of the tessellation control shader is to perform tasks such as computing the value of per-patch inputs to the tessellation evaluation shader and to calculate the values of the inner and outer tessellation levels that will be used by the fixed-function tessellator. However, in some simple applications, there are no per-patch inputs to the tessellation evaluation shader, and the tessellation control shader only writes constants to the tessellation levels. In this case, it's actually possible to set up a program with a tessellation evaluation shader, but without a tessellation control shader.

When no tessellation control shader is present, the default values of all inner and outer tessellation levels is 1.0. You can change this by calling **glPatchParameterfv()**, whose prototype is

```
void glPatchParameterfv(GLenum pname,
                        const GLfloat * values);
```

If **pname** is **GL_PATCH_DEFAULT_INNER_LEVEL**, then **values** should point to an array of two floating-point values that will be used as the new default inner tessellation levels in the absence of a tessellation control shader. Likewise, if **pname** is **GL_PATCH_DEFAULT_OUTER_LEVEL**, then **values** should point to an array of four floating-point values that will be used as the new default outer tessellation levels.

If no tessellation control shader is part of the current pipeline, then the number of control points that is presented to the tessellation evaluation shader is the same as the number of control points per patch set by the **glPatchParameteri()** when the **pname** parameter is set to

`GL_PATCH_VERTICES`. In this case, the input to the tessellation evaluation shader comes directly from the vertex shader. That is, the input to the tessellation evaluation shader is an array formed from the outputs of the vertex shader invocations that generated the patch.

Communication between Shader Invocations

Although the purpose of output variables in tessellation control shaders is primarily to pass data to the tessellation evaluation shader, they also have a secondary purpose. That is, to communicate data between control shader invocations. As you have read, the tessellation control shader runs a single invocation for each *output* control point in a patch. Each output variable in the tessellation control shader is therefore an array, the length of which is the number of control points in the output patch. Normally, each tessellation control shader invocation will take responsibility for writing to one element of this array.

What might not be obvious is that tessellation control shaders can actually *read* from their output variables — including those that might be written by other invocations! Now, the tessellation control shader is designed in such a way that the invocations can run in parallel. However, there is no ordering guarantee over how those shaders actually execute your code. That means that you have no idea if, when you read from another invocation's output variable, that that invocation has actually written data there.

To deal with this, GLSL includes the `barrier()` function. This is known as a flow-control barrier, as it enforces relative order to the execution of multiple shader invocations. The `barrier()` function really shines when used in compute shaders — we'll get to that later. However, it's available in a limited form in tessellation control shaders, too, with a number of restrictions. In particular, in a tessellation control shader, `barrier()` may only be called directly from within your `main()` function, and can't be inside any control flow structures (such as `if`, `else`, `while`, or `switch`).

When you call `barrier()`, the tessellation control shader invocation will stop and wait for all the other invocations in the same patch to catch up. It won't continue execution until all the other invocations have reached the same point. This means that if you write to an output variable in a tessellation control shader and then call `barrier()`, you can be sure that all the other invocations have done the same thing by the time `barrier()` returns, and therefore it's safe to go ahead and read from the other invocations' output variables.

Tessellation Example — Terrain Rendering

To demonstrate a potential use for tessellation, we will cover a simple terrain rendering system based on quadrilateral patches and *displacement mapping*. The code for this example is part of the `dispmmap` sample. A displacement map is a texture that contains the displacement from a surface at each location. Each patch represents a small region of a landscape that is tessellated depending on its likely screen-space area. Each tessellated vertex is moved along the tangent to the surface by the value stored in the displacement map. This adds geometric detail to the surface without needing to explicitly store the positions of each tessellated vertex. Rather, only the displacements from an otherwise flat landscape are stored in the displacement map and are applied at runtime in the tessellation evaluation shader. The displacement map (which is also known as a height map) used in the example is shown in Figure 8.11.

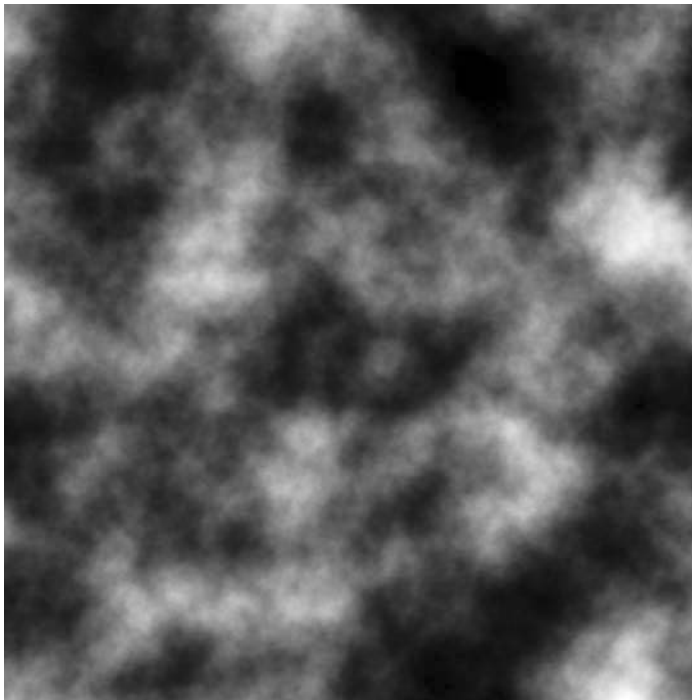


Figure 8.11: Displacement map used in terrain sample

Our first step is to set up a simple vertex shader. As each patch is effectively a simple quad, we can use constants in the shader to represent the four vertices rather than setting up vertex arrays for it. The complete

shader is shown in Listing 8.8. The shader uses the instance number (stored in `gl_InstanceID`) to calculate an offset for the patch, which is a one-unit square in the xz plane, centered on the origin. In this application, we will render a grid of 64×64 patches, and so the x and y offsets for the patch are calculated by taking `gl_InstanceID` modulo 64 and `gl_InstanceID` divided by 64. The vertex shader also calculates the texture coordinates for the patch, which are passed to the tessellation control shader in `vs_out.tc`.

```
#version 430 core

out VS_OUT
{
    vec2 tc;
} vs_out;

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

    int x = gl_InstanceID & 63;
    int y = gl_InstanceID >> 6;
    vec2 offs = vec2(x, y);

    vs_out.tc = (vertices[gl_VertexID].xz + offs + vec2(0.5)) / 64.0;
    gl_Position = vertices[gl_VertexID] + vec4(float(x - 32), 0.0,
                                              float(y - 32), 0.0);
}
```

Listing 8.8: Vertex shader for terrain rendering

Next, we come to the tessellation control shader. Again, the complete shader is shown in Listing 8.9. In this example, the bulk of the rendering algorithm is implemented in the tessellation control shader, and the majority of the code is only executed by the first invocation. Once we have determined that we are the first invocation by checking that `gl_InvocationID` is zero, we calculate the tessellation levels for the whole patch. First, we project the corners of the patch into normalized device coordinates by multiplying the incoming coordinates by the model-view-projection matrix and then dividing each of the four points by their own homogeneous $.w$ component.

Next, we calculate the length of each of the four edges of the patch in normalized device space after projecting them onto the xy plane by ignoring their z components. Then, the shader calculates the tessellation levels of each edge of the patch as a function of its length using a simple scale and bias. Finally, the inner tessellation factors are simply set to the

minimum of the outer tessellation factors calculated from the edge lengths in the horizontal or vertical directions.

You may also have noticed a piece of code in Listing 8.9 that checks whether all of the z coordinates of the projected control points are less than zero and then sets the outer tessellation levels to zero if this happens. This is an optimization that culls entire patches that are behind² the viewer.

```
#version 430 core

layout (vertices = 4) out;

in VS_OUT
{
    vec2 tc;
} tcs_in[];

out TCS_OUT
{
    vec2 tc;
} tcs_out[];

uniform mat4 mvp;

void main(void)
{
    if (gl_InvocationID == 0)
    {
        vec4 p0 = mvp * gl_in[0].gl_Position;
        vec4 p1 = mvp * gl_in[1].gl_Position;
        vec4 p2 = mvp * gl_in[2].gl_Position;
        vec4 p3 = mvp * gl_in[3].gl_Position;
        p0 /= p0.w;
        p1 /= p1.w;
        p2 /= p2.w;
        p3 /= p3.w;
        if (p0.z <= 0.0 ||
            p1.z <= 0.0 ||
            p2.z <= 0.0 ||
            p3.z <= 0.0)
        {
            gl_TessLevelOuter[0] = 0.0;
            gl_TessLevelOuter[1] = 0.0;
            gl_TessLevelOuter[2] = 0.0;
            gl_TessLevelOuter[3] = 0.0;
        }
        else
        {
            float l0 = length(p2.xy - p0.xy) * 16.0 + 1.0;
            float l1 = length(p3.xy - p2.xy) * 16.0 + 1.0;
            float l2 = length(p3.xy - p1.xy) * 16.0 + 1.0;
            float l3 = length(p1.xy - p0.xy) * 16.0 + 1.0;
            gl_TessLevelOuter[0] = l0;
```

2. This optimization is actually not foolproof. If the viewer were at the bottom of a very steep cliff and looking directly upwards, all four corners of the base patch may be behind the viewer, whereas the cliff cutting through the patch will extend into the viewer's field of view.

```

        gl_TessLevelOuter[1] = l1;
        gl_TessLevelOuter[2] = l2;
        gl_TessLevelOuter[3] = l3;
        gl_TessLevelInner[0] = min(l1, l3);
        gl_TessLevelInner[1] = min(l0, l2);
    }
}

gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
tcs_out[gl_InvocationID].tc = tcs_in[gl_InvocationID].tc;
}

```

Listing 8.9: Tessellation control shader for terrain rendering

Once the tessellation control shader has calculated the tessellation levels for the patch, it simply copies its input to its output. It does this per instance and passes the resulting data to the tessellation evaluation shader, which is shown in Listing 8.10.

```

#version 430 core

layout (quads, fractional_odd_spacing) in;

uniform sampler2D tex_displacement;

uniform mat4 mvp;
uniform float dmap_depth;

in TCS_OUT
{
    vec2 tc;
} tes_in[];

out TES_OUT
{
    vec2 tc;
} tes_out;

void main(void)
{
    vec2 tc1 = mix(tes_in[0].tc, tes_in[1].tc, gl_TessCoord.x);
    vec2 tc2 = mix(tes_in[2].tc, tes_in[3].tc, gl_TessCoord.x);
    vec2 tc = mix(tc2, tc1, gl_TessCoord.y);

    vec4 p1 = mix(gl_in[0].gl_Position,
                  gl_in[1].gl_Position,
                  gl_TessCoord.x);
    vec4 p2 = mix(gl_in[2].gl_Position,
                  gl_in[3].gl_Position,
                  gl_TessCoord.x);
    vec4 p = mix(p2, p1, gl_TessCoord.y);

    p.y += texture(tex_displacement, tc).r * dmap_depth;

    gl_Position = mvp * p;
    tes_out.tc = tc;
}

```

Listing 8.10: Tessellation evaluation shader for terrain rendering

The tessellation evaluation shader shown in Listing 8.10 first calculates the texture coordinate of the generated vertex by linearly interpolating the texture coordinates passed from the tessellation control shader of Listing 8.9 (which were in turn generated by the vertex shader of Listing 8.8). It then applies a similar interpolation to the incoming control point positions to produce the position of the outgoing vertex. However, once it's done that, it uses the texture coordinate that it calculated to offset the vertex in the y direction before multiplying that result by the model-view-projection matrix (the same one that was used in the tessellation control shader). It also passes the computed texture coordinate on to the fragment shader in `tes_out.tc`. That fragment shader is shown in Listing 8.11.

```
#version 430 core

out vec4 color;

layout (binding = 1) uniform sampler2D tex_color;

in TES_OUT
{
    vec2 tc;
} fs_in;

void main(void)
{
    color = texture(tex_color, fs_in.tc);
}
```

Listing 8.11: Fragment shader for terrain rendering

The fragment shader shown in Listing 8.11 is really pretty simple. All it does is use the texture coordinate that the tessellation evaluation shader gave it to look up a color for the fragment. The result of rendering with this set of shaders is shown in Figure 8.12.

Of course, if we've done our job correctly, you shouldn't be able to tell that the underlying geometry is tessellated. However, if you look at the wireframe version of the image shown in Figure 8.13, you can clearly see the underlying triangular mesh of the landscape. The goals of the program are that all of the triangles rendered on the screen have roughly similar screen-space area and that sharp transitions in the level of tessellation are not visible in the rendered image.

Tessellation Example — Cubic Bézier Patches

In the displacement mapping example, all we did was use a (very large) texture to drive displacement from a flat surface and then use tessellation

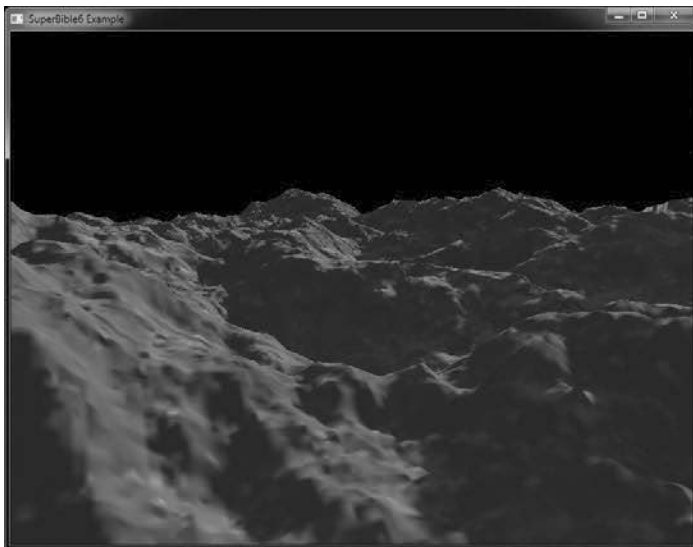


Figure 8.12: Terrain rendered using tessellation

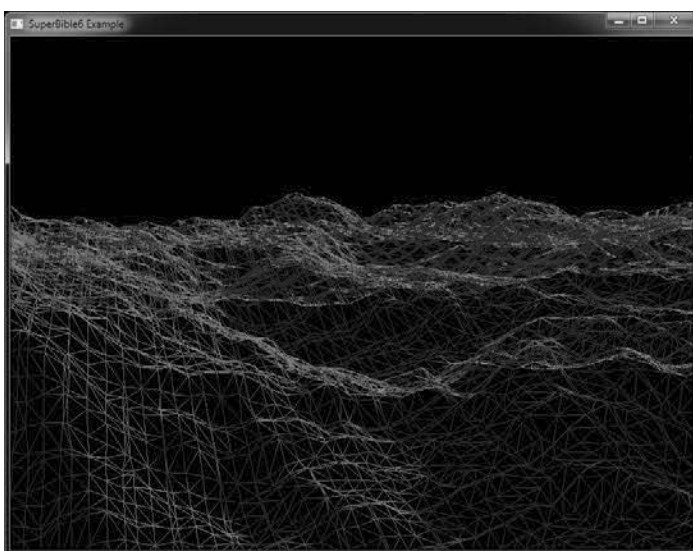


Figure 8.13: Tessellated terrain in wireframe

to increase the number of polygons in the scene. This is a type of brute force, data driven approach to geometric complexity. In the `cubicbezier` example described here, we will use math to drive geometry — we're going

to render a *cubic Bézier patch*. If you look back to Chapter 4, you'll see that we've covered all the number crunching we'll need here.

A cubic Bézier patch is a type of *higher order surface* and is defined by a number of *control points*³ that provide input to a number of interpolation functions that define the surface's shape. A Bézier patch has 16 control points, laid out in a 4×4 grid. Very often (including in this example), they are equally spaced in two dimensions varying only in distance from a shared plane. However, they don't have to be. Free-form Bézier patches are extremely powerful modeling tools, being used natively by many pieces of modeling and design software. With OpenGL tessellation, it's possible to render them directly.

The simplest method of rendering a Bézier patch is to treat the four control points in each row of the patch as the control points for a single cubic Bézier curve, just as was described in Chapter 4. Given our 4×4 grid of control points, we have 4 curves, and if we interpolate along each of them using the same value of t , we will end up with 4 new points. We use these 4 points as the control points for a second cubic Bézier curve. Interpolating along this second curve using a new value for t gives us a second point that lies on the patch. The two values of t (let's call them t_0 and t_1) are the *domain* of the patch and are what is handed to us in the tessellation evaluation shader in `gl_TessCoord.xy`.

In this example, we'll perform tessellation in view space. That means that in our vertex shader, we'll transform our patch's control points into view space by multiplying their coordinates by the model-view matrix — that is all. This simple vertex shader is shown in Listing 8.12.

```
#version 430 core

in vec4 position;

uniform mat4 mv_matrix;

void main(void)
{
    gl_Position = mv_matrix * position;
}
```

Listing 8.12: Cubic Bézier patch vertex shader

3. It should now be evident why the tessellation control shader is so named.

Once our control points are in view space, they are passed to our tessellation control shader. In a more advanced⁴ algorithm, we could project the control points into screen space, determine the length of the curve, and set the tessellation factors appropriately. However, in this example, we'll settle with a simple fixed tessellation factor. As in previous examples, we set the tessellation factors only when `gl_InvocationID` is zero, but pass all of the other data through once per invocation. The tessellation control shader is shown in Listing 8.13.

```
#version 430 core

layout (vertices = 16) out;

void main(void)
{
    if (gl_InvocationID == 0)
    {
        gl_TessLevelInner[0] = 16.0;
        gl_TessLevelInner[1] = 16.0;
        gl_TessLevelOuter[0] = 16.0;
        gl_TessLevelOuter[1] = 16.0;
        gl_TessLevelOuter[2] = 16.0;
        gl_TessLevelOuter[3] = 16.0;
    }

    gl_out[gl_InvocationID].gl_Position =
        gl_in[gl_InvocationID].gl_Position;
}
```

Listing 8.13: Cubic Bézier patch tessellation control shader

Next, we come to the tessellation evaluation shader. This is where the meat of the algorithm lies. The shader in its entirety is shown in Listing 8.14. You should recognize the `cubic_bezier` and `quadratic_bezier` functions from Chapter 4. The `evaluate_patch` function is responsible for evaluating⁵ the vertex's coordinate given the input patch coordinates and the vertex's position within the patch.

```
#version 430 core

layout (quads, equal_spacing, cw) in;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
```

4. To do this right, we'd need to evaluate the length of the Bézier curve, which involves calculating an integral over a non-closed form... which is hard.

5. You should also now see why the tessellation evaluation shader is so named.

```

out TES_OUT
{
    vec3 N;
} tes_out;

vec4 quadratic_bezier(vec4 A, vec4 B, vec4 C, float t)
{
    vec4 D = mix(A, B, t);
    vec4 E = mix(B, C, t);

    return mix(D, E, t);
}

vec4 cubic_bezier(vec4 A, vec4 B, vec4 C, vec4 D, float t)
{
    vec4 E = mix(A, B, t);
    vec4 F = mix(B, C, t);
    vec4 G = mix(C, D, t);

    return quadratic_bezier(E, F, G, t);
}

vec4 evaluate_patch(vec2 at)
{
    vec4 P[4];
    int i;

    for (i = 0; i < 4; i++)
    {
        P[i] = cubic_bezier(gl_in[i + 0].gl_Position,
                           gl_in[i + 4].gl_Position,
                           gl_in[i + 8].gl_Position,
                           gl_in[i + 12].gl_Position,
                           at.y);
    }

    return cubic_bezier(P[0], P[1], P[2], P[3], at.x);
}

const float epsilon = 0.001;

void main(void)
{
    vec4 p1 = evaluate_patch(gl_TessCoord.xy);
    vec4 p2 = evaluate_patch(gl_TessCoord.xy + vec2(0.0, epsilon));
    vec4 p3 = evaluate_patch(gl_TessCoord.xy + vec2(epsilon, 0.0));

    vec3 v1 = normalize(p2.xyz - p1.xyz);
    vec3 v2 = normalize(p3.xyz - p1.xyz);

    tes_out.N = cross(v1, v2);

    gl_Position = proj_matrix * p1;
}

```

Listing 8.14: Cubic Bézier patch tessellation evaluation shader

In our tessellation evaluation shader, we calculate the surface normal to the patch by evaluating the patch position at two points very close to the point under consideration, using the additional points to calculate two

vectors that lie on the patch and then taking their cross product. This is passed to the fragment shader shown in Listing 8.15.

```
#version 430 core
out vec4 color;

in TES_OUT
{
    vec3 N;
} fs_in;

void main(void)
{
    vec3 N = normalize(fs_in.N);
    vec4 c = vec4(1.0, -1.0, 0.0, 0.0) * N.z +
            vec4(0.0, 0.0, 0.0, 1.0);
    color = clamp(c, vec4(0.0), vec4(1.0));
}
```

Listing 8.15: Cubic Bézier patch fragment shader

This fragment shader performs a very simple lighting calculation using the z component of the surface normal. The result of rendering with this shader is shown in Figure 8.14.

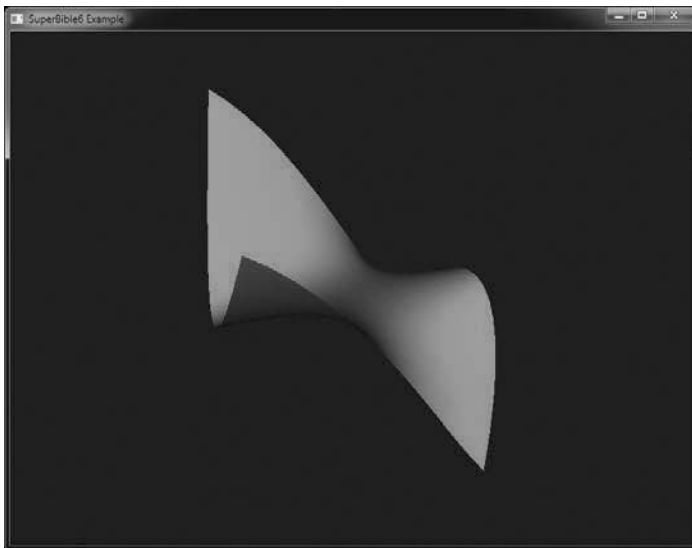


Figure 8.14: Final rendering of a cubic Bézier patch

Because the rendered patch shown in Figure 8.14 is smooth, it is hard to see the tessellation that has been applied to the shape. The left of Figure 8.15 shows a wireframe representation of the tessellated patch, and the right side of Figure 8.15 shows the patch's control points and the control cage, which is formed by creating a grid of lines between the control points.

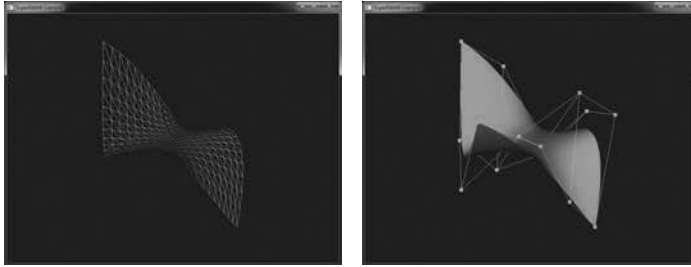


Figure 8.15: A Bézier patch and its control cage

Geometry Shaders

The geometry shader is unique in contrast to the other shader types in that it processes a whole primitive (triangle, line, or point) at once and can actually change the amount of data in the OpenGL pipeline programmatically. A vertex shader processes one vertex at a time; it cannot access any other vertex's information and is strictly one-in, one-out. That is, it cannot generate new vertices, and it cannot stop the vertex from being processed further by OpenGL. The tessellation shaders operate on patches and can set tessellation factors, but have little further control over how patches are tessellated, and cannot produce disjoint primitives. Likewise, the fragment shader processes a single fragment at a time, cannot access any data owned by another fragment, cannot create new fragments, and can only destroy fragments by discarding them. On the other hand, a geometry shader has access to all of the vertices in a primitive (up to six with the primitive modes `GL_TRIANGLES_ADJACENCY` and `GL_TRIANGLE_STRIP_ADJACENCY`), can change the type of a primitive, and can even create and destroy primitives.

Geometry shaders are an optional part of the OpenGL pipeline. When no geometry shader is present, the outputs from the vertex or tessellation evaluation shader are interpolated across the primitive being rendered and are fed directly to the fragment shader. When a geometry shader is present, however, the outputs of the vertex or tessellation evaluation

shader become the inputs to the geometry shader, and the outputs of the geometry shader are what are interpolated and fed to the fragment shader. The geometry shader can further process the output of the vertex or tessellation evaluation shader, and if it is generating new primitives (this is called amplification), it can apply different transformations to each primitive as it creates them.

The Pass-Through Geometry Shader

As explained back in Chapter 3, “Following the Pipeline,” the simplest geometry shader that allows you to render anything is the *pass-through* shader, which is shown in Listing 8.16.

```
#version 430 core

layout (triangles) in;
layout (triangle_strip) out;
layout (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();
    }
    EndPrimitive();
}
```

Listing 8.16: Source code for a simple geometry shader

This is a very simple pass-through geometry shader, which sends its input to its output without modifying it. It looks similar to a vertex shader, but there are a few extra differences to cover. Going over the shader a few lines at a time makes everything clear. The first few lines simply set up the version number (430) of the shader just like in any other shader. The next couple of lines are the first geometry shader-specific parts. They are shown again in Listing 8.17.

```
#version 430 core

layout (triangles) in;
layout (triangle_strip) out;
layout (max_vertices = 3) out;
```

Listing 8.17: Geometry shader layout qualifiers

These set the input and output primitive modes using a layout qualifier. In this particular shader we’re using **triangles** for the input and

`triangle_strip` for the output. Other primitive types, along with the layout qualifier, are covered later. For the geometry shader's output, not only do we specify the primitive type, but the maximum number of vertices expected to be generated by the shader (through the `max_vertices` qualifier). This shader produces individual triangles (generated as very short triangle strips), so we specified 3 here.

Next is our `main()` function, which is again similar to what might be seen in a vertex or fragment shader. The shader contains a loop, and the loop runs a number of times determined by the length of the built-in array, `gl_in`. This is another geometry shader-specific variable. Because the geometry shader has access to all of the vertices of the input primitive, the input has to be declared as an array. All of the built-in variables that are written by the vertex shader (such as `gl_Position`) are placed into a structure, and an array of these structures is presented to the geometry shader in a variable called `gl_in`.

The length of the `gl_in[]` array is determined by the input primitive mode, and because in this particular shader, triangles are the input primitive mode, the size of `gl_in[]` is three. The inner loop is given again in Listing 8.18.

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

Listing 8.18: Iterating over the elements of `gl_in[]`

Inside our loop, we're generating vertices by simply copying the elements of `gl_in[]` to the geometry shader's output. A geometry shader's outputs are similar to the vertex shader's outputs. Here, we're writing to `gl_Position`, just as we would in a vertex shader. When we're done setting up all of the new vertex's attributes, we call `EmitVertex()`. This is a built-in function, specific to geometry shaders that tells the shader that we're done with our work for this vertex and that it should store all that information away and prepare to start setting up the next vertex.

Finally, after the loop has finished executing, there's a call to another special, geometry shader-only function, `EndPrimitive()`. `EndPrimitive()` tells the shader that we're done producing vertices for the current primitive and to move on to the next one. We specified `triangle_strip` as the output for our shader, and so if we continue to call `EmitVertex()` more than three times, OpenGL continues adding triangles to the triangle strip. If we need our geometry shader to generate separate, individual triangles or multiple,

unconnected triangle strips (remember, geometry shaders can create new or amplify geometry), we could call `EndPrimitive()` between each one to mark their boundaries. If you don't call `EndPrimitive()` somewhere in your shader, the primitive is automatically ended when the shader ends.

Using Geometry Shaders in an Application

Geometry shaders, like the other shader types, are created by calling the `glCreateShader()` function and using `GL_GEOMETRY_SHADER` as the shader type, as follows:

```
glCreateShader(GL_GEOMETRY_SHADER);
```

Once the shader has been created, it is used like any other shader object. You give OpenGL your shader source code by calling `glShaderSource()`, compile the shader using the `glCompileShader()` function, and attach it to a program object by calling the `glAttachShader()` function. Then the program is linked as normal using the `glLinkProgram()` function. Now that you have a program object with a geometry shader linked into it, when you draw geometry using a function like `glDrawArrays()`, the vertex shader will run once per vertex, the geometry shader will run once per primitive (point, line, or triangle), and the fragment will run once per fragment. The primitives received by a geometry shader must match what it is expecting based in its own input primitive mode. When tessellation is not active, the primitive mode you use in your drawing commands must match the input primitive mode of the geometry shader. For example, if the geometry shader's input primitive mode is points, then you may only use `GL_POINTS` when you call `glDrawArrays()`. If the geometry shader's input primitive mode is triangles, then you may use `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, or `GL_TRIANGLE_FAN` in your `glDrawArrays()` call. A complete list of the geometry shader input primitive modes and the allowed geometry types is given in Table 8.1.

Table 8.1: Allowed Draw Modes for Geometry Shader Input Modes

Geometry Shader Input Mode	Allowed Draw Modes
<code>points</code>	<code>GL_POINTS</code>
<code>lines</code>	<code>GL_LINES</code> , <code>GL_LINE_LOOP</code> , <code>GL_LINE_STRIP</code>
<code>triangles</code>	<code>GL_TRIANGLES</code> , <code>GL_TRIANGLE_FAN</code> , <code>GL_TRIANGLE_STRIP</code>
<code>lines_adjacency</code>	<code>GL_LINES_ADJACENCY</code>
<code>triangles_adjacency</code>	<code>GL_TRIANGLES_ADJACENCY</code>

When tessellation is active, the mode you use in your drawing commands should always be `GL_PATCHES`, and OpenGL will convert the patches into points, lines, or triangles during the tessellation process. In this case, the input primitive mode of the geometry shader should match the tessellation primitive mode. The input primitive type is specified in the body of the geometry shader using a layout qualifier. The general form of the input layout qualifier is

```
layout (primitive_type) in;
```

This specifies that `primitive_type` is the input primitive type that the geometry shader is expected to handle, and `primitive_type` must be one of the supported primitive modes: `points`, `lines`, `triangles`, `lines_adjacency`, or `triangles_adjacency`. The geometry shader runs once per primitive. This means that it'll run once per point for `GL_POINTS`; once per line for `GL_LINES`, `GL_LINE_STRIP`, and `GL_LINE_LOOP`; and once per triangle for `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, and `GL_TRIANGLE_FAN`. The inputs to the geometry shader are presented in arrays containing all of the vertices making up the input primitive. The predefined inputs are stored in a built-in array called `gl_in[]`, which is an array of structures defined in Listing 8.19.

```
in gl_PerVertex
{
    vec4   gl_Position;
    float  gl_PointSize;
    float  gl_ClipDistance[];
} gl_in[];
```

Listing 8.19: The definition of `gl_in[]`

The members of this structure are the built-in variables that are written in the vertex shader: `gl_Position`, `gl_PointSize`, and `gl_ClipDistance[]`. You should recognize this structure from its declaration as an output block in the vertex shader described earlier in this chapter. These variables appear as global variables in the vertex shader because the block doesn't have an instance name there, but their values end up in the `gl_in[]` array of block instances when they appear in the geometry shader. Other variables written by the vertex shader also become arrays in the geometry shader. In the case of individual varyings, outputs in the vertex shader are declared as normal, and the inputs to the geometry shader have a similar declaration, except that they are arrays. Consider a vertex shader that defines outputs as

```
out vec4 color;
out vec3 normal;
```

The corresponding input to the geometry shader would be

```
in vec4 color[];
in vec3 normal[];
```

Notice that both the color and normal varyings have become arrays in the geometry shader. If you have a large amount of data to pass from the vertex to the geometry shader, it can be convenient to wrap per-vertex information passed from the vertex shader to the geometry shader into an interface block. In this case, your vertex shader will have a definition like this:

```
out VertexData
{
    vec4 color;
    vec3 normal;
} vertex;
```

And the corresponding input to the geometry shader would look like this:

```
in VertexData
{
    vec4 color;
    vec3 normal;
    // More per-vertex attributes can be inserted here
} vertex[];
```

With this declaration, you'll be able to access the per-vertex data in the geometry shader using `vertex[n].color` and so on. The length of the input arrays in the geometry shader depends on the type of primitives that it will process. For example, points are formed from a single vertex, and so the arrays will only contain a single element, whereas triangles are formed from three vertices, and so the arrays will be three elements long. If you're writing a geometry shader that's designed specifically to process a particular primitive type, you can explicitly size your input arrays, which provides a small amount of additional compile-time error checking. Otherwise, you can let your arrays be automatically sized by the input primitive type layout qualifier. A complete mapping of the input primitive modes and the resulting size of the input arrays is shown in Table 8.2.

Table 8.2: Sizes of Input Arrays to Geometry Shaders

Input Primitive Type	Size of Input Arrays
<code>points</code>	1
<code>lines</code>	2
<code>triangles</code>	3
<code>lines_adjacency</code>	4
<code>triangles_adjacency</code>	6

You also need to specify the primitive type that will be generated by the geometry shader. Again, this is determined using a layout qualifier, like so:

```
layout (primitive_type) out;
```

This is similar to the input primitive type layout qualifier, the only difference being that you are declaring the output of the shader using the `out` keyword. The allowable output primitive types from the geometry shader are `points`, `line_strip`, and `triangle_strip`. Notice that geometry shaders only support outputting the strip primitive types (not counting points—obviously, there is no such thing as a point strip).

There is one final layout qualifier that must be used to configure the geometry shader. Because a geometry shader is capable of producing a variable amount of data per vertex, OpenGL must be told how much space to allocate for all that data by specifying the maximum number of vertices that the geometry shader is expected to produce. To do this, use the following layout qualifier:

```
layout (max_vertices = n) out;
```

This sets the maximum number of vertices that the geometry shader may produce to `n`. Because OpenGL may allocate buffer space to store intermediate results for each vertex, this should be the smallest number possible that still allows your application to run correctly. For example, if you are planning to take points and produce one line at a time, then you can safely set this to two. This gives the shader hardware the best opportunity to run fast. If you are going to heavily tessellate the incoming geometry, you might want to set this to a much higher number, although this may cost you some performance. The upper limit on the number of vertices that a geometry shader can produce depends on your OpenGL implementation. It is guaranteed to be at least 256, but the absolute maximum can be found by calling `glGetIntegerv()` with the `GL_MAX_GEOMETRY_OUTPUT_VERTICES` parameter.

You can also declare more than one layout qualifier with a single statement by separating them with a comma, like so:

```
layout (triangle_strip, max_vertices = n) out;
```

With these layout qualifiers, a boilerplate `#version` declaration, and an empty `main()` function, you should be able to produce a geometry shader that compiles and links but does absolutely nothing. In fact, it will discard any geometry you send it, and nothing will be drawn by your application. We need to introduce two important functions: `EmitVertex()` and `EndPrimitive()`. If you don't call these, nothing will be drawn.

`EmitVertex()` tells the geometry shader that you've finished filling in all of the information for this vertex. Setting up the vertex works much like the vertex shader. You need to write into the built-in variable `gl_Position`. This sets the clip-space coordinates of the vertex that is produced by the geometry shader, just like in a vertex shader. Any other attributes that you want to pass from the geometry shader to the fragment shader can be declared in an interface block or as global variables in the geometry shader. Whenever you call `EmitVertex`, the geometry shader stores the values currently in all of its output variables and uses them to generate a new vertex. You can call `EmitVertex()` as many times as you like in a geometry shader, until you reach the limit you specified in your `max_vertices` layout qualifier. Each time, you put new values into your output variables to generate a new vertex.

An important thing to note about `EmitVertex()` is that it makes the values of any of your output variables (such as `gl_Position`) undefined. So, for example, if you want to emit a triangle with a single color, you need to write that color with every one of your vertices; otherwise, you will end up with undefined results.

`EmitPrimitive()` indicates that you have finished appending vertices to the end of the primitive. Don't forget, geometry shaders only support the strip primitive types (`line_strip` and `triangle_strip`). If your output primitive type is `triangle_strip` and you call `EmitVertex()` more than three times, the geometry shader will produce multiple triangles in a strip. Likewise, if your output primitive type is `line_strip` and you call `EmitVertex()` more than twice, you'll get multiple lines. In the geometry shader, `EndPrimitive()` refers to the strip. This means that if you want to draw individual lines or triangles, you have to call `EndPrimitive()` after every two or three vertices. You can also draw multiple strips by calling `EmitVertex()` many times between multiple calls to `EndPrimitive()`.

One final thing to note about calling `EmitVertex()` and `EndPrimitive()` in the geometry shader is that if you haven't produced enough vertices to produce a single primitive (e.g., you're generating `triangle_strip` outputs and you call `EndPrimitive()` after two vertices), nothing is produced for that primitive, and the vertices you've already produced are simply thrown away.

Discarding Geometry in the Geometry Shader

The geometry shader in your program runs once per primitive. What you do with that primitive is entirely up to you. The two functions

`EmitVertex()` and `EndPrimitive()` allow you to programmatically append new vertices to your triangle or line strip and to start new strips. You can call them as many times as you want (until you reach the maximum defined by your implementation). You're also allowed to not call them at all. This allows you to clip geometry away and discard primitives. If your geometry shader runs and you never call `EmitVertex()` for that particular primitive, nothing will be drawn. To illustrate this, we can implement a custom backface culling routine that culls geometry as if it were viewed from an arbitrary point in space. This is implemented in the `gsculling` example.

First, we set up our shader version and declare our geometry shader to accept triangles and to produce triangle strips. Backface culling doesn't really make a lot of sense for lines or points. We also define a uniform that will hold our custom viewpoint in world space. This is shown in Listing 8.20.

```
#version 330

// Input is triangles, output is triangle strip. Because we're going
// to do a 1 in 1 out shader producing a single triangle output for
// each one input, max_vertices can be 3 here.
layout (triangles) in;
layout (triangle_strip, max_vertices=3) out;

// Uniform variables that will hold our custom viewpoint and
// model-view matrix
uniform vec3 viewpoint;
uniform mat4 mv_matrix;
```

Listing 8.20: Configuring the custom culling geometry shader

Now inside our `main()` function, we need to find the face normal for the triangle. This is simply the cross products of any two vectors in the plane of the triangle—we can use the triangle edges for this. Listing 8.21 shows how this is done.

```
// Calculate two vectors in the plane of the input triangle
vec3 ab = gl_in[1].gl_Position.xyz - gl_in[0].gl_Position.xyz;
vec3 ac = gl_in[2].gl_Position.xyz - gl_in[0].gl_Position.xyz;
vec3 normal = normalize(cross(ab, ac));
```

Listing 8.21: Finding a face normal in a geometry shader

Now that we have the normal, we can determine whether it faces toward or away from our user-defined viewpoint. To do this, we need to transform the normal into the same coordinate space as the viewpoint, which is

world space. Assuming we have the model-view matrix in a uniform, simply multiply the normal by this matrix. To be more accurate, we should multiply the vector by the inverse of the transpose of the upper-left 3×3 submatrix of the model-view matrix. This is known as the normal matrix, and you're free to implement this and put it in its own uniform if you like. However, if your model-view matrix only contains translation, uniform scale (no shear), and rotation, you can use it directly. Don't forget, the normal is a three-element vector, and the model-view matrix is a 4×4 matrix. We need to extend the normal to a four-element vector before we can multiply the two. We can then take the dot product of the resulting vector with the vector from the viewpoint to any point on the triangle.

If the sign of the dot product is negative, that means that the normal is facing away from the viewer and the triangle should be culled. If it is positive, the triangle's normal is pointing toward the viewer, and we should pass the triangle on. The code to transform the face normal, perform the dot product, and test the sign of the result is shown in Listing 8.22.

```
// Calculate the transformed face normal and the view direction vector
vec3 transformed_normal = (vec4(normal, 0.0) * mv_matrix).xyz;
vec3 vt = normalize(gl_in[0].gl_Position.xyz - viewpoint);

// Take the dot product of the normal with the view direction
float d = dot(vt, normal);

// Emit a primitive only if the sign of the dot product is positive
if (d > 0.0)
{
    for (int i = 0; i < 3; i++)
    {
        gl_Position = gl_in[i].gl_Position;
        EmitVertex();
    }
    EndPrimitive();
}
```

Listing 8.22: Conditionally emitting geometry in a geometry shader

In Listing 8.22, if the dot product is positive, we copy the input vertices to the output of the geometry shader and call `EmitVertex()` for each one. If the dot product is negative, we simply don't do anything at all. This results in the incoming triangle being discarded altogether and nothing being drawn.

In this particular example, we are generating at most one triangle output for each triangle input to the geometry shader. Although the output of the

geometry shader is a triangle strip, our strips only contain a single triangle. Therefore, there doesn't strictly need to be a call to `EndPrimitive()`. We just leave it there for completeness.

Figure 8.16 shows a the result of this shader.



Figure 8.16: Geometry culled from different viewpoints

In Figure 8.16, the virtual viewer has been moved to different positions. As you can see, different parts of the model have been culled away by the geometry shader. It's not expected that this example is particularly useful, but it does demonstrate the ability for a geometry shader to perform geometry culling based on application-defined criteria.

Modifying Geometry in the Geometry Shader

The previous example either discarded geometry or passed it through unmodified. It is also possible to modify vertices as they pass through the geometry shader to create new, derived shapes. Even though your geometry shader is passing vertices on one-to-one (i.e., no amplification or culling is taking place), this still allows you to do things that would otherwise not be possible with a vertex shader alone. If the input geometry is in the form of triangle strips or fans, for example, the resulting geometry will have shared vertices and shared edges. Using the vertex shader to move shared vertices will move all of the triangles that share that vertex. It is not possible, then, to separate two triangles that share an edge in the original geometry using the vertex shader alone. However, this is trivial using the geometry shader.

Consider a geometry shader that accepts triangles and produces `triangle_strip` as output. The input to a geometry shader that accepts triangles is individual triangles, regardless of whether they originated

from a `glDrawArrays()` or a `glDrawElements()` function call, or whether the primitive type was `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, or `GL_TRIANGLE_FAN`. Unless the geometry shader outputs more than three vertices, the result is independent, unconnected triangles.

In this next example, we “explode” a model by pushing all of the triangles out along their face normals. It doesn’t matter whether the original model is drawn with individual triangles or with triangle strips or fans. As with the previous example, the input is triangles, the output is `triangle_strip`, and the maximum number of vertices produced by the geometry shader is three because we’re not amplifying or decimating geometry. The setup code for this is shown in Listing 8.23.

```
#version 330

// Input is triangles, output is triangle strip. Because we’re going to do a
// 1 in 1 out shader producing a single triangle output for each one input,
// max_vertices can be 3 here.
layout (triangles) in;
layout (triangle_strip, max_vertices=3) out;
```

Listing 8.23: Setting up the “explode” geometry shader

To project the triangle outward, we need to calculate the face normal of each triangle. Again, to do this we can take the cross product of two vectors in the plane of the triangle—two edges of the triangle. For this, we can reuse the code from Listing 8.21. Now that we have the triangle’s face normal, we can project vertices along that normal by an application-controlled amount. That amount can be stored in a uniform (we call it `explode_factor`) and updated by the application. This simple code is shown in Listing 8.24.

```
for (int i = 0; i < 3; i++)
{
    gl_Position = gl_in[i].gl_Position +
                  vec4(explode_factor * normal, 0.0);
}
```

Listing 8.24: Pushing a face out along its normal

The result of running this geometry shader on a model is shown in Figure 8.17. The model has been deconstructed, and the individual triangles have become visible.



Figure 8.17: Exploding a model using the geometry shader

Generating Geometry in the Geometry Shader

Just as you are not required to call `EmitVertex()` or `EndPrimitive()` at all if you don't want to produce any output from the geometry shader, it is also possible to call `EmitVertex()` and `EndPrimitive()` as many times as you need to produce new geometry. That is, until you reach the maximum number of output vertices that you declared at the start of your geometry shader. This functionality can be used for things like making multiple copies of the input or breaking the input into smaller pieces. This is the subject of the next example, which is the `gstessellate` sample in the book's accompanying source code. The input to our shader is a tetrahedron centered around the origin. Each face of the tetrahedron is made from a single triangle. We tessellate incoming triangles by producing new vertices halfway along each edge and then moving all of the resulting vertices so that they are variable distances from the origin. This transforms our tetrahedron into a spiked shape.

Because the geometry shader operates in object space (remember, the tetrahedron's vertices are centered around the origin), we need to do no coordinate transforms in the vertex shader and, instead, do the transforms in the geometry shader after we've generated the new vertices. To do this, we need a simple, pass-through vertex shader. Listing 8.25 shows a simple pass-through vertex shader.

```
#version 330

in vec4 position;

void main(void)
{
    gl_Position = position;
}
```

Listing 8.25: Pass-through vertex shader

This shader only passes the vertex position to the geometry shader. If you have other attributes associated with the vertices such as texture coordinates or normals, you need to pass them through the vertex shader to the geometry shader as well.

As in the previous example, we accept triangles as input to the geometry shader and produce a triangle strip. We break the strip after every triangle so that we can produce separate, independent triangles. In this example, we produce four output triangles for every input triangle. We need to declare our maximum output vertex count as 12—four triangles times three vertices. We also need to declare a uniform matrix to store the model-view transformation matrix in the geometry shader because we do that transform after generating vertices. Listing 8.26 shows this code.

```
#version 430 core

layout (triangles) in;
layout (triangle_strip, max_vertices = 12) out;

// A uniform to store the model-view-projection matrix
uniform mat4 mvp;
```

Listing 8.26: Setting up the “tessellator” geometry shader

First, let’s copy the incoming vertex coordinates into a local variable. Then, given the original, incoming vertices, we find the midpoint of each edge by taking their average. In this case, however, rather than simply dividing by two, we multiply by a scale factor, which will allow us to alter the *spikiness* of the resulting object. Code to do this is shown in Listing 8.27.

```
// Copy the incoming vertex positions into some local variables
vec3 a = gl_in[0].gl_Position.xyz;
vec3 b = gl_in[1].gl_Position.xyz;
vec3 c = gl_in[2].gl_Position.xyz;
```

```

// Find a scaled version of their midpoints
vec3 d = (a + b) * stretch;
vec3 e = (b + c) * stretch;
vec3 f = (c + a) * stretch;

// Now, scale the original vertices by an inverse of the midpoint
// scale
a *= (2.0 - stretch);
b *= (2.0 - stretch);
c *= (2.0 - stretch);

```

Listing 8.27: Generating new vertices in a geometry shader

Because we are going to generate several triangles using almost identical code, we can put that code into a function (shown in Listing 8.28) and call it from our main tessellation function.

```

void make_face(vec3 a, vec3 b, vec3 c)
{
    vec3 face_normal = normalize(cross(c - a, c - b));
    vec4 face_color = vec4(1.0, 0.2, 0.4, 1.0) * (mat3(mvMatrix) * face_normal);
    gl_Position = mvpMatrix * vec4(a, 1.0);
    color = face_color;
    EmitVertex();

    gl_Position = mvpMatrix * vec4(b, 1.0);
    color = face_color;
    EmitVertex();

    gl_Position = mvpMatrix * vec4(c, 1.0);
    color = face_color;
    EmitVertex();

    EndPrimitive();
}

```

Listing 8.28: Emitting a single triangle from a geometry shader

Notice that the `make_face` function calculates a face color based on the face's normal in addition to emitting the positions of its vertices. Now, we simply call `make_face` four times from our main function, which is shown in Listing 8.29.

```

make_face(a, d, f);
make_face(d, b, e);
make_face(e, c, f);
make_face(d, e, f);

```

Listing 8.29: Using a function to produce faces in a geometry shader

Figure 8.18 shows the result of our simple geometry shader-based tessellation program.

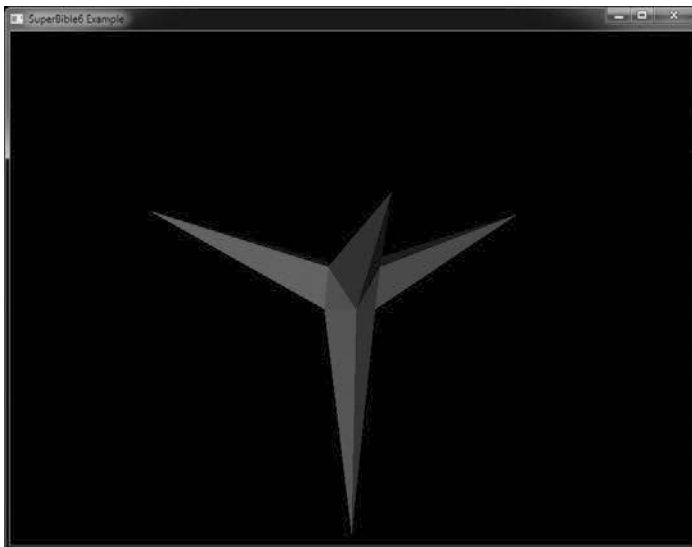


Figure 8.18: Basic tessellation using the geometry shader

Note that using the geometry shader for heavy tessellation may not produce the most optimal performance. If something more complex than that shown in this example is desired, it's best to use the hardware tessellation functions of OpenGL. However, if simple amplification of between two and four output primitives for each input primitive is desired, the geometry shader is probably the way to go.

Changing the Primitive Type in the Geometry Shader

So far, all of the geometry shader examples we've gone through have taken triangles as input and produced triangle strips as output. This doesn't change the geometry type. However, geometry shaders can input and output different types of geometry. For example, you can transform points into triangles or triangles into points. In the `normalviewer` example, which we'll describe next, we're going to change the geometry type from triangles to lines. For each vertex input to the shader, we take the vertex normal and represent it as a line. We also take the face normal and represent that as another line. This allows us to visualize the model's normals—both at each vertex and for each face. Note, though, that if you want to draw the normals on top of the original model, you need to draw everything twice—once with the geometry shader to visualize the normals and once without the geometry shader to show the model. You can't output a mix of two different primitives from a single geometry shader.

For our geometry shader, in addition to the members of the `gl_in` structure, we need the per-vertex normal, and that will have to be passed through the vertex shader. An updated version of the pass-through vertex shader from Listing 8.25 is given in Listing 8.30.

```
#version 330

in vec4 position;
in vec3 normal;

out Vertex
{
    vec3 normal;
} vertex;

void main(void)
{
    gl_Position = position;
    vertex.normal = normal;
}
```

Listing 8.30: A pass-through vertex shader that includes normals

This passes the position attribute straight through to the `gl_Position` built-in variable and places the normal into an output block.

The setup code for the geometry shader is shown in Listing 8.31. In this example, we accept triangles and produce line strips, each of a single line. Because we output a separate line for each normal we visualize, we produce two vertices for each vertex consumed, plus two more for the face normal. Therefore, the maximum number of vertices that we output per input triangle is eight. To match the Vertex output block that we declared in the vertex shader, we also need to declare a corresponding input interface block in the geometry shader. As we're going to do the object-space-to-world-space transformation in the geometry shader, we declare a `mat4` uniform called `mvp` to represent the model-view-projection matrix. This is necessary so that we can keep the vertex's position in the same coordinate system as its normal until we produce the new vertices representing the line.

```
#version 330

layout (triangles) in;
layout (line_strip) out;
layout (max_vertices = 8) out;

in Vertex
{
    vec3 normal;
} vertex[];
```

```
// Uniform to hold the model-view-projection matrix
uniform mat4 mvp;

// Uniform to store the length of the visualized normals
uniform float normal_length;
```

Listing 8.31: Setting up the “normal visualizer” geometry shader

Each input vertex is transformed into its final position and emitted from the geometry shader, and then a second vertex is produced by displacing the input vertex along its normal and transforming that into its final position as well. This makes the length of all of our normals one but allows any scaling encoded in our model-view-projection matrix to be applied to them along with the model. We multiply the normals by the application-supplied uniform `normal_length`, allowing them to be scaled to match the model. Our inner loop is shown in Listing 8.32.

```
gl_Position = mvp * gl_in[0].gl_Position;
gs_out.normal = gs_in[0].normal;
gs_out.color = gs_in[0].color;
EmitVertex();

gl_Position = mvp * (gl_in[0].gl_Position +
                    vec4(gs_in[0].normal * normal_length, 0.0));
gs_out.normal = gs_in[0].normal;
gs_out.color = gs_in[0].color;
EmitVertex();
EndPrimitive();
```

Listing 8.32: Producing lines from normals in the geometry shader

This generates a short line segment at each vertex pointing in the direction of the normal. Now, we need to produce the face normal. To do this, we need to pick a suitable place from which to draw the normal, and we need to calculate the face normal itself in the geometry shader along which to draw the line.

As in the earlier example given in Listing 8.33, we use a cross product of two of the triangle’s edges to find the face normal. To pick a starting point for the line, we choose the centroid of the triangle, which is simply the average of the coordinates of the input vertices. Listing 8.33 shows the shader code.

```
vec3 ab = gl_in[1].gl_Position.xyz - gl_in[0].gl_Position.xyz;
vec3 ac = gl_in[2].gl_Position.xyz - gl_in[0].gl_Position.xyz;
vec3 face_normal = normalize(cross(ab, ac));

vec4 tri_centroid = (gl_in[0].gl_Position +
                    gl_in[1].gl_Position +
                    gl_in[2].gl_Position) / 3.0;
```

```

gl_Position = mvp * tri_centroid;
gs_out.normal = gs_in[0].normal;
gs_out.color = gs_in[0].color;
EmitVertex();

gl_Position = mvp * (tri_centroid +
                    vec4(face_normal * normal_length, 0.0));
gs_out.normal = gs_in[0].normal;
gs_out.color = gs_in[0].color;
EmitVertex();
EndPrimitive();

```

Listing 8.33: Drawing a face normal in the geometry shader

Now when we render a model, we get the image shown in Figure 8.19.

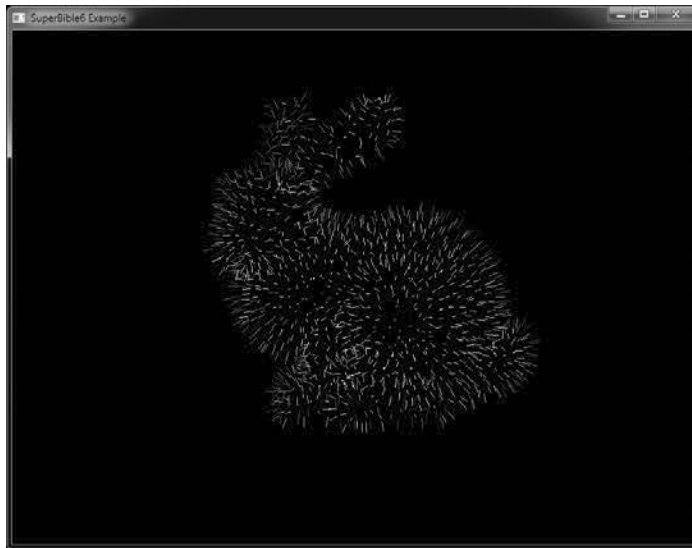


Figure 8.19: Displaying the normals of a model using a geometry shader

Multiple Streams of Storage

When only a vertex shader is present, there is a simple one-in, one-out relationship between the vertices coming into the shader and the vertices stored in the transform feedback buffer. When a geometry shader is present, each shader invocation may store zero, one, or more vertices into the bound transform feedback buffers. Not only this, but it's actually possible to configure up to four output *streams* and use the geometry shader to send its output to whichever one it chooses. This can be used, for example, to sort geometry or to render some primitives while storing other geometry in transform feedback buffers. There are a couple of pretty

major limitations when multiple output streams are used in a geometry shader; first, the output primitive mode from the geometry shader for all streams must be set to **points**. Second, although it's possible to simultaneously render geometry and to store data into transform feedback buffers, only the first stream may be rendered — the others are for storage only. If your application fits with these constraints, then this can be a very powerful feature.

To set up multiple output streams from your geometry shader, use the **stream** layout qualifier to select one of four streams. As with most other output layout qualifiers, the **stream** qualifier may be applied directly to a single output or to an output block. It can also be applied directly to the **out** keyword without declaring an output variable, in which case it will affect all further output declarations until another **stream** layout qualifier is encountered. For example, consider the following output declarations in a geometry shader:

```
out vec4          foo; // "foo" is in stream 0 (the default).
layout (stream=2) out vec4 bar; // "bar" is part of stream 2.
out vec4          baz; // "baz" is back in stream 0.
layout (stream=1) out;    // Everything from here on is in stream 1.
out int           apple; // "apple" and "orange" are part
out int           orange; // of stream 1.
layout (stream=3) out MY_BLOCK // Everything in "MY_BLOCK" is in
stream 3.
{
    vec3          purple;
    vec3          green;
};
```

In the geometry shader, when you call `EmitVertex()`, the vertex will be recorded into the first output stream (stream 0). Likewise, when you call `EndPrimitive()`, it will end the primitive being recorded to stream 0. However, you can call `EmitStreamVertex()` and `EndStreamPrimitive()`, both of which take an integer argument specifying the stream to send the output to:

```
void EmitStreamVertex(int stream);

void EndStreamPrimitive(int stream);
```

The **stream** argument must be a compile time constant. If rasterization is enabled, then any primitives sent to stream 0 will be rasterized.

New Primitive Types Introduced by the Geometry Shader

Four new primitive types were introduced with geometry shaders: `GL_LINES_ADJACENCY`, `GL_LINE_STRIP_ADJACENCY`,

`GL_TRIANGLES_ADJACENCY`, and `GL_TRIANGLE_STRIP_ADJACENCY`. These primitive types are really only useful when rendering with a geometry shader active. When the new adjacency primitive types are used, for each line or triangle passed into the geometry shader, it not only has access to the vertices defining that primitive, but it also has access to the vertices of the primitive that is next to the one it's processing.

When you render using `GL_LINES_ADJACENCY`, each line segment consumes four vertices from the enabled attribute arrays. The two center vertices make up the line; the first and last vertices are considered the adjacent vertices. The inputs to the geometry shader are therefore four-element arrays. In fact, because the input and output types of the geometry shader do not have to be related, `GL_LINES_ADJACENCY` can be seen as a way of sending generalized four-vertex primitives to the geometry shader. The geometry shader is free to transform them into whatever it pleases. For example, your geometry shader could convert each set of four vertices into a triangle strip made up of two triangles. This allows you to render quads using the `GL_LINES_ADJACENCY` primitive. It should be noted, though, that if you draw using `GL_LINES_ADJACENCY` when no geometry shader is active, regular lines will be drawn using the two innermost vertices of each set of four vertices. The two outermost vertices will be discarded, and the vertex shader will not run on them at all.

Using `GL_LINE_STRIP_ADJACENCY` produces a similar effect. The difference is that the entire strip is considered to be a primitive, with one additional vertex on each end. If you send eight vertices to OpenGL using `GL_LINES_ADJACENCY`, the geometry shader will run twice, whereas if you send the same vertices using `GL_LINE_STRIP_ADJACENCY`, the geometry shader will run five times. Figure 8.20 should make things clear. The eight vertices in the top row are sent to OpenGL with the `GL_LINES_ADJACENCY` primitive mode. The geometry shader runs twice on four vertices each time—ABCD and EFGH. In the second row, the same eight vertices are sent to OpenGL using the `GL_LINE_STRIP_ADJACENCY` primitive mode. This time, the geometry shader runs five times—ABCD, BCDE, and so on until EFGH. In each case, the solid arrows are the lines that would be rendered if no geometry shader were present.

The `GL_TRIANGLES_ADJACENCY` primitive mode works similarly to the `GL_LINES_ADJACENCY` mode. A triangle is sent to the geometry shader for each set of six vertices in the enabled attribute arrays. The first, third, and fifth vertices are considered to make up the real triangle, and the second, fourth, and sixth vertices are considered to be in between the triangle's

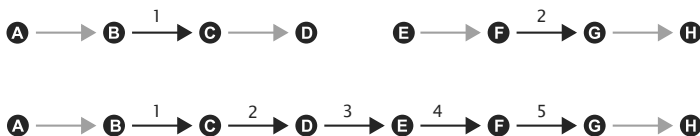


Figure 8.20: Lines produced using lines with adjacency primitives

vertices. This means that the inputs to the geometry shader are six-element arrays. As before, you can do anything you want to the vertices using the geometry shader; `GL_TRIANGLES_ADJACENCY` is a good way to get arbitrary six-vertex primitives into the geometry shader.

Figure 8.21 shows this.

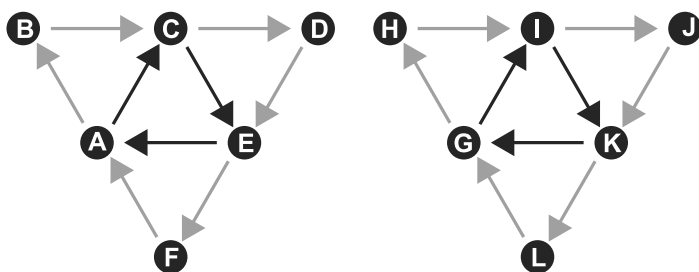


Figure 8.21: Triangles produced using `GL_TRIANGLES_ADJACENCY`

The final, and perhaps most complex (or alternatively the most difficult to understand), of these primitive types is `GL_TRIANGLE_STRIP_ADJACENCY`. This primitive represents a triangle strip with every other vertex (the first, third, fifth, seventh, ninth, and so on) forming the strip. The vertices in between are the adjacent vertices. Figure 8.22 demonstrates the principle. In the figure, the vertices A through P represent 16 vertices sent to OpenGL. A triangle strip is generated from every other vertex (A, C, E, G, I, and so on), and the vertices that come between them (B, D, F, H, J, and so on) are the adjacent vertices.

There are special cases for the triangles that come at the start and end of the strip, but once the strip is started, the vertices fall into a regular pattern that is more clearly seen in Figure 8.23.

The rules for the ordering of `GL_TRIANGLE_STRIP_ADJACENCY` are spelled out clearly in the OpenGL Specification—in particular, the special cases are noted there. You are encouraged to read that section of the specification if you want to work with this primitive type.

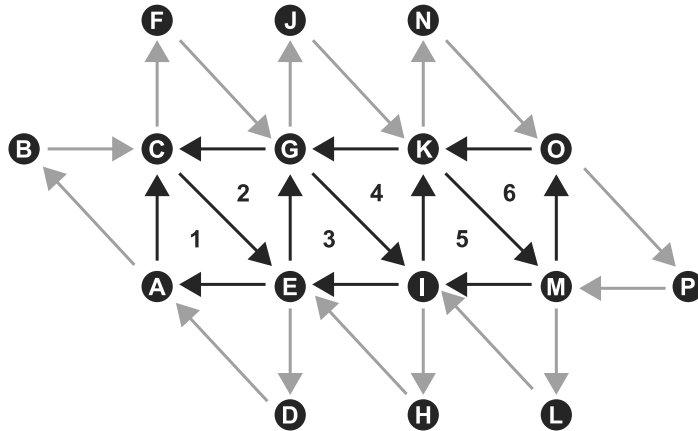


Figure 8.22: Triangles produced using `GL_TRIANGLE_STRIP_ADJACENCY`

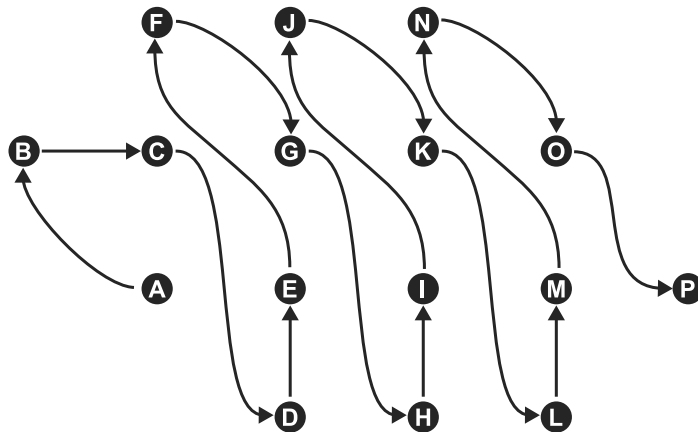


Figure 8.23: Ordering of vertices for `GL_TRIANGLE_STRIP_ADJACENCY`

Rendering Quads Using a Geometry Shader

In computer graphics, the word *quad* is used to describe a quadrilateral – a shape with four sides. Modern graphics APIs do not support rendering quads directly, primarily because modern graphics hardware does not support quads. When a modeling program produces an object made from quads, it will often include the option to export the geometry data by converting each quad into a pair of triangles. These are then rendered by the graphics hardware directly. In some graphics hardware, quads are supported, but internally the hardware will do this conversion from quads to pairs of triangles for you.

In many cases, breaking a quad into a pair of triangles works out just fine and the visual image isn't much different than what would have been rendered had native support for quads been present. However, there are a large class of cases where breaking a quad into a pair of triangles *doesn't* produce the correct result. Take a look at Figure 8.24.

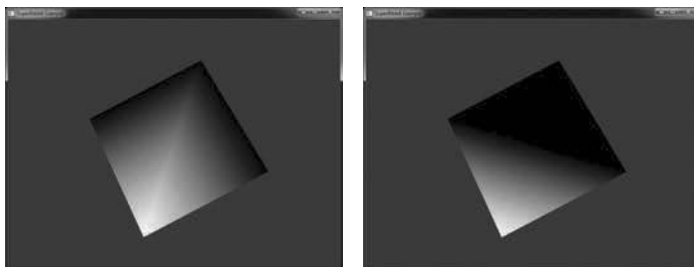


Figure 8.24: Rendering a quad using a pair of triangles

In Figure 8.24, we have rendered a quad as a pair of triangles. In both images, the vertices are wound in the same order. There are three black vertices and one white vertex. In the left image, the split between the triangles runs vertically through the quad. The topmost and two side vertices are black and the bottommost vertex is white. The seam between the two triangles is clearly visible as a bright line. In the right image, the quad has been split horizontally. This has produced the topmost triangle, which contains only black vertices and is therefore entirely black, and the bottommost triangle, which contains one white vertex and two black ones, therefore displaying a black to white gradient.

The reason for this is that during rasterization and interpolation of the per-vertex colors presented to the fragment shader, we're only rendering a triangle. There are only three vertices' worth of information available to us at any given time, and therefore, we can't take into consideration the "other" vertex in the quad.

Clearly, neither image is correct, but neither is obviously better than the other. Also, the two images are radically different. If we rely on our export tools, or worse a runtime library, to split quads for us, we do not have any control over which of these two images we'll get. What can we do about that? Well, the geometry shader is able to accept primitives with the `GL_LINES_ADJACENCY` type, and each of these has four vertices — exactly enough to represent a quad. This means that by using lines with adjacency, we can get four vertices' worth of information at least as far as the geometry shader.

Next, we need to deal with the rasterizer. Recall, the output of the geometry shader can only be points, lines, or triangles, and so the best we can do is to break each quad (represented by a `lines_adjacency` primitive) into a pair of triangles. You might think this leaves us in the same spot as we were before. However, we now have the advantage that we can pass whatever information we like on to the fragment shader.

To correctly render a quad, we must consider the parameterization of the domain over which we want to interpolate our colors (or any other attribute). For triangles, we use barycentric coordinates, which are three-dimensional coordinates used to weight the three corners of the triangle. However, for a quad, we can use a two-dimensional parameterization. Consider the quad shown in Figure 8.25.

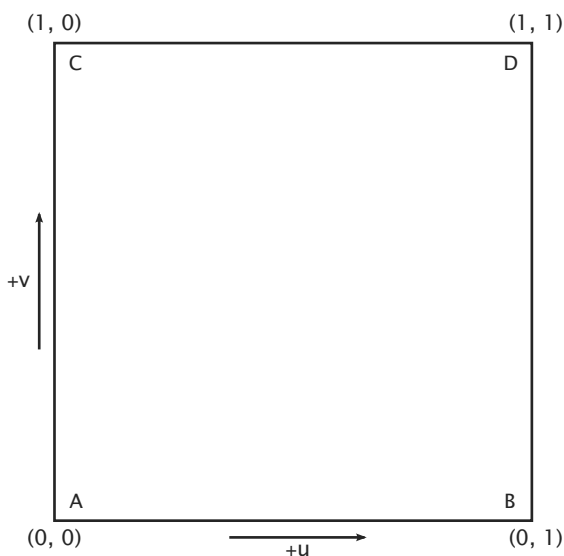


Figure 8.25: Parameterization of a quad

Domain parameterization of a quad is two-dimensional and can be represented as a two-dimensional vector. This can be smoothly interpolated over the quad to find the value of the vector at any point within it. For each of the quad's four vertices *A*, *B*, *C*, and *D*, the values of the vector will be $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$, respectively. We can generate these values per vertex in our geometry shader and pass them to the fragment shader.

To use this vector to retrieve the interpolated values of our other per-fragment attributes, we make the following observation: The value of

any interpolant will move smoothly between vertex *A* and *B* and between *C* and *D* with the *x* component of the vector. Likewise, a value along the edge *AB* will move smoothly to the corresponding value on edge *CD*. Thus, given the values of the attributes at the vertices *A* through *D*, we can use the domain parameter to interpolate a value of each attribute at any point inside the quad.

Thus, our geometry shader simply passes all four of the per-vertex attributes, unmodified, as `flat` outputs to the fragment shader, along with a smoothly varying domain parameter per vertex. The fragment shader then uses the domain parameter and *all four* per-vertex attributes to perform the interpolation directly.

The geometry shader is shown in Listing 8.34, and the fragment shader is shown in Listing 8.35 — both are taken from the `gsquads` example. Finally, the result of rendering the same geometry as shown in Figure 8.24 is shown in Figure 8.26.

```
#version 430 core

layout (lines_adjacency) in;
layout (triangle_strip, max_vertices = 6) out;

in VS_OUT
{
    vec4 color;
} gs_in[4];

out GS_OUT
{
    flat vec4 color[4];
    vec2 uv;
} gs_out;

void main(void)
{
    gl_Position = gl_in[0].gl_Position;
    gs_out.uv = vec2(0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[1].gl_Position;
    gs_out.uv = vec2(1.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[2].gl_Position;
    gs_out.uv = vec2(1.0, 1.0);

    // We're only writing the output color for the last
    // vertex here because they're flat attributes,
    // and the last vertex is the provoking vertex by default
    gs_out.color[0] = gs_in[1].color;
    gs_out.color[1] = gs_in[0].color;
    gs_out.color[2] = gs_in[2].color;
    gs_out.color[3] = gs_in[3].color;
    EmitVertex();
```

```

    EndPrimitive();

    gl_Position = gl_in[0].gl_Position;
    gs_out.uv = vec2(0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[2].gl_Position;
    gs_out.uv = vec2(1.0, 1.0);
    EmitVertex();

    gl_Position = gl_in[3].gl_Position;
    gs_out.uv = vec2(0.0, 1.0);

    // Again, only write the output color for the last vertex
    gs_out.color[0] = gs_in[1].color;
    gs_out.color[1] = gs_in[0].color;
    gs_out.color[2] = gs_in[2].color;
    gs_out.color[3] = gs_in[3].color;
    EmitVertex();

    EndPrimitive();
}

```

Listing 8.34: Geometry shader for rendering quads

```

#version 430 core

in GS_OUT
{
    flat vec4 color[4];
    vec2 uv;
} fs_in;

out vec4 color;

void main(void)
{
    vec4 c1 = mix(fs_in.color[0], fs_in.color[1], fs_in.uv.x);
    vec4 c2 = mix(fs_in.color[2], fs_in.color[3], fs_in.uv.x);

    color = mix(c1, c2, fs_in.uv.y);
}

```

Listing 8.35: Fragment shader for rendering quads

Multiple Viewport Transformations

You learned in “Viewport Transformation” back in Chapter 3 about the viewport transformation and how you can specify the rectangle of the window you’re rendering into by calling **glViewport()** and **glDepthRange()**. Normally, you would set the viewport dimensions to cover the entire window or screen, depending on whether your application is running on a desktop or is taking over the whole display. However, it’s possible to move the viewport around and draw into

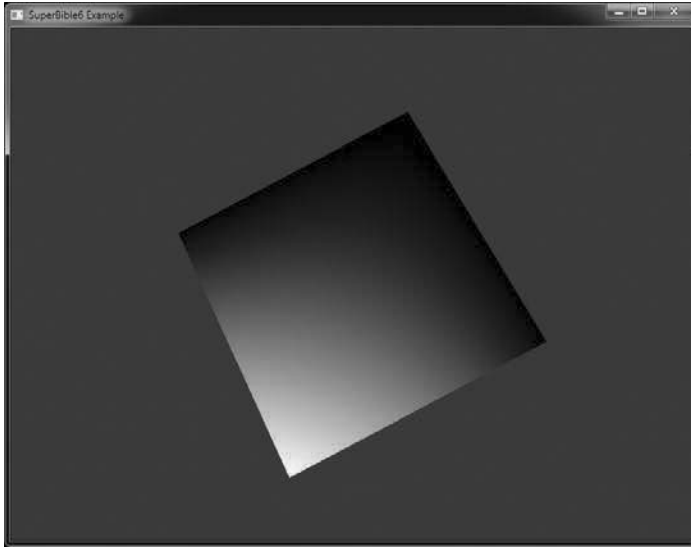


Figure 8.26: Quad rendered using a geometry shader

multiple virtual windows within a single larger framebuffer. Furthermore, OpenGL also allows you to use multiple viewports *at the same time*. This feature is known as viewport arrays.

To use a viewport array, we first need to tell OpenGL what the bounds of the viewports we want to use are. To do this, call **glViewportIndexedf()** or **glViewportIndexedfv()**, whose prototypes are

```
void glViewportIndexedf(GLuint index,
                        GLfloat x,
                        GLfloat y,
                        GLfloat w,
                        GLfloat h);

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

For both **glViewportIndexedf()** and **glViewportIndexedfv()**, *index* is the index of the viewport you wish to modify. Also notice that the viewport parameters to the indexed viewport commands are floating-point values rather than the integers used for **glViewport()**. OpenGL supports a minimum⁶ of 16 viewports, and so *index* can range from 0 to 15.

6. The actual number of viewports that are supported by OpenGL can be determined by querying the value of `GL_MAX_VIEWPORTS`.

Likewise, each viewport also has its own depth range, which can be specified by calling **glDepthRangeIndexed()**, whose prototype is

```
void glDepthRangeIndexed(GLuint index,
                        GLdouble n,
                        GLdouble f);
```

Again, index may be between 0 and 15. In fact, **glViewport()** really sets the extent of all of the viewports to the same range, and **glDepthRange()** sets the depth range of all viewports to the same range. If you want to set more than one or two of the viewports at a time, you might consider using **glViewportArrayv()** and **glDepthRangeArrayv()**, whose prototypes are

```
void glViewportArrayv(GLuint first,
                    GLsizei count,
                    const GLfloat * v);

void glDepthRangeArrayv(GLuint first,
                    GLsizei count,
                    const GLdouble * v);
```

These functions set either the viewport extents or depth range for count viewports starting with the viewport indexed by first to the parameters specified in the array v. For **glViewportArrayv()**, the array contains a sequence of x, y, width, height values, in that order. For **glDepthRangeArrayv()**, the array contains a sequence of n, f pairs, in that order.

Once you have specified your viewports, you need to direct geometry into them. This is done by using a geometry shader. Writing to the built-in variable `gl_ViewportIndex` selects the viewport to render into. Listing 8.36 shows what such a geometry shader might look like.

```
#version 430 core

layout (triangles, invocations = 4) in;
layout (triangle_strip, max_vertices = 3) out;

layout (std140, binding = 0) uniform transform_block
{
    mat4 mvp_matrix[4];
};

in VS_OUT
{
    vec4 color;
} gs_in[];

out GS_OUT
{
    vec4 color;
} gs_out;
```

```

void main(void)
{
    for (int i = 0; i < gl_in.length(); i++)
    {
        gs_out.color = gs_in[i].color;
        gl_Position = mvp_matrix[gl_InvocationID] *
                      gl_in[i].gl_Position;
        gl_ViewportIndex = gl_InvocationID;
        EmitVertex();
    }
    EndPrimitive();
}

```

Listing 8.36: Rendering to multiple viewports in a geometry shader

When the shader of Listing 8.36 executes, it produces four invocations of the shader. On each invocation, it sets the value of `gl_ViewportIndex` to the value of `gl_InvocationID`, directing the result of each of the geometry shader instances to a separate viewport. Also, for each invocation, it uses a separate model-view-projection matrix, which it retrieves from the uniform block, `transform_block`. Of course, a more complex shader could be constructed, but this is sufficient to demonstrate direction of transformed geometry into a number of different viewports. We have implemented this code in the `multipleviewport` sample, and the result of running this shader on our simple spinning cube is shown in Figure 8.27.

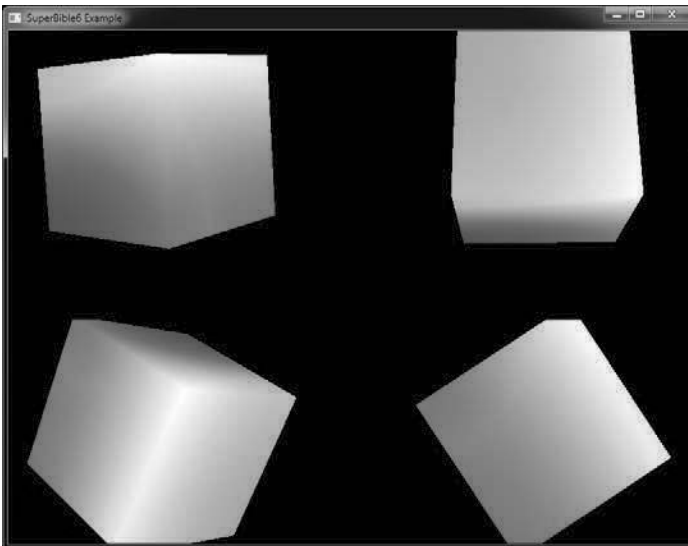


Figure 8.27: Result of rendering to multiple viewports

You can clearly see the four copies of the cube rendered by Listing 8.36 in Figure 8.27. Because each was rendered into its own viewport, it is clipped separately, and so where the cubes extend past the edges of their respective viewports, their corners are cut off by OpenGL's clipping stage.

Summary

In this chapter, you have read about the two tessellation shader stages, the fixed-function tessellation engine, and the way they interact. You have also read about geometry shaders and have seen how both the tessellator and the geometry shader can be used to change the amount of data in the OpenGL pipeline. You have also seen some of the additional functionality in OpenGL that can be accessed using tessellation and geometry shaders. You have seen how, conceptually, tessellation shaders and geometry shaders process vertices in groups — in the case of tessellation shaders, those groups forming *patches*, and in the case of geometry shaders, those groups forming traditional primitives such as lines and triangles. You've seen the special *adjacency* primitive types accessible to geometry shaders. After the geometry shader ends, primitives are eventually sent to the rasterizer and then to per-fragment operations, which will be the subject of the next chapter.

Index

- a 100 megapixel virtual framebuffer listing (9.23), 401
- 1D textures, 244
- 2D
 - array textures, loading, 163–165
 - Gaussian filters, 411
 - pixel formats, 630
 - prefix sums, 452
- 3D
 - Linux, 682
 - math with GLKit, 667–669
- abstraction layers, 4, 5
- acceleration
 - calculating, 269
 - structures, 579
- access
 - map buffer types, 601
 - synchronization
 - atomic counters, 137
 - images, 176–177
 - memory, 129–133
 - textures, arrays, 163–165
- adaptive HDR to LDR conversion fragment
 - shader listing (9.25), 407–408
- adding
 - basevertex, 234–235
 - bloom effect to scene listing (9.28), 414
 - device context parameters, 628
 - fog effects, 541–544
 - views, 650–652
- addresses, querying extension functions, 622
- adjacency primitive types, 340
- advanced framebuffer formats, 399–418
- advanced occlusion queries, 483–484
- Aero user interfaces, 592
- AFR (alternate frame rendering), 610
- algorithms
 - flocking, 462–471
 - prefix sum, 452
- aliasing, 140
- allocating memory using buffers, 92–95
- alpha-to-coverage, 392
- alternate frame rendering. *See* AFR
- alternative rendering methods, 548–580
- ALU (Arithmetic and Logic Unit) performance, 605
- ambient light, 504
- ambient occlusion, 558–565
 - fragment shader listing (12.32), 564–565
- AMD drivers, 625, 682
- a more complete conditional rendering
 - example listing (11.6), 482–483
- analysis
 - graphics processors, 594
 - performance analysis tools, 589–597
- AND operator, 47
- Android
 - development environments, 729–734
 - handheld platforms, 729
- angles, Euler, 72
- animating color over time listing (2.2), 16
- antialiasing, 384–399
 - by filtering, 385–387
 - multi-sample, 387–389
- AoSs (array-of-structures), 102
- APIENTRY macro, 583
- APIs (Application Programming Interfaces), 3,
7. *See also* interfaces
 - trace tools, 594
 - Windows, 623

- application of fog in a fragment shader listing (12.24), 543–544
- Application Programming Interfaces. *See* APIs
- applications
 - barriers, 131–132
 - cleaning up, 646–647
 - Cocoa, 650
 - debugging, 582–589
 - design (OpenGL ES 3.0), 714
 - frameworks, 14–16
 - geometry shaders, 313–317
 - Linux, 687–693
 - loading textures from files, 144
 - performance optimization, 589–616
 - shaders, 5
 - starting, 21
 - tuning for speed, 597–616
- applying
 - arrow keys, 671–673
 - contexts, 699–701
 - extensions, 618–622
 - simple exposure coefficient to an HDR image
 - listing (9.24), 406
- ARB (Architectural Review Board), 7, 8, 618
- areas, signed, 40
- Arithmetic and Logic Unit performance. *See* ALU performance
- array-of-structures (AoSs), 102
- arrays, 15, 192–194
 - accessing, 163–165
 - allocating, 638
 - indexes, 245
 - instanced, 288
 - multi-dimensional, 194
 - sizes of input, 315
 - textures, 160–165
 - VAOs (vertex array objects), 272
- arrow keys, 671–673
- assigning binding points, 117
- associative, 62
- asteroids
 - configuring, 254
 - field vertex shader listing (7.14), 255–257
 - rendering, 257
 - vertex shader inputs for, 254
- atmospheric effects, 540–544
- atomic counters, 133–137
- atomic operations, 126, 128–129, 171–176
- attachments
 - completeness, 377
 - multiple framebuffer, 368–370
 - rendering with no, 399–401
 - texture layers to a framebuffer
 - listing (9.12), 375
 - textures, 367
- attenuation, distance-based point size, 230
- attributes, 51
 - Cocoa pixel format, 655–658
 - configs, 690
 - instancing, 245
 - pixel format, 636–637
 - vertices, 28–29, 97, 224
- averaging values, 457
- axes, coordinates, 68
- back buffers, 365
- back end processes, 11, 341
- back-facing, 39
- back-lighting, 515–517
- bandwidth, memory, 178, 549
- barriers, 446
 - applications, 131–132
 - shaders, 132–133
- barycentric coordinates, 35, 284–285, 288
- baseinstance parameter, 239
- basevertex parameter, 234–235, 239, 240
- basic conditional rendering example listing (11.5), 481
- basic setup of Windows operating systems, 627–632
- Bézier curves, 85, 86, 87
- big-picture views, 11
- binaries, programs, 216–218, 609
- binding, 92
 - buffers, 261
 - framebuffers, 366
 - points, 117, 262
- Bishop, Lara, 718
- Bit-Level-Image-Transfer, 432
- bittangent vectors, 519
- blending, 357–363
 - blend equations, 358, 361–362
 - color, 406
 - dual-source, 361
 - factor, 406
 - functions, 358–360
- Blinn-Phong fragment shader listing (12.5), 514
- Blinn-Phong lighting model, 513–515
- blit, 431
- Block Partitioned Texture Compression (BPTC), 179
- blocks
 - interfaces, 31–32
 - shaders, storage, 126–133
 - uniforms, 108–121
- Block Transfer, 432
- bloom, light, 409–414
- bloom fragment shader listing (9.26), 410–411
- blur fragment shader listing (9.27), 412–413
- boids, 449
- Boolean flags, 655
- Boolean occlusion queries, 483
- Boolean vectors, 196
- border color, texture, 159
- BPTC (Block Partitioned Texture Compression), 179

- brute force, 579
- bubble, 494
- buffers, 10, 92–95
 - asteroids, 254
 - back, 365
 - binding, 261
 - command, 590
 - data
 - allocating memory using, 92–95
 - feeding vertex shaders from, 97–103
 - filling and copying in, 95–97
 - depth, 46
 - double buffering, 634, 661
 - element array, 279
 - EGL, 726–727
 - G-buffers, 548, 549–551
 - mapping, 600–603
 - object storage, 251
 - point indexes, 228
 - swap values, 637
 - TBO (texture buffer object), 266, 269
 - textures, 140
 - UBO (Uniform Buffer Object), 108
- building. *See also* configuration
 - Linux applications, 687–693
 - model-view matrices listing (5.21), 122
- built-in functions, 194–201
- built-in outputs, 441
- built-in variables, 24
 - gl_InstanceID, 288
 - gl_FrontFacing, 223
 - gl_Position, 277
 - gl_VertexID, 92
 - gl_int, 146
 - fragment, 43
 - tessellation, 35
- bump mapping, 518
- C++, 737
- calculations. *See also* math
 - acceleration, 269
 - antialiasing, 385
 - colors, fragments, 152
 - contributions to ambient light, 504
 - damping force, 270
 - dot products, 54–55
 - formulas, indexes, 250
 - G-buffers, 552
 - lighting models. *See* lighting models
 - orientation, 257
 - per-fragment lighting, 518
 - per-instance rotations, 163
 - reflection and refraction, 57–58
 - shadow maps, 539
 - toon shaders, 547
- callback functions, 583, 584. *See also* functions
- camera space, 64, 65
- Cartesian frames, 519
- casting shadows, 534–540
- Cathode Ray Tubes (CRTs), 416
- cell shading, 545–547
- centroid sampling, 395–399
- CGL (Core OpenGL), 648, 674–675
 - specifications, 625n3
- checking completeness of a framebuffer object
 - listing (9.13), 378–379
- child windows (in Cocoa), 671
- choosing. *See also* selecting
 - 8 sample antialiasing listing (9.19), 388
 - and setting a pixel format listing (14.4), 632
- chunks, SBM model file format, 752–756
- clamping
 - depth, 354–355
 - tone mapping, 406
- classes, 628
 - GLKit, 662
 - GLKTextureInfo, 663
 - GLKViewController, 741
 - textures, 183
- cleaning up applications, 646–647
- clipping, 38–39, 276–282
 - an object against a plane and a sphere listing (7.20), 281
 - lines, 276
- clip spaces, 17, 38, 64, 66
- Cocoa
 - GLKit, 669–671
 - Mac OS X, 649–662
 - pixel format attributes, 655–658
- code
 - called when the view changes size listing (14.14), 661
 - errors, 584
- colors
 - calculating, 257
 - grass, 244
 - inputs, vertex shaders, 225
 - masking, 363–364
 - OpenGL ES 3.0, 709
 - output, 357–364
 - sRGB color spaces, 416–418
 - tone mapping, 404
- columns
 - column major, 60
 - column primary, 60
 - images, 454
 - layouts, 60
- combining geometry and primitive restart, 235–237
- commands
 - buffers, 590
 - drawing, 231–259, 595
 - clipping, 276–282
 - indexed, 231–237
 - indirect draws, 250–259
 - instancing, 237–250

- commands (*continued*)
 - stencil buffers, 348
 - storing transformed vertices, 259–275
- glxinfo, 684
- SwapBuffers(), 593
- synchronization, 699
- comments, chunks, 755
- communication
 - compute shaders, 444–449
 - between shader invocations, 299
- commutativity, 110
- comparison operators, 352
- compatibility profiles, 9
- compiling
 - makefiles, 687
 - programs, 201–219
 - shaders, 218, 606–609
 - simple shaders listing (2.5), 18–19
- completeness
 - attachments, 377
 - framebuffers, 377
 - whole framebuffer, 377
- complex number, 75
- complex shader, 339
- compressing textures, 177–181, 606
- compute shaders, 47–48, 437–472
 - applying, 438–439
 - communication, 444–449
 - examples, 450–471
 - executing, 439–444
 - flocking, 462–471
 - to generate a 2D prefix sum listing (10.7), 455–456
 - image inversion listing (10.2), 444
 - parallel prefix sum, 450–462
 - with race conditions listing (10.4), 447
 - synchronizing, 445–449
 - for updates in flocking example listing (10.11), 466
- concatenation
 - model-view transformations, 76–79
 - transformations, 73–75
- concave polygons, 10
- conditions
 - conditionally emitting geometry in a geometry shader listing (8.22), 319
 - conditional rendering, 481, 598
 - race, 446, 447
- configs
 - EGL, 720–725
 - management and visuals, 689–693
- configuration
 - Android projects, 730–731
 - asteroids, 270–271
 - comparison operators, 352
 - cubes, geometry, 233
 - the custom culling geometry shader listing (8.20), 318
 - GL3W, 686
 - GLFW, 686
 - iOS projects, 734–736
 - Mesa, 685
 - scalars, 105–106
 - separable program pipelines listing (6.3), 208
 - uniforms
 - arrays, 106–107
 - matrices, 107–108
 - Windows operating systems, 627–632
- connecting vertices, 267
- construction
 - and initialization of the GLKView listing (14.20), 738
 - matrices, 60–63
- consuming G-buffers, 551–554
- container objects, 603
- contention, 129
- contexts
 - advanced creation, 641–643
 - applying, 699–701
 - current, 611
 - debug, 582–589
 - devices, 627, 629
 - managing, 695–699, 725–726
- controlling
 - movement smoothly with keyboard bit flags and a timer listing (14.15), 672–673
 - winding order, 296
- control points, 83, 284, 324
- control shaders, tessellation, 33–34
- coordinates
 - barycentric, 35, 284–285, 288
 - eye-space, 542
 - floating point, 152
 - homogeneous, 39
 - normalized device, 39
 - objects, 64–65
 - spaces, 62
 - textures, 141, 146–148, 529
 - transformations, 63–66, 66–73
 - view, 65–66
 - window, 40
 - world, 65
- copying
 - from an array texture to a stereo back buffer listing (9.17), 383–384
 - data between framebuffers, 431–433
 - data in buffers, 95–97
 - data into a texture, 433–434
- Core OpenGL. *See* CGL
- core profiles, 9, 652. *See also* profiles
- counters
 - atomic, 133–137
 - performance, 597
- counting are using atomic counters listing (5.31), 135
- coverage, sample, 391–393

CPU (central processing unit) queues, 590
creating. *See also* configuration; formatting
and compiling a compute shader listing
(10.1), 438–439
a debug context with the sb6 framework
listing (13.1), 582
and initializing the full-screen window
listing (14.16), 676–677
integer framebuffer attachments listing
(9.29), 415
program member variables listing (2.6), 21
shared contexts on Windows listing (14.8),
643
a simple window listing (14.2), 629–630
a stereo window listing (9.14), 380
cross products, 56–57
CRTs (Cathode Ray Tubes), 416
csplines, 90
cube maps, 527–532
cubes
geometry
configuring, 233
drawing indexed, 234
maps, rendering to, 375–376
spinning, 121
cubic Bézier curves, 85, 86
cubic Bézier patches
fragment shader listing (8.15), 309
tessellation control shader listing (8.13),
307
tessellation evaluation shader listing (8.14),
308
tessellation example, 304–310
vertex shader listing (8.12), 306
cubic Bézier splines, 88
cubic Hermite splines, 89
culling, 40–41, 175
geometry, 320
current context, 611
curves, 82, 83–87
Bézier, 85
gamma, 418
Hermite, 198
quintic Bézier, 87
transfer, 407
damping force, calculating, 270
data, 91
atomic counters, 133–137
buffers, 92–95
allocating memory using, 92–95
feeding vertex shaders from, 97–103
filling and copying in, 95–97
driven rendering engines, 613
manipulation, built-in functions, 199–201
shader storage blocks, 126–133
stores, 92
textures, 137–185
types, 188–194
uniforms, 103–126
dds2ktx utility, 761–762
debugging applications, 581, 582–589
decay, exponential, 543
declaration
arrays, 192–193
atomic counters, 133
multiple outputs in a fragment shader listing
(9.7), 370
of multiple vertex attributes listing (7.1),
225
of PIXELFORMATDESCRIPTOR listing (14.3),
631
shader storage blocks, 126
two inputs to vertex shaders listing (5.6),
100
uniform blocks listing (5.10), 110
of vertex attributes listing (3.1), 28
vertices, 227
default block uniforms, 104–105
default framebuffer, 365
Deferred Procedure Call. *See* DPC
deferred shading, 548–558
downsides to, 556–558
normal mapping, 554–556
with normal mapping listing (12.31), 556
definitions
of `gl_in[]` listing (8.19), 314
of the Objective-C `GLCorePovfileViewController`
listing (14.10), 653
degenerate primitives, 24
denormals, 189
depth
buffers, 46
clamping, 354–355
of field effect, 457
of field using summed area tables listing
(10.8), 459–460
functions, 352
as seen from light, 537
tests, 46, 351–355
deriving a fragment's color from its position
listing (3.10), (3.12), 43, 44
design, 4–5, 714
destinations
factors, 358
subsystems, 132
detection of edges, 397–399
determining closest intersection point listing
(12.37), 572–573
development
Android environments, 729–734
builds, 607
OpenGL ES, 707–708
devices, context, 627, 629
diffuse light, 504, 505
disabling interpolation, 342–343

- discarding
 - geometry in geometry shaders, 317–320
 - rasterizers, 273
- dispatching the image copy compute shader listing (10.3), 444
- dispatch, indirect, 439–441
- displacement mapping, 300
 - GPU PerfStudio 2, 594
 - tessellation evaluation shader listing (12.23), 542
- displaying. *See also* viewing
 - an array texture–fragment shader listing (9.11), 373
 - an array texture–vertex shader listing (9.10), 373
 - EGL, 718–720
 - objects and X Window System, 689
- distance-based point size attenuation, 230
- distributions
 - grass, 242, 243
 - Linux. *See* Linux
- DMA packets, 593
- domains, 306
 - parameterization, 334
- dot products, 54–55
- double buffering, 634, 661
 - sync frame rates, 677
- double precision, 53, 60, 107
- downsides to deferred shading, 556–558
- DPC (Deferred Procedure Call), 593
- drain, queues, 591
- drawing
 - asteroids listing (7.15), 257–258
 - commands, 231–259, 595
 - clipping, 276–282
 - indexed, 231–237
 - indirect draws, 250–259
 - instancing, 237–250
 - stencil buffers, 348
 - storing transformed vertices, 259–275
 - data written to a transform feedback buffer listing (11.9), 491
 - a face normal in the geometry shader listing (8.33), 327–328
 - indexed cube geometry listing (7.3), 234
 - the same geometry many times listing (7.4), 238
 - into a stereo window listing (9.15), 381
 - Stonehenge, 663–665
 - triangles, 24–25
- drivers
 - Linux, 685–686
 - Windows graphics, 624–626
- dual-source blending, 361
- EAC (Ericsson Alpha Compression), 179
- early testing, 355–357
- edges
 - detection of, 397–399
 - jaggies, 384
- effects, atmospheric, 540–544
- EGL, 718–728
 - configs, 720–725
 - displays, 718–720
 - eglBindAPI(), 767
 - eglChooseConfig(), 720, 724
 - eglCreateContext(), 725
 - eglDestroyContext(), 726
 - eglDestroySurface(), 725
 - eglGetConfigAttrib(), 724, 725
 - eglGetConfigs(), 724
 - eglGetError(), 727
 - eglGetProcAddress(), 728
 - eglInitialize(), 720
 - eglMakeCurrent(), 726
 - eglQueryAPI(), 720
 - eglQueryString(), 727
 - eglReleaseThread(), 720
 - eglSwapBuffers(), 726
 - eglSwapInterval(), 722
 - eglWaitGL(), 727
 - eglWaitNative(), 727
 - errors, 727
 - extensions, 728
 - strings, 727
 - windows, 720
- elements, types, 193
- eliminating visual tearing, 646
- embedded environments, negotiating, 728–729
- emitting a single triangle from a geometry shader listing (8.28), 324
- EmitVertex() function, 36
- endianness, 145
- engines
 - data driven rendering, 613
 - Quartz, 647
 - tessellation, 34, 285
- enumerating pixel formats, 640–641
- environment mapping, 522–532
 - cube maps, 527–532
 - equirectangular, 525–527
 - spherical environment maps, 523–525
- environments
 - Android development, 729–734
 - EGL, 718–728
 - negotiating embedded, 728–729
 - OpenGL ES, 713–718
- equal spacing mode, 295
- equations
 - blend, 358, 361–362
 - quadratic, 85
- equirectangular environment mapping, 525–527
 - fragment shader listing (12.11), 526

- Ericsson Alpha Compression (EAC), 179
- Ericsson Texture Compression (ETC2), 179
- errors
 - code, 584
 - compiling, 201
 - EGL, 727
 - linker, 204
 - shaders, 203
- Essential Mathematics for Games and Interactive Applications*, 718
- ETC2 (Ericsson Texture Compression), 179
- Euler angles, 72
- evaluation, TES (tessellation evaluation shader), 284
- Event Trace Logs, 592
- examples
 - compute shaders, 450–471
 - shader storage block declaration listing (5.27), 127
 - stencil buffer usage listing (9.1), 350
 - subroutine uniform declaration listing (6.5), 213
 - uniform blocks
 - declaration listing (5.9), 109
 - with offsets listing (5.11), 111
 - use of indirect draw commands listing (7.10), 253
- executing compute shaders, 439–444
- exponential decay, 543
- exponents
 - bits, 189
 - shared, 181
- extending GLSurfaceView listing (14.18), 732
- extensions, 8, 617
 - applying, 618–622
 - EGL, 728
 - GLX, 695
 - vendor-specific, 728
 - WGL (Windows-GL), 634–639
- EXT extensions, 618
- extinction, 541, 542
- eye space, 64, 65, 542
- fades, 533
- failures, programs, 204
- FBOs (user-defined framebuffers), 368, 606
 - attachment completeness, 377
 - tests, 379
- feedback, transforms
 - applying, 260–265
 - starting, pausing, and stopping, 264–266
- feeding vertex shaders from buffers, 97–103
- fetching vertices, 28
- figuring out if occlusion query results are ready
 - listing (11.2), 478
- files
 - loading
 - objects, 102–103
 - from textures, 144–148
 - SBM model file format, 751–757
- filling
 - data in buffers, 95–97
 - a linked-list in a fragment shader listing (5.45), 174
- fill performance, increasing, 678–679
- filtering
 - 2D Gaussian filters, 411
 - antialiasing by, 385–387
 - mipmapping, 155–157
 - modes, 148
 - textures, 151–153
 - trilinear, 156
 - variables, 457
- finding. *See also* searching
 - a face normal in a geometry shader listing (8.21), 318
 - a pixel format with
 - wglChoosePixelFormatARB() listing (14.6), 639
- first fragment shaders listing (2.4), 18
- first geometry shader listing (3.9), 37
- first OpenGL application listing (2.1), 14
- first rule of flocking listing (10.12), 467
- first tessellation control shader listing (3.7), 34
- first vertex shaders listing (2.3), 18
- fixed-function stages, 5
- fixed outputs, 443
- fixed-point
 - data, 227
 - math, 716–718
- flags, 671
 - Boolean, 655
- flat inputs, 342
- floating-point
 - coordinates, 152
 - data, 604
 - fragment shaders, 342n1
 - framebuffers, 401–414
 - numbers, 189
 - texture formats, 402–403
- flocking
 - compute shaders, 462–471
 - vertex shader body listing (10.16), 470
- flow control barriers, 446
- FMA (fused multiply-add), 198, 199
- focal depth, 457
- focal distance, 457
- fog, 541–544
- format layout qualifiers, 169
- formatting
 - advanced framebuffers, 399–418
 - applications, 14–16
 - contexts, 696–699
 - enumerating pixels, 640–641

- formatting (*continued*)
 - pixels, 630–632
 - SBM model file format, 751–757
 - textures, 138–139, 182–185
 - windows, 628–630
- formulas, calculating indexes, 250
- fractals, rendering Julia, 566–568
- fractional even spacing, 295
- fractional segments, 295
- fragments, 341–435
 - antialiasing, 384–399
 - color output, 357–364
 - depth testing, 351–355
 - early testing, 355–357
 - off-screen rendering, 364–384
 - opacity, 15
 - OpenGL ES 3.0, 713
 - pre-fragment tests, 345–357
 - rasterization, 41
 - redeclaration of, 356
 - stencil testing, 348–351
- fragment shaders, 42–45, 342–345, 595
 - for the Alien Rain sample listing (5.42), 163
 - with an input listing (3.4), 31
 - for cube map environment rendering listing (12.16), 531–532
 - with external function declaration listing (6.2), 206
 - for generating shaped points listing (9.33), 425–426
 - with input interface blocks listing (3.6), 32
 - for normal mapping listing (12.8), 521
 - performing image loads and stores listing (5.44), 171
 - for per-fragment shininess listing (12.17), 534
 - producing high-frequency output listing (9.22), 393–394
 - ray tracing in, 568–580
 - for rendering quads listing (8.35), 336
 - with single texture coordinate listing (5.39), 147
 - for sky box rendering listing (12.14), 530
 - for the star field effect listing (9.32), 423
 - for terrain rendering listing (8.11), 304
- framebuffers, 341–435
 - advanced framebuffer formats, 399–418
 - antialiasing, 384–399
 - binding, 366
 - completeness, 377
 - copying data between, 431–433
 - default, 365
 - floating-point, 401–414
 - integers, 415–416
 - layered, 371, 382, 383
 - logical operations, 363–364
 - multiple attachments, 368–370
 - objects, 366
 - off-screen rendering, 364–384
 - OpenGL ES 3.0, 713
 - operations, 45–47, 135
 - reading from a, 429–431
 - stacks, 575
- frames
 - AFR (alternate frame rendering), 610
 - Cartesian, 519
 - sync frame rates, 677–679
- frameworks, applications, 14–16
- front end processes, 10
- front-facing, 39
- frustum matrix, 81
- full-screen
 - rendering, 644–645, 675–677
 - views (X Window System), 704
- functionality, 621
- functions. *See also* gl functions
 - blending, 358–360
 - built-in, 194–201
 - callback, 583, 584
 - depth, 352
 - EmitVertex(), 36
 - EndPrimitive(), 36
 - init(), 582
 - main(), 311
 - multi versions of, 252
 - normalization, 200
 - overloading, 143, 166, 194
 - pointers, 622
 - portability of, 633
 - RegisterClass, 628
 - shaders, 19–20
 - stencils, 349
 - vmath::perspective, 82
 - vmath::rotate, 72
- fused multiply-add. *See* FMA
- gamers, 729
- gamma curves, 418
- G-buffers, 548
 - consuming, 551–554
 - generating, 549–551
 - unpacking, 552
 - visualizing, 552
- GDI (Graphics Device Interface), 627–628
 - ChoosePixelFormat(), 634, 638
 - SetPixelFormat(), 632
 - SwapBuffers(), 593, 634
- generating
 - binding, and initializing buffers listing (5.1), 94
 - binding, and initializing textures listing (5.33), 138
 - G-buffers, 549–551

geometry in geometry shaders, 322–325
 new vertices in a geometry shader listing
 (8.27), 323–324
 geometry, 10
 cubes
 configuring, 233
 drawing indexed, 234
 drawing commands, 249. *See also* drawing
 commands
 primitive restart, combining, 235–237
 transformations, 63
 uniforms, 121–126
 geometry shaders, 36–38, 310–340
 changing the primitive type in, 35–328
 discarding geometry in, 317–320
 generating geometry in, 322–325
 layered rendering, 371
 layout qualifiers listing (8.17), 311
 modifying geometry in, 320–322
 multiple streams of storage, 328–329
 multiple viewport transformations, 336–340
 new primitive types introduced by, 329–336
 pass-through, 311–313
 quads (quadrilaterals), rendering using,
 332–336
 for rendering quads listing (8.34), 335–336
 using in an application, 313–317
 getting ready for instanced rendering listing
 (7.9), 248
 getting ready for shadow mapping listing
 (12.18), 536
 getting the result from a query object listing
 (11.1), 478
 gimbal locks, 72, 76
 GL3W
 configuring, 686
 installing, 687
 gl functions
 glActiveTexture(), 146, 150
 glAttachShader(), 19, 20, 47, 313, 438
 glBeginConditionalRender(), 481, 482
 glBeginQuery(), 484, 486, 488, 489, 490
 glBeginQueryIndexed(), 490
 glBeginTransformFeedback(), 265, 266
 glBindBuffer(), 93, 263, 600
 glBindBufferBase(), 262, 263
 glBindBufferRange(), 262, 263
 glBindFramebuffer(), 365, 367
 glBindImageTexture(), 167
 glBindProgramPipeline(), 209, 216
 glBindSampler(), 149
 glBindTexture(), 138, 663
 glBindTransformFeedback(), 491, 604
 glBindVertexArray(), 20, 21, 258, 603, 709
 glBindVertexBuffer(), 224, 229
 glBlendColor(), 358
 glBlendEquation(), 361
 glBlendEquationSeparate(), 361
 glBlendFunc(), 358, 362
 glBlendFuncSeparate(), 358, 362
 glBlitFramebuffer(), 433, 434
 glBufferData(), 92–95, 109, 113, 127, 262,
 600
 glBufferSubData(), 94, 95, 113, 134
 glCheckFramebufferStatus(), 377
 glClear(), 347
 glClearBufferfv(), 15, 18, 347, 481
 glClearBufferiv(), 349
 glClearBufferSubData(), 95, 96, 134
 glClientWaitSync(), 495–497
 glColorMask(), 363, 364, 477
 glColorMaski(), 363, 364
 glCompileShader(), 19, 20, 47, 201, 204,
 313, 438, 607
 glCompressedTexSubImage2D(), 180
 glCompressedTexSubImage3D(), 181
 glCopyBufferSubData(), 96, 262
 glCopyImageSubData(), 433
 glCopyTexSubImage2D(), 433
 glCreateProgram(), 19, 439
 glCreateShader(), 19, 20, 313, 438
 glCreateShaderProgramv(), 209
 glCullFace(), 41
 glDebugMessageCallback(), 583, 585
 glDebugMessageControl(), 586
 glDebugMessageInsert(), 587
 glDeleteProgram(), 21, 205
 glDeleteQueries(), 475
 glDeleteShader(), 20, 21, 202
 glDeleteSync(), 498
 glDeleteTextures(), 146
 glDeleteTransformFeedbacks(), 492
 glDeleteVertexArrays(), 709
 glDepthFunc(), 353
 glDepthMask(), 353
 glDepthRange(), 88
 glDepthRangeArrayv(), 338
 glDepthRangeIndexed(), 338
 glDisable(), 280, 352, 393, 540
 glDisableVertexAttribArray(), 100
 glDispatchCompute(), 439–442, 444, 481
 glDispatchComputeIndirect(), 439, 440, 442
 glDrawArrays(), 21–22, 24–26, 122, 164,
 231, 234–236, 238–240, 265, 273, 313,
 321, 481, 492, 705, 709
 glDrawArraysIndirect(), 250–252, 449
 glDrawArraysInstanced(), 239–240, 245,
 492, 709
 glDrawArraysInstancedBaseInstance(), 232,
 239, 250, 258, 439
 glDrawBuffer(), 432, 477, 661
 glDrawBuffers(), 377, 634
 glDrawElements(), 231, 234–240, 251–252,
 275, 321, 492, 709

gl functions (*continued*)

glDrawElementsBaseVertex(), 234, 235, 239
glDrawElementsIndirect(), 250, 251, 252
glDrawElementsInstanced(), 239, 240, 245, 709
glDrawElementsInstancedBaseVertex(), 239
glDrawRangeElements(), 709
glDrawTransformFeedback(), 492, 493, 604, 615, 622
glDrawTransformFeedbackInstanced(), 492, 493
glDrawTransformFeedbackStream(), 493
glEnable(), 41, 348, 391, 392, 540
glEnableVertexAttribArray(), 98, 99
glEndConditionalRender(), 481, 483
glEndQuery(), 476, 484, 486, 489, 490
glEndQueryIndexed(), 490
glEndTransformFeedback(), 491, 497, 712
glFenceSync(), 494, 495, 497, 498
glFinish(), 493, 494, 598, 603
glFlush(), 493, 661, 673, 676
glFlushMappedBufferRange(), 601, 602
glFramebufferParameteri(), 400
glFramebufferTexture(), 366, 375, 376, 390
glFramebufferTexture2D(), 376
glFramebufferTextureLayer(), 374
glFrontFace(), 41
glGenBuffers(), 93
glGenerateMipmap(), 157, 435
glGenFramebuffers(), 365
glGenProgramPipelines(), 207
glGenQueries(), 474, 475
glGenTextures(), 137, 138, 145, 167, 182
glGenTransformFeedbacks(), 491
glGenVertexArrays(), 20, 21, 709
glGetActiveSubroutineName(), 215
glGetActiveUniformsiv(), 114, 115
glGetAttribLocation(), 100, 240
glGetBooleanv(), 713
glGetBufferSubData(), 259
glGetCompressedTexImage(), 180
glGetError(), 379, 474, 496, 584, 598, 727
glGetFloatv(), 680, 713
glGetInteger64v(), 497, 713
glGetIntegeri_v(), 440
glGetIntegerv(), 116, 117, 120, 150, 230, 263, 280, 316, 446, 598, 619, 680, 697, 713
glGetInternalFormativ(), 180
glGetProgramBinary(), 217, 219
glGetProgramInfoLog(), 205, 206
glGetProgramInterfaceiv(), 210
glGetProgramiv(), 205, 217, 441
glGetProgramResourceIndex(), 214
glGetProgramResourceiv(), 210, 211, 212
glGetProgramResourceName(), 211
glGetProgramStageiv(), 215
glGetQueryObjectuiv(), 476, 477, 479, 481, 484, 487, 488, 491

glGetShaderInfoLog(), 202, 206
glGetShaderiv(), 201, 202, 204, 205, 607
glGetString(), 644
glGetSynciv(), 495
glGetTexImage(), 435
glGetTexLevelParameteriv(), 180
glGetTexParameteriv(), 180
glGetUniformBlockIndex(), 117
glGetUniformLocation(), 104, 105, 151, 598
glInvalidateBufferData(), 614, 615
glInvalidateBufferSubData(), 614, 615
glInvalidateFramebuffer(), 615
glInvalidateSubFramebuffer(), 615
glInvalidateTexImage(), 614, 615
glInvalidateTexSubImage(), 614
glIsTransformFeedback(), 492
glLinkProgram(), 20, 47, 204, 206, 217, 264, 313, 438, 608
glLogicOp(), 362, 713
glMapBuffer(), 95, 109, 113, 127, 259, 600, 601, 603, 739
glMapBufferRange(), 134, 429, 599, 600, 601, 602, 603
glMemoryBarrier(), 131, 132, 137, 177
glMinSampleShading(), 394
glMultiDrawArrays(), 709
glMultiDrawArraysIndirect(), 232, 257, 258
glMultiDrawElements(), 709
glMultiDrawElementsIndirect(), 232
glObjectLabel(), 588
glObjectPtrLabel(), 588, 589
glPatchParameterfv(), 298
glPatchParameteri(), 33, 298
glPixelStorei(), 430
glPointParameteri(), 423
glPointSize(), 22, 26, 37, 230
glPolygonMode(), 36, 296
glPopDebugGroup(), 585, 588
glProgramBinary(), 219
glProgramParameteri(), 207, 217
glPushDebugGroup(), 585
glQueryCounter(), 485, 486
glReadBuffer(), 429, 430
glReadPixels(), 429, 430, 431, 433, 435, 598, 599, 600, 680
glSampleCoverage(), 392
glSamplerParameterf(), 149, 150
glSamplerParameterfv(), 159
glSamplerParameteri(), 149, 150, 158
glScissorIndexed(), 346
glScissorIndexedv(), 346
glShaderSource(), 19, 20, 47, 203, 209, 313, 438
glStencilFunc(), 351
glStencilFuncSeparate(), 348, 350, 351
glStencilMaskSeparate(), 351
glStencilOp(), 351
glStencilOpSeparate(), 348, 349, 350, 351

- glTexBuffer(), 273
- glTexParameterI(), 536
- glTexStorage2D(), 138, 144, 154, 156, 167, 180, 389, 527
- glTexStorage2DMultisample(), 389, 390
- glTexStorage3D(), 161, 180, 389
- glTexStorage3DMultisample(), 389, 390
- glTexSubImage2D(), 138, 139, 144, 154, 185, 429, 433, 435, 527, 613
- glTexSubImage3D(), 161, 614
- glTextureView(), 182
- glTransformFeedbackVaryings(), 260, 261, 263, 264
- glUniform*(), 105, 106, 108
- glUniform1i(), 151
- glUniform4fv(), 439
- glUniformBlockBinding(), 118, 119
- glUniformSubroutinesuiv(), 215
- glUnmapBuffer(), 601
- glUseProgram(), 21, 120, 215, 216, 439, 711
- glUseProgramStages(), 207, 215
- glVertexAttrib*(), 29, 30, 99, 100, 165
- glVertexAttrib4fv(), 29
- glVertexAttribBinding(), 224, 228
- glVertexAttribDivisor(), 470
- glVertexAttribFormat(), 224, 225, 227, 228
- glVertexAttribI*(), 165
- glVertexAttribIFormat(), 227
- glVertexAttribIPointer(), 755
- glVertexAttribPointer(), 98, 99, 101, 102, 224, 245, 263, 605, 709
- glViewport(), 40, 122, 336, 337, 338, 401, 674, 679
- glViewportArrayv(), 338
- glViewportIndexedf(), 337
- glViewportIndexedfv(), 337
- glWaitSync(), 497–498
- GLFW
 - configuring, 686
 - installing, 686–687
- GLKit, 648, 662–673
 - 3D math with, 667–669
 - Cocoa, 669–671
 - iOS, 737–738
- GLKTextureInfo class, 663
- GLKViewController class, 741
- global illumination, 558
- global work groups, 440–441, 456
- gloss maps, 533
- GLSL (OpenGL Shading Language), 17, 740–741
- GL_TRIANGLES_ADJACENCY primitive mode, 330, 331
- GLUT (OpenGL Utility Toolkit), 648, 680–681
 - main function to set up OpenGL listing (14.17), 681
- GLX
 - glXChooseFBConfig(), 692, 693, 694
 - glXCopyContext(), 698
 - glXCreateContextAttribsARB(), 696, 697
 - glXCreateNewContext(), 696
 - glXCreateWindow(), 694
 - glXDestroyContext(), 698
 - glXDestroyWindow(), 694
 - glXGetClientString(), 695
 - glXGetCurrentReadDrawable(), 700
 - glXGetFBConfigAttrib(), 692
 - glXGetFBConfigs(), 689, 692, 694
 - glXGetProcAddress(), 695
 - glXIsDirect(), 699
 - glXMakeContextCurrent(), 699
 - glXMakeCurrent(), 611
 - glXQueryContext(), 700
 - glXQueryDrawable(), 700
 - glXQueryExtensionsString(), 695
 - glXQueryServerString(), 695
 - glXQueryVersion(), 688
 - glXSwapBuffers(), 700, 703
 - glXWaitGL(), 699
 - glXWaitX(), 700
 - queries, 700–701
 - strings (Linux), 695
 - synchronization, 699
 - windows, 701–704
- GLX-interfacing with X Window System, 688–689
- Google, 707
- Gouraud shading, 507
 - fragment shader listing (12.2), 508
 - vertex shader listing (12.1), 507–508
- GPU PerfStudio 2, 594–597
- GPUs (Graphics Processing Units), 5, 609–611
- GPUView, 590–594
- graphics, 3
 - math, 49. *See also* math
 - output, 627
 - pipelines, 4–6, 27–48. *See also* pipelines
 - processors, 218, 594
 - programs, 438
- Graphics Device Interface. *See* GDI
- graphics drivers (Windows), 624–626
- Graphics Processing Units. *See* GPUs
- graphics processors, compute shaders, 437–472
- graphs, exponential decay, 543
- grass
 - colors, 244
 - distribution, 242, 243
 - length of, 245
 - positioning, 241
- gravity, 270
- groups
 - messages, 587
 - outputs, 296
 - work, 440–441
- guard bands, 278

- hardware, 4, 9
 - Linux, 685–686
 - queues, 590
 - rasterizers, 10
 - support, 625
- hazards, 129, 137
- HDR (High Dynamic Range), 403–404, 606
- header of a .KTX file listing (5.36), 144–145
- heads-up display (HUD), 485
- Hermite curve, 198
- High Dynamic Range. *See* HDR
- higher order surfaces, 324
- highlights, specular, 505–509
- hints, 209, 614
- histograms, 405
- history, 3, 6–10
 - of Linux, 682–683
 - of OpenGL ES, 706
- homogenous coordinates, 39
- homogenous vectors, 53
- Hooke's law, 269, 270, 271
- HUD (heads-up display), 485, 595
- ICD (Installable Client Driver), 624, 626
- identity matrix, 67–68
- IEEE-754, 188
- illumination, global, 558
- images
 - access synchronization, 176–177
 - atomic operations on, 171–176
 - columns, 454
 - stereo, viewing in, 380
 - transposing, 412
 - units, 167
 - variables, 165
- increasing fill performance, 678–679
- indexes
 - arrays, 245
 - data chunks (SBM model file format), 753
 - drawing commands, 231–237
 - formulas, calculating, 250
 - global work groups, 456
 - queries, 489–490
 - uniforms, 112
- indirect draws, 250–259
- infinity, 200
- in flight (executing hardware commands), 4
- init() function, 582
- initializing
 - array textures listing (5.40), 161–162
 - core context views listing (14.11), 654
 - a G-buffer listing (12.27), 550
 - shader storage buffers for flocking listing (10.9), 464
 - textures, 138–139
- inner loop of the julia renderer listing (12.34), 567
- inner products, 54–55
- inputs
 - compute shaders, 441–444
 - flat, 342
 - to the flock rendering vertex shader listing (10.15), 469
 - primitive types, 315
 - smooth, 342
 - vertex shaders, 224–229
- inscattering, 541
- inserting geometry shaders, 37
- Installable Client Driver. *See* ICD
- installing
 - GL3W, 687
 - GLFW, 686–687
- instancing
 - arrays, 288
 - drawing commands, 237–250
 - rendering, 245–250, 249
- integers, 189
 - framebuffers, 415–416
- Integrated Raster Imaging System Graphics Library. *See* IRIS GL
- Interface Builder, 651, 659
- interfaces
 - Aero user, 592
 - APIs (Application Programming Interfaces), 3
 - blocks, 31–32
 - GDI (Graphics Device Interface), 627–628
 - GLX-interfacing with X Window System, 688–689
 - Mac OS X, 648–649
 - matching, 209–213
 - overriding, 652–659
- interleaved attributes, 101
- internal formats, 138
- interpolation, 44, 82
 - curves, 85
 - disabling, 342
 - Hermite, 198
 - linear, 83
 - perspective-correct, 344, 345
 - splines, 88
 - and storage qualifiers, 342–345
- Interrupt Service Routine. *See* ISR
- invocations, 188
- iOpenGL, 734–744
- iOS
 - C++, 737
 - configuring, 734–736
 - GLKit, 737–738
 - GLSL (OpenGL Shading Language), 740–741
- IRIS GL (Integrated Raster Imaging System Graphics Library), 682
- isoline spirals tessellation evaluation shader listing (8.7), 292

- ISR (Interrupt Service Routine), 593
- items, work, 47
- iterating over elements of `gl_in[]` listing (8.18), 312
- jaggies, 384
- Julia set, 566–568
- Khronos Group, 706–707
- Khronos Texture File format, 529
- Kilgard, Mark J., 680
- knots, 87n4
- .KTX (Khronos TeXture) format, 144, 145
- ktxtool utility, 759–761
- languages, overview of, 188–201
- layers, 162
 - abstraction, 4, 5
 - rendering, 370–376
 - rendering using a geometry shader listing (9.9), 372
- layouts
 - columns, 60
 - qualifiers, 29, 104, 370
 - binding, 118
 - control points, 33
 - depth, 356, 357
 - format, 169
 - geometry shader, 311
 - location, 370
 - shared, 110
 - standard, 110, 116
- length
 - of grass, 245
 - of vectors, 57
- levels, 138
 - generating mipmapping, 157
- libraries, math, 54, 59, 61
- light bloom, 409–414
- lighting a fragment using data from a G-buffer listing (12.30), 553
- lighting models, 504–544
 - Blinn-Phong lighting model, 513–515
 - environment mapping, 522–532
 - normal mapping, 518–522
 - Phong lighting model, 504–513
 - rim lighting, 515–517
- light spaces, occluding, 559
- linear interpolation, 83
- linear texturing, 245
- lines, 82
 - clipping, 276–282
 - parallel, 80
 - smoothing, 385
- links
 - makefiles, 687
 - programs, 204–206
- Linux, 682–704
 - applications, building, 687–693
 - applying contexts, 699–701
 - config management and visuals, 689–693
 - GLX
 - creating windows, 701–704
 - strings, 695
 - history of, 682–683
 - managing contexts, 695–699
 - rendering, 693–694
 - starting, 683–687
 - windows, 693–694
 - X Window System, 683
- loading
 - 2D array textures, 163–165
 - a cube map texture listing (12.12), 528–529
 - a .KTX file listing (5.37), 145
 - objects from files, 102–103
 - textures, 144–148, 665–667
- local work groups, 47, 440–441, 444
- locations, 104
 - uniforms, 114
 - of vertex attributes to zero, 29
- locks, gimbal, 72, 76
- logical operations, 363–364
- logs, Event Trace Logs, 592
- lookout matrix, 77–79
- loops
 - main, 633
 - rendering, 273
- Mac OS X, 647–681
 - CGL (Core OpenGL), 674–675
 - Cocoa, 649–662
 - full-screen rendering, 675–677
 - GLUT (OpenGL Utility Toolkit), 680–681
 - interfaces, 648–649
 - multi-threaded OpenGL, 679–680
 - OpenGL on, 647–681
 - rendering in, 660
 - retina displays, 673–674
 - sync frame rates, 677–679
- macros, `APIENTRY`, 583
- magnification filters, 152
- main body of the flocking update compute shader listing (10.14), 468–469
- `main()` function, 311
- main loops, 633
- makefiles, 687
- managing
 - config management and visuals, 689–693
 - context, 695–699, 725–726
 - viewports, 660
- Mandelbrot sets, 566
- mantissa bits, 189

- mapping
 - buffers, 600–603
 - a buffer's data store listing (5.3), 95
 - bump, 518
 - displacement, 300
 - environment, 522–532
 - cube maps, 527–532
 - equiangular, 525–527
 - spherical environment maps, 523–525
 - gloss maps, 533
 - GPU PerfStudio 2, 594
 - normal, 518–522, 554–556, 605
 - rendering to cubes, 375–376
 - shadows, 534–540
 - tone, 404–409
 - vertex shader inputs, 228
- marching rays, 560
- masking colors, 363–364
- matching interfaces, 209–213
- material properties, 532–534
- math, 49
 - 3D math with GLKit, 667–669
 - built-in functions, 197–199
 - curves, 83–87
 - fixed-point, 716–718
 - library, 54, 59, 61
 - matrices, construction and operators, 60–63
 - operators, 54–58
 - quaternions, 75–76
 - splines, 87–90
 - transformations, 63–82
 - concatenation, 73–75
 - model-view transforms, 76–79
 - projection, 79–81
 - vectors, 51–54
- matrices, 53, 58–60, 190–192
 - built-in functions, 195–197
 - construction, 60–63
 - drawing commands, 232
 - frustum, 82
 - identity, 67–68
 - lookout, 77–79
 - operators, 60–63
 - perspective, 81
 - rotation, 70–72
 - scaling, 72–73
 - shadows, 538
 - transformations, 62, 66
 - translation, 68–70
 - uniforms, 107–108
- member variables, 21
- memory
 - access synchronization, 129–133
 - allocation using buffers, 92–95
 - atomic operations, 128–129
 - bandwidth, 178, 549
 - hazards. *See* hazards
 - optimization, 613–616
- Mesa, 682, 685
- messages
 - debug, 586, 587
 - loops, 633
- methods, 14, 194
- minification filters, 152
- mipmapping, 138, 153–155
 - cube map support, 529
 - example program, 158
 - filtering, 155–157
 - levels, 157–158
- mobile platforms, 705–744
 - Android development environments, 729–734
 - EGL, 718–728
 - gamers, 729
 - iOpenGL, 734–744
 - negotiating embedded environments, 728–729
 - OpenGL ES, 705–709
 - OpenGL ES 3.0, 709–713
- models
 - lighting, 504–544
 - Blinn-Phong lighting model, 513–515
 - environment mapping, 522–532
 - normal mapping, 518–522
 - Phong lighting model, 504–513
 - Rim lighting model, 515–517
 - SBM model file format, 751–757
 - transformations, 63, 67
- model space, 64
- model-view transforms, 76–79, 667
- models
 - filtering, 148
 - parameters, 265
 - separable, 207
 - wrapping, 148
- modifying
 - geometry in geometry shaders, 320–322
 - the primitive type in geometry shaders, 325–328
- monolithic program objects, 206
- movement keys, 671
- MSAA (multi-sample antialiasing), 387–389
- multi-dimensional arrays, 194
- multiple framebuffer attachments, 368–370
- multiple GPUs, 609–611
- multiple interleaved vertex attributes listing (5.8), 102
- multiple separate vertex attributes listing (5.7), 101
- multiple streams of storage, 328–329
- multiple textures, 150–151
- multiple threads, 611–613

- multiple vertices, 24
 - attributes, 225
 - shader inputs, 100–102
- multiple viewport transformations, 336–340
- multiplication, 62
 - coordinate spaces, 63–66
 - matrices, 62
 - model-view transformations, 76–79
 - quaternions, 75–76
- multi-sampling, 46n3
 - aliasing, 140
 - antialiasing, 387–389
 - textures, 389–393
- multi-threaded OpenGL, 679–680
- multi versions of functions, 252
- naïve rotated point sprite fragment shader
 - listing (9.34), 427
- names, 92, 138
- NaN (Not a Number), 189, 200
- NDC (Normalized Device Coordinate) Space, 64, 66
- negative reflections, 506
- negotiating embedded environments, 728–729
- new primitive types introduced by geometry
 - shaders, 329–336
- Newton's laws, 269, 270, 271
- noninstanced rendering, 245–250
- non-photo-realistic rendering, 544–547
- normalization, 52
 - buffers, 98
 - functions, 200
 - positive values, 226
- Normalized Device Coordinate Space. *See* NDC Space
- normalized device spaces, 39
- normal mapping, 518–522, 605
 - deferred shading, 554–556
- normals, finding, 328
- Not a Number (NaN), 189, 200
- NSOpenGL, 648, 652–659
- NULL pointers, 94
- NVIDIA drivers, 625, 682
- Nyquist rate, 384
- objects
 - buffers, storage, 251
 - container, 603
 - coordinates, 64–65
 - display objects and X Window System, 689
 - files, loading, 102–103
 - framebuffers, 366
 - instancing, 237–250
 - list chunks, 756
 - monolithic program, 206
 - program pipeline, 207, 608
 - programs, 17
 - queries, 474
 - rotation, 70–72
 - samplers, 148
 - separable program, 608
 - shaders, 17
 - space, 64
 - stacks, 575
 - sync, 494
 - TBO (texture buffer object), 266, 269
 - texture, 148
 - UBO (Uniform Buffer Object), 108
 - VAOs (vertex array objects), 272
- object-space coordinate data, 605
- occlusion
 - ambient, 558–565
 - queries, 475–484
- off-screen rendering, 364–384
- offsets, 250
 - polygons, 540
- opacity fragments, 15
- OpenGL
 - Mac OS X on, 647–681. *See also* Mac OS X
 - multi-threaded, 679–680
 - in Windows, 623–647. *See also* Windows
- OpenGL ES, 705–709, 713–718
- OpenGL ES 3.0, 709–713
- OpenGL Shading Language. *See* GLSL
- operating systems. *See* platforms
- operators
 - AND, 47
 - comparison, 352
 - matrices, 60–63
 - OR, 47
 - standard, 192
 - vectors, 54–58
- optimization
 - compute shaders, 437–472
 - with extensions, 619–622
 - memory, 613–616
 - performance. *See* performance optimization
- orientation, calculating, 257
- origin of OpenGL, 6–10
- OR operator, 47
- orthographic projections, 80, 81, 83
- orthonormal, 519
- outputs
 - colors, 357–364
 - compute shaders, 441–444
 - graphics, 627
 - groups, 296
 - vertex shaders, 229–230
- outputting information about the OpenGL
 - context listing (14.12), 660
- overdraw, 548
- overloading functions, 15, 143, 166, 194
- overriding NSOpenGL, 652–659

- packed data formats, 227
- packed vertex attributes, 247
- packets
 - DMA, 593
 - present, 593
 - standard queue, 592
- parallax, 379
- parallelism, 4, 42
- parallel lines, 80
- parallel prefix sum (compute shader example), 450–462
- parallaxization, 450–462
- parameters
 - domains, 334
 - mode, 265
 - points, 423–424
- passing data between tessellation shaders, 296–299
- pass-through
 - geometry shaders, 311–313
 - vertex shader listing (8.25), 323
 - vertex shader that includes normals listing (8.30), 326
- patches, 284, 340
 - cubic Bézier patches (tessellation example), 304–310
 - domains, 306
 - processes, 284
- Paul, Brian, 682
- pausing transform feedback, 264–266
- performance
 - counters, 597
 - increasing fill, 678–679
 - optimization, 581, 589–616, 597–616
- performance analysis tools, 589–597
 - GPU PerfStudio 2, 594–597
 - GPUView, 590–594
 - WPT (Windows Performance Toolkit), 590–594
- per-fragment lighting, calculating, 518
- per-indirect draw attribute setup listing (7.13), 255
- per-instance rotations, calculating, 163
- per-patch inner/outer tessellation factors, 284
- perspective
 - coordinates, 66
 - division, 39
 - matrices, 81
 - perspective-correct interpolation, 344–345
 - projections, 80, 81, 82
- perturbations, random, 242
- per-vertex lighting (Gouraus shading), 509
- Phong lighting model, 504–513
- Phong shading, 509–513, 519, 521
 - fragment shader listing (12.4), 511–512
 - vertex shader listing (12.3), 510–511
- physical simulation example, 266–275
- pipelines, 10–11, 17, 27–48
 - clipping, 38–39
 - compute shaders, 47–48
 - fragment shaders, 42–45
 - framebuffer operations, 45–47
 - geometry shaders, 36–38
 - graphics, 4–6
 - interface blocks, 31–32
 - tessellation, 32–36
 - vertices
 - passing data from stage to stage, 29–32
 - passing data to shaders, 28–29
- pixels, 10–11, 17
 - advanced formats, 643
 - calculating, 385
 - centroid sampling, 396, 397
 - counting, 480
 - enumerating formats, 640–641
 - format attributes, 643–644
 - formatting, 630–632
 - Phong shading, 521
- platforms, 617
 - extensions, 618–622
 - Linux, 682–704
 - Mac OS X on, 647–681. *See also* Mac OS X
 - mobile, 705–744
 - Windows, 623–647. *See also* Windows
- pointers, 15
 - functions, 622
 - NULL, 94
- point mode, tessellation, 292–294
- points
 - binding, 117, 262
 - clipping, 276
 - control, 83, 284, 304
 - parameters, 423–424
 - rotation, 426–428
 - shaped, 424–426
 - sizing, 22
 - sprites, 419–428
 - textures, 420
 - variables, 230
- polygons
 - concave, 10
 - offsets, 540
 - smoothing, 386
- portability of functions, 633
- positioning
 - antialiasing sample, 387
 - calculating, 257
 - control points, 284
 - grass, 241
 - math, 51
- positive value normalization, 226
- predication, 481

- prefix sum, 450–462
 - implementation using a compute shader listing (10.6), 453
- pre-fragment tests, 345–357
- pre-optimizing shaders, 609
- present packets, 593
- primitive mode tessellation, 285–294
- primitiveMode values, 265
- primitive processing, 283–340
 - communication between shader invocations, 299
 - cubic Bézier patches (tessellation example), 304–310
 - geometry shaders, 310–340
 - terrain rendering (tessellation example), 300–304
 - tessellation, 284–310
- primitives, 10–11
 - assembly, 10, 36, 175
 - degenerate, 24
 - restart, combining geometry, 235–237
 - types, adjacency, 340
- printing interface information listing (6.4), 212
- processes
 - back end, 11, 341
 - fragments, 341–435
 - front end, 10
 - primitive, 283–340. *See also* primitive processing
 - vertices, 224–230
- processors
 - GPUs (Graphics Processing Units), 5
 - graphics, 218, 594
- producing
 - lines from normals in the geometry shader listing (8.32), 327
 - multiple vertices in a vertex shaders listing (2.8), 24
- products
 - cross, 56–57
 - dot, 54–55
 - inner, 54–55
 - vectors, 55
- profiles, 9. *See also* core profiles
- programmable stages, 28
- programs, 13, 187. *See also* applications
 - binaries, 216–219, 609
 - compiling, 201–219
 - compute, 438. *See also* compute shaders
 - linking, 204–206
 - monolithic objects, 206
 - objects, 17
 - pipeline objects, 207, 608
 - separable, 206–213
 - shaders, 5
- projection
 - matrices, 123
 - orthographic, 419
 - perspective, 76, 80, 667
 - transformations, 79–81
- properties, material, 532–534
- pseudo-code
 - for `glDrawArraysInstanced()` listing (7.5), 240
 - for `glDrawElementsInstanced()` listing (7.6), 240
- publication dates, 7
- pulling vertices, 28
- pushing a face out along its normal listing (8.24), 321
- quadratic Bézier curves, 85, 86
- quadratic equations, 86
- quads (quadrilaterals)
 - geometry shaders, rendering using, 332–336
 - tessellation using, 285–288
- qualifiers
 - binding layout, 119
 - centroid, 395
 - format layout, 169
 - layout, 29, 104, 370
 - storage, interpolation and, 342–345
- Quartz, 647
- quaternions, 72, 75–76
- queries, 474–493
 - extension functions, 622
 - GLX, 700–701
 - indexed, 489–490
 - objects, 522
 - result availability, 475
 - retrieving, 476
 - timer, 484
 - transform feedback, 487–489
 - occlusion, 475–484
 - results, 476–480, 490–493
 - timer, 484–487
 - transform feedback, 487–493
- queues
 - CPU, 590
 - drain, 591
 - hardware, 590
 - software, 590
 - standard queue packets, 592
- race conditions, 446, 447
- radians, 199
- random perturbations, 242
- rasterization, 41–42
 - back end processes, 341
 - OpenGL ES 3.0, 712
- rasterizers, 10
 - discard, 273
 - guards bands, 278

- rates, sync frame, 677–679
- RAW (Read-After-Write), 129
- ray-plane intersection test listing (12.38), 578
- rays, 560, 568–580
- ray-sphere intersection test listing (12.36), 572
- RC (rendering context), 628, 632–634
- Read-After-Write (RAW), 129
- reading
 - back texture data, 434–435
 - from a framebuffer, 429–431
 - state or data from OpenGL, 597–600
 - textures, 148–165
 - from textures in GLSL listing (5.35), 141
- Realtech VR OpenGL Extensions Viewer, 619
- rectangles
 - prefix sums, 460
 - textures, 140n4
- redeclaration of fragments, 356
- Red-Green Texture Compression (RGTC), 179
- redirecting the current folder to point our resources listing (14.21), 742–743
- reflection, 57–58, 507
- reflectivity, 504
- refraction, 57–58
- RegisterClass function, 628
- registering a window class listing (14.1), 628
- rendering, 503
 - 3D graphics, 630
 - AFR (alternate frame rendering), 610
 - with all blending functions listing (9.3), 359–360
 - alternative rendering methods, 548–580
 - Android projects, 731–734
 - asteroids, 257
 - atmospheric effects, 540–544
 - casting shadows, 534–540
 - conditional, 481, 598
 - context. *See* RC
 - cube maps, 375–376, 529
 - data driven engines, 613
 - deferred shading, 548–558
 - environment mapping, 522–532
 - full-screen, 644–645, 675–677
 - HDR (High Dynamic Range), 403–404
 - instancing, 238, 245–250, 249
 - Julia fractals, 566–568
 - layers, 370–376
 - lighting models, 504–544
 - Linux, 693–694
 - loops, 273
 - for the Alien Rain sample listing (5.43), 164
 - for the flocking example listing (10.10), 465
 - listing (5.23), (5.26), 123, 125–126
 - in Mac OS X, 660
 - material properties, 532–534
 - to multiple viewports in a geometry shader listing (8.36), 338–339
 - with no attachments, 399–401
 - noninstanced, 245–250
 - non-photo-realistic, 544–547
 - off-screen, 364–384
 - pipelines. *See* pipelines
 - quads (quadrilaterals) using geometry shaders, 332–336
 - scissor tests, 347
 - screen-space techniques, 558–565
 - single points listing (2.7), 22
 - single triangles listing (2.9), 25
 - sky boxes, 531
 - starfields, 420–423
 - in stereo, 379–384
 - Stonehenge, 663–665
 - surfaces, 725
 - synchronization, 726–727
 - terrain, 300–304
 - a texture listing (9.5), 367–368
 - textures, 610
 - to two layers with a geometry shader listing (9.16), 382–383
 - when query results aren’t available listing (11.4), 480
 - without a TCS (tessellation control shader), 298–299
 - without triangles, 565–580
- render() method, 14
- resolution, retina displays (Mac OS X), 673–674
- resources
 - 3D graphics books, 748
 - OpenGL books, 747–748
 - web sites, 748–749
- restarting geometry and primitives, 235–237
- results
 - primitive queries, 490–493
 - queries, 476–480
- retina displays (Mac OS X), 673–674
- retrieving
 - compiler logs from a shader listing (6.1), 202–203
 - indices of uniform block members listing (5.12), (5.15), 112, 115
 - information about uniform block members listing (5.13), 113
 - a program binary listing (6.7), 217–218
- return values, framebuffer completeness, 378
- RGTC (Red-Green Texture Compression), 179
- rim lighting, 515–517
 - shader function listing (12.6), 516
- rotated point sprites
 - fragment shader listing (9.36), 427
 - vertex shader listing (9.35), 427–428
- rotation
 - matrices, 70–72
 - points, 426–428

- roughness, 533
- rules, uniform blocks, 111
- samples
 - centroid sampling, 395–399
 - coverage, 391–393
 - multi-sample
 - antialiasing, 387–389
 - textures, 389–393
 - objects, 148
 - parameters, 149
 - rate shading, 393–395
 - types, 142
 - variables, 141
- sampling rates, 384
- SB6
 - sb6GetProcAddress(), 622, 635
 - sb6IsExtensionSupported(), 620, 622
- sb6mtool utility, 762–764
- SBM model file format, 751–757
 - chunk headers, 752
 - defined chunks, 752–756
 - examples, 757
 - file headers, 751–752
- scalability, 4, 67
- scalars, 105–106, 188–189
- scaling matrix, 72–73
- scissor tests, 46, 345–348
- screen-space techniques, 558–565
- searching bind sections of buffers, 262
- segments, fractional, 295
- selecting
 - OpenGL ES versions, 708–709
 - pixel formats, 631–662, 654
- selectors, 150
- separable mode, 207
- separable program objects, 608
- separate attributes, 100
- separate programs, 206–213
- serialization, 129
- servers (X Window System), 693
- sets
 - julia, 566–568
 - Mandelbrot, 566
- setting. *See also* configuration
 - the debug callback function listing (13.2), 583
 - single floats in uniform blocks listing (5.14), 114
 - up a full-screen window listing (14.9), 645
 - up a layered framebuffer listing (9.8), 371
 - up a multisample framebuffer attachment listing (9.20), 390
 - up and rendering Android listing (14.19), 733
 - up an FBO with multiple attachments listing (9.6), 369–370
 - up a shadow matrix listing (12.20), 538
 - up a simple framebuffer object listing (9.4), 367
 - up atomic counter buffers listing (5.29), (5.30), 134
 - up cube geometry listing (5.20), 121–122
 - up indexed cube geometry listing (7.2), 233
 - up indirect draw buffers for Asteroids listing (7.11), 254
 - up matrices for shadow mapping listing (12.19), 536
 - up matrices in uniform blocks listing (5.17), 116
 - up scissor rectangle arrays listing (9.1), 346–347
 - up the “Explode” geometry shader listing (8.23), 321
 - up the julia set renderer listing (12.33), 567
 - up the “Normal Visualizer” geometry shader listing (8.31), 326–327
 - up the “Tessellator” geometry shader listing (8.26), 323
 - up vertex attributes listing (5.4), 99
 - values of subroutine uniforms listing (6.6), 216
- SGI (Silicon Graphics, Inc.), 6, 682
- shaders, 5, 187
 - applying, 16–23
 - barriers, 132–133
 - blocks, storage, 126–133
 - communication between invocations, 299
 - compiling, 218, 606–609
 - compute, 47–48, 437–472
 - applying, 438–439
 - communication, 444–449
 - examples, 450–471
 - executing, 439–444
 - flocking, 462–471
 - synchronizing, 445–449
- cores, 5
- fragments, 42–45, 342–345
- geometry, 36–38, 310–340
 - changing the primitive type in, 325–328
 - discarding geometry in, 317–320
 - generating geometry in, 322–325
 - layered rendering, 371
 - modifying geometry in, 320–322
 - multiple streams of storage, 328–329
 - multiple viewport transformations, 336–340
 - new primitive types introduced by, 329–336
 - pass-through, 311–313
 - storage blocks, 126, 129
 - tessellation. *See* tessellation
 - using in an application, 313–317
 - vertex. *See* vertex shaders
- objects, 17
- OpenGL ES 3.0, 710–712
- pre-optimizing, 609

- shaders (*continued*)
 - subroutines, 213–216
 - TCS (tessellation control shader), 284
 - TES (tessellation evaluation shader), 284
 - tessellation
 - control, 33–34
 - evaluation, 34–36
 - passing data, 296–299
 - textures
 - reading from in, 141–144
 - writing to in, 165–176
 - vertices, 24
 - feeding from buffers, 97–103
 - inputs, 224–229
 - multiple inputs, 100–102
 - outputs, 229–230
 - passing to data to, 28–29
- shading
 - cell, 545–547
 - deferred, 548–558
 - downsides to, 556–558
 - normal mapping, 554–556
 - Gouraud, 507
 - Phong, 509–513, 519
 - sample rate, 393–395
- shadows
 - casting, 534–540
 - mapping, 534–539
 - matrix, 538
 - sampler, 535
- shaped points, 424–426
- shared memory, 445
- sharing
 - exponents, 181
 - layouts, 110
 - variables, 444
- shininess factor, 506
- shutting down applications, 21
- side effects, 443
- signaled states, 494
- signed areas, 40
- signed integers, 189
- Silicon Graphics, Inc. (SGI), 6, 680
- simple
 - application side conditional rendering
 - listing (11.3), 479
 - do-nothing compute shader listing (3.13), 47
 - instanced vertex shader listing (7.8), 247
 - isoline tessellation control shader example
 - listing (8.5), 291
 - isoline tessellation control shader example
 - listing (8.6), 291
 - multisample maximum resolve listing (9.21), 391
 - prefix sum implementation in C++ listing (10.5), 450
 - quad tessellation control shader example
 - listing (8.1), 287
 - quad tessellation evaluation shader example
 - listing (8.2), 287–288
 - simplified fragment shader for shadow mapping listing (12.22), 539
 - simplified vertex shader for shadow mapping listing (12.21), 538
 - triangle tessellation control shader example
 - listing (8.3), (8.4), 289, 290
 - vertex shader with pre-vertex color listing (7.7), 246
- single buffering, 661
- sizing
 - input arrays, 315
 - points, 22
 - variables, points, 230
- sky boxes, 529, 531
- smoothing
 - inputs, 342
 - lines, 385
 - polygons, 386
- SoAs (structure-of-arrays), 101
- software queues, 590
- source code for a simple geometry shader listing (8.16), 311
- source factors, 358
- spaces, color, 416–418
- specifications, CGL (Core OpenGL), 625n3
- specifying
 - bindings for uniform blocks listing (5.18), 119
 - data for arrays in uniform blocks listing (5.16), 115
 - varyings, 260
- specular albedo, 512
- specular highlights, 505–509
- speed, tuning applications for, 597–616
- spherical environment mapping, 523–525
 - fragment shader listing (12.10), 524
 - vertex shader listing (12.9), 523–524
- spinning cube
 - fragment shader listing (5.25), 124
 - vertex shader listing (5.24), 123
- splines, 82, 87–90
- spring mass system
 - example, 266
 - iteration loop listing (7.18), 273
 - rendering loop listing (7.19), 273
 - vertex setup listing (7.16), 269
 - vertex shader listing (7.17), 271–272
- sprites, points, 419–428
- sRGB color spaces, 416–418
- SSAO (screen space ambient occlusion), 559
- stacks, implementing, 575
- stages, 5
 - fixed-function, 5
 - shaders, 17
- standard layouts, 110, 116
- standard operators, 192

- standard queue packets, 592
- starfields, rendering, 420–423
- starting
 - applications, 21
 - Linux, 683–687
 - transform feedback, 264–266
- state, OpenGL ES 3.0, 713
- stencil tests, 46, 348–351
- stereo, rendering in, 379–384
- Stonehenge, 663–665
- stopping transform feedback, 264–266
- storage
 - buffer objects, 251
 - multiple streams of, 328–329
 - qualifiers
 - interpolation and, 342–345
 - patch, 445
 - shared, 445
 - shaders, blocks, 126–133
 - transforms, vertices, 259–275
- strings
 - EGL, 727
 - GLX, 695
- structure-of-arrays (SoAs), 101
- structures, 192–194
 - acceleration, 579
 - VERTEX, 225
- subdivision modes, tessellation, 294–296
- subroutines, shaders, 213–216
- subsystems, destination, 132
- summed area tables, 456, 460
- sums, prefixes, 450–462
- support
 - core profiles, 652
 - hardware, 625
 - ICD (Installable Client Driver), 624, 626
 - on Linux, 684
- surfaces, render, 626, 693–694, 725
- SwapBuffers() command, 593
- swap values, buffers, 637
- swizzling, 191
- sync frame rates, 677–679
- sync objects, 494
- synchronization, 493–498
 - access
 - to atomic counters, 137
 - to images, 176–177
 - to memory, 129–133
 - compute shaders, 445–449
 - and fences, 494–498
 - GLX, 699
 - rendering, 726–727
- tables, summed area, 456, 460
- taking a screenshot with `glReadPixels()` listing (9.37), 430–431
- tangents
 - space normals, 554
 - vectors, 519
- targets, 92, 137, 139–140
- TBN (Tangent, Bitangent, Normal) matrix, 519, 555
- TBO (texture buffer object), 266, 269
- TCS (tessellation control shader), 284, 298–299, 595
 - for terrain rendering listing (8.9), 302–303
- terrain, rendering, 300–304
- TES (tessellation evaluation shader), 34–36, 284, 595
 - for terrain rendering listing (8.10), 303
- tessellation, 32–36, 284–310
 - control shaders, 33–34
 - engines, 34
 - evaluation shaders, 34–36
 - examples
 - cubic Bézier patches, 304–310
 - terrain rendering, 300–304
 - isolines, 290–292
 - point mode, 292–294
 - primitive modes, 285–294
 - quads (quadrilaterals), 285–288
 - shaders, passing data, 296–299
 - subdivision modes, 294–296
 - triangles, 288–290
- tessellation control shader. *See* TCS
- tessellation evaluation shader. *See* TES
- tests
 - depth, 46, 351–355
 - early testing, 355–357
 - FBOs (user-defined framebuffers), 379
 - pre-fragment, 345–357
 - scissor, 46, 345–348
 - stencil, 46, 348–351
- texels as light, 545–547
- texture buffer object. *See* TBO
- textures, 19, 137–185
 - 1D, 244
 - arrays, 160–165, 163–165, 370
 - attaching, 367
 - base level, 154
 - border color, 159
 - compression, 177–181, 606
 - coordinates, 146–148, 529
 - copying data into a, 433–434
 - files, loading from, 144–148
 - filtering, 151–153
 - floating-point, 402–403
 - formatting, 138–139
 - initializing, 138–139
 - linear, 245
 - loading, 665–667
 - max level, 154
 - multiple, 150–151
 - multi-sample, 389–393
 - objects, 160
 - OpenGL ES 3.0, 712

- textures (*continued*)
 - points, 420
 - a point sprite in the fragment shader listing (9.30), 420
 - reading, 148–165, 434–435
 - rendering, 610
 - shaders
 - reading from in, 141–144
 - writing to in, 165–176
 - stars, 420
 - targets, 139–140
 - TBO (texture buffer object), 266, 269
 - views, 181–185
 - wrap mode, 158–160
 - threads, multiple, 611–613
 - three-component vertices, 53
 - tightly packed arrays, 101
 - timer queries, 484–487
 - timeslicing, 446
 - timing operations
 - using `glQueryCounter()` listing (11.8), 485–486
 - using timer queries listing (11.7), 484–485
 - tokens, 96, 226
 - tone mapping, 404–409
 - tools, 759–764
 - `dds2ktx` utility, 761–762
 - GLUT (OpenGL Utility Toolkit), 680–681
 - `ktxtool` utility, 759–761
 - performance analysis, 589–597
 - Realtech VR OpenGL Extensions Viewer, 619
 - `sb6mtool` utility, 762–764
 - toon fragment shader listing (12.26), 546–547
 - toon vertex shader listing (12.25), 546
 - transfer curves, tone mapping, 407
 - transformations
 - concatenation, 73–75
 - coordinates, 66–73
 - coordinate spaces, 63–66
 - geometry, 63
 - matrices, 62, 66
 - rotation, 70–72
 - scaling, 72–73
 - perspective, 68–70
 - models, 63
 - model-view, 76–79, 667
 - multiple viewport transformations, 336–340
 - order of, 105
 - overview of, 63–82
 - projection, 79–81
 - uniforms, geometry, 121–126
 - vertices, storage, 259–275
 - view, 76
 - viewports, 76
- transform feedback
 - applying, 260–265
 - ending the pipeline with, 266
 - physical simulation example, 266–275
 - queries, 487–493
 - starting, pausing, and stopping, 264–266
 - translation, 68–70
 - transparency, 558
 - transposing images, 412
 - traversing a linked-list in a fragment shader listing (5.46), 175–176
 - triangles, 10
 - clipping, 277, 278
 - drawing, 24–25
 - `GL_TRIANGLES_ADJACENCY` primitive mode, 330, 331
 - guard bands, 278–279
 - rendering without, 565–580
 - tessellation using, 288–290
 - troubleshooting, 581
 - tuning applications for speed, 597–616
 - turning on line smoothing listing (9.18), 386
 - types
 - buffers, assigning to, 92
 - data, 188–194
 - elements, 193
 - matrices, 190–192
 - of projection transformations, 79–81
 - samplers, 132
 - scalars, 188–189
 - textures, 139–140
 - tokens, 96
 - vectors, 190–192
 - vertex attributes, 226
 - UBO (Uniform Buffer Object), 108
 - unary negation, 76
 - under sampling data, 384
 - uniform blocks binding layout qualifiers listing (5.19), 119
 - Uniform Buffer Object. *See* UBO
 - uniforms, 103–126
 - arrays, 106
 - blocks, 108–121
 - buffers, 92, 109
 - default block, 104–105
 - geometry, 121–126
 - matrices, 107
 - subroutines, 213, 215
 - units
 - images, 167
 - textures, 137
 - vectors, 52
 - unpacking data from a G-buffer listing (12.29), 552
 - unsignaled states, 494
 - unsigned integers, 189
 - updating
 - the content of buffers listing (5.2), 94
 - depth buffers, 352–353

- projection matrices listing (5.22), 123
- stencil buffers, 351
- texture data listing (5.34), 138–139
- uniforms, 108
- vertex attributes listing (3.2), 29
- user-defined clipping, 279–282
- user-defined framebuffers. *See* FBOs
- using. *See also* applying
 - attributes in vertex shaders listing (5.5), 99
 - a function to produce faces in a geometry shader listing (8.29), 324
 - a gradient texture to color a julia set listing (12.35), 568
 - results of atomic counters in uniform blocks listing (5.32), 136
 - shader storage blocks in place of vertex attributes listing (5.28), 127–128
 - shader storage blocks listing (5.28), 127–128
- utilities. *See* tools
- values, 15
 - averaging, 456
 - coverage, 392
 - interpolation, 44
 - normalization, 226
 - primitiveMode, 265
 - return, framebuffer completeness, 378
- Van Verth, James, 718
- VAOs (vertex array objects), 20, 97, 272, 595
- variables
 - built-in, 24
 - filtering, 457
 - images, 165
 - members, 21
 - point sizes, 230
 - sampler, 141
 - sharing, 444
- varyings, 260, 261
 - centroid sampling, 395
 - per-patch user-defined, 284
- vectors, 15, 51–54, 190–192
 - bittangent, 519
 - Boolean, 196
 - built-in functions, 195–197
 - homogenous, 53
 - length of, 57
 - operators, 54–58
 - Phong lighting, 506
 - products, 55
 - reflection, 507
 - rim lighting, 516
 - tangent, 519
 - uniforms, 105–106
 - unit, 52
- vendor extensions, 618, 728
- versions, 7
 - development (OpenGL ES), 707–708
- vertex array objects. *See* VAOs
- vertex shaders, 22, 24, 595. *See also* shaders
 - for the Alien Rain sample (5.41), 162–163
 - with an output listing (3.3), (3.11), 30, 44
 - for cube map environment rendering listing (12.15), 531
 - feeding from buffers, 97–103
 - inputs, 224–229
 - inputs for Asteroids listing (7.12), 255
 - multiple inputs, 100–102
 - for normal mapping listing (12.7), 520
 - with output interface blocks listing (3.5), 31
 - outputs, 229–230
 - with single texture coordinate listing (5.38), 147
 - for sky box rendering listing (12.13), 530
 - for the star field effect listing (9.31), 422
 - for terrain rendering listing (8.8), 301
- VERTEX structure, 225
- vertices
 - attributes, 28–29, 97
 - basevertex, adding, 234–235
 - buffers, 92
 - clipping, 38–39
 - connections, 267
 - data chunks (SBM model file format), 753–755
 - fetching, 28
 - multiple, 24
 - OpenGL ES 3.0, 709
 - per-patch, 284n1
 - pipelines
 - passing data from stage to stage, 29–32
 - passing data to shaders, 28–29
 - processing, 224–230
 - shaders. *See* vertex shaders
 - transforms, storage, 259–275
- viewing
 - images in stereo, 380
 - normals, 328
 - Realtech VR OpenGL Extensions Viewer, 619
 - retina displays (Mac OS X), 673–674
 - X Window System, 704
- viewports
 - managing, 660
 - multiple, 336–340
 - transformation, 39–40
- views
 - adding, 650–652
 - coordinates, 65–66
 - model-view transforms, 667
 - space, 64, 555n6
 - textures, 181–185
 - transformations, 76–79
- visuals
 - config management and, 689–693
 - tearing, 646, 678

- `vmath::perspective` function, 82
- `vmath::rotate` function, 72
- volume, 140
 - clipping, 276
 - local work groups, 441
- `vsync`, 591
- WAR (Write-After-Read), 131
- warnings, shaders, 203
- WAW (Write-After-Write), 131
- web sites, 748–749
- WGF (Windows Graphics Foundation), 627
- WGL (Windows-GL), 623, 634–639
 - `wglChoosePixelFormatARB()`, 638–641
 - `wglCreateContext()`, 633, 635, 641
 - `wglCreateContextAttribsARB()`, 641, 642, 643, 646
 - `wglDeleteContext()`, 646
 - `wglGetExtensionsStringARB()`, 635
 - `wglGetPixelFormatAttribARB()`, 644
 - `wglGetPixelFormatAttribfvARB()`, 640
 - `wglGetPixelFormatAttribivARB()`, 640, 644
 - `wglGetProcAddress()`, 636, 646, 725
 - `wglMakeCurrent()`, 611, 612, 633
 - `wglSwapIntervalEXT()`, 646
- whole framebuffer completeness, 377
- Win32, 623
- winding order, 41, 296
- windows
 - child windows (in Cocoa), 671
 - coordinates, 40
 - EGL, 720
 - formatting, 628–630
 - GLX, 701–704
 - Linux, 693–694
 - space, 64
 - surfaces, 626
- Windows-GL. *See* WGL
- Windows Graphics Foundation. *See* WGF
- Windows main message loop listing (14.5), 633
- Windows operating systems, 623–647
 - basic setup, 627–632
 - graphics drivers, 624–626
 - Windows 95, 623
 - Windows NT version 3.5, 623
 - Windows Vista, 626
- Windows Performance Toolkit. *See* WPT
- Windows Presentation Foundation. *See* WPF
- workgroups, 47
 - maximum size of, 440–441
- working while waiting for a sync object listing (11.10), 495
- work items, 493
- world coordinates, 65
- world space, 64
- WPF (Windows Presentation Foundation), 627
- WPT (Windows Performance Toolkit), 590–594
- wrapping
 - modes, 148
 - textures, 147, 148, 158–160
- Write-After-Read (WAR), 131
- Write-After-Write (WAW), 131
- writing to a G-buffer listing (12.28), 551
- `XCLOSEDisplay()`, 689
- `Xcode`, 649, 739–740
- `XCreateWindow()`, 693, 694, 702
- `XDestroyWindow()`, 694
- `XFree()`, 692, 693
- `XOpenDisplay()`, 689, 690
- XOR operator, 47
- X Window System, 683
 - display objects and, 689
 - full-screen views, 704
 - GLX-interfacing with, 688–689
- y-axis, 243
- z-axis, 76
- zeroes, 24