

OpenGL[®]

Programming Guide

Eighth Edition

*The Official Guide to Learning
OpenGL[®], Version 4.3*



Dave Shreiner • Graham Sellers • John Kessenich • Bill Licea-Kane

The Khronos OpenGL ARB Working Group

FREE SAMPLE CHAPTER



SHARE WITH OTHERS

Praise for OpenGL[®] Programming Guide, Eighth Edition

“Wow! This book is basically one-stop shopping for OpenGL information. It is the kind of book that I will be reaching for a lot. Thanks to Dave, Graham, John, and Bill for an amazing effort.”

—Mike Bailey, professor, Oregon State University

“The most recent Red Book parallels the grand tradition of OpenGL; continuous evolution towards ever-greater power and efficiency. The eighth edition contains up-to-the-minute information about the latest standard and new features, along with a solid grounding in modern OpenGL techniques that will work anywhere. The Red Book continues to be an essential reference for all new employees at my simulation company. What else can be said about this essential guide? I laughed, I cried, it was much better than *Cats*—I’ll read it again and again.”

—Bob Kuehne, president, Blue Newt Software

“OpenGL has undergone enormous changes since its inception twenty years ago. This new edition is your practical guide to using the OpenGL of today. Modern OpenGL is centered on the use of shaders, and this edition of the Programming Guide jumps right in, with shaders covered in depth in Chapter 2. It continues in later chapters with even more specifics on everything from texturing to compute shaders. No matter how well you know it or how long you’ve been doing it, if you are going to write an OpenGL program, you want to have a copy of the OpenGL[®] Programming Guide handy.”

—Marc Olano, associate professor, UMBC

“If you are looking for the definitive guide to programming with the very latest version of OpenGL, look no further. The authors of this book have been deeply involved in the creation of OpenGL 4.3, and everything you need to know about the cutting edge of this industry-leading API is laid out here in a clear, logical, and insightful manner.”

—Neil Trevett, president, Khronos Group

This page intentionally left blank

OpenGL[®] Programming Guide Eighth Edition

OpenGL® Series



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.

PEARSON

Addison-Wesley Cisco Press EXAM/CRAM IBM Press que PRENTICE HALL SAMS | Safari

OpenGL[®] Programming Guide Eighth Edition

*The Official Guide to
Learning OpenGL[®], Version 4.3*

*Dave Shreiner
Graham Sellers
John Kessenich
Bill Licea-Kane*

The Khronos OpenGL ARB Working Group

◆ 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

OpenGL programming guide : the official guide to learning OpenGL, version 4.3 /
Dave Shreiner, Graham Sellers, John Kessenich, Bill Licea-Kane ; the Khronos OpenGL
ARB Working Group.—Eighth edition.

pages cm

Includes index.

ISBN 978-0-321-77303-6 (pbk. : alk. paper)

1. Computer graphics. 2. OpenGL. I. Shreiner, Dave. II. Sellers, Graham.

III. Kessenich, John M. IV. Licea-Kane, Bill. V. Khronos OpenGL ARB Working Group.

T385.O635 2013

006.6'63—dc23

2012043324

Copyright © 2013 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: 978-0-321-77303-6

ISBN-10: 0-321-77303-9

Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan.

First printing, March 2013

*For my family—Vicki, Bonnie, Bob, Cookie, Goatee, Phantom, Squiggles,
Tuxedo, and Toby.*

—DRS

To Emily: welcome, we're so glad you're here! Chris and J.: you still rock!

—GJAS

In memory of Phil Karlton, Celeste Fowler, Joan Eslinger, and Ben Cheatham.

This page intentionally left blank

Contents

Figures	xxiii
Tables	xxix
Examples	xxxiii
About This Guide	xli
What This Guide Contains	xli
What's New in This Edition	xliii
What You Should Know Before Reading This Guide	xliii
How to Obtain the Sample Code	xliv
Errata.....	xlv
Style Conventions.....	xlv
1. Introduction to OpenGL	1
What Is OpenGL?	2
Your First Look at an OpenGL Program.....	3
OpenGL Syntax	8
OpenGL's Rendering Pipeline.....	10
Preparing to Send Data to OpenGL.....	11
Sending Data to OpenGL.....	11
Vertex Shading	12
Tessellation Shading	12
Geometry Shading.....	12
Primitive Assembly	12
Clipping	13
Rasterization	13
Fragment Shading	13

Per-Fragment Operations	13
Our First Program: A Detailed Discussion	14
Entering main()	14
OpenGL Initialization	16
Our First OpenGL Rendering	28
2. Shader Fundamentals	33
Shaders and OpenGL	34
OpenGL's Programmable Pipeline	35
An Overview of the OpenGL Shading Language	37
Creating Shaders with GLSL	37
Storage Qualifiers	45
Statements	49
Computational Invariance	54
Shader Preprocessor	56
Compiler Control	58
Global Shader-Compilation Option	59
Interface Blocks	60
Uniform Blocks	61
Specifying Uniform Blocks in Shaders	61
Accessing Uniform Blocks from Your Application	63
Buffer Blocks	69
In/Out Blocks	70
Compiling Shaders	70
Our LoadShaders() Function	76
Shader Subroutines	76
GLSL Subroutine Setup	77
Selecting Shader Subroutines	78
Separate Shader Objects	81
3. Drawing with OpenGL	85
OpenGL Graphics Primitives	86
Points	87
Lines, Strips, and Loops	88
Triangles, Strips, and Fans	89
Data in OpenGL Buffers	92
Creating and Allocating Buffers	92
Getting Data into and out of Buffers	95

Accessing the Content of Buffers.....	100
Discarding Buffer Data	107
Vertex Specification	108
VertexAttribPointer in Depth	108
Static Vertex-Attribute Specification.....	112
OpenGL Drawing Commands	115
Restarting Primitives	124
Instanced Rendering	128
Instanced Vertex Attributes	129
Using the Instance Counter in Shaders.....	136
Instancing Redux	139
4. Color, Pixels, and Framebuffers	141
Basic Color Theory	142
Buffers and Their Uses	144
Clearing Buffers	146
Masking Buffers	147
Color and OpenGL	148
Color Representation and OpenGL.....	149
Vertex Colors.....	150
Rasterization	153
Multisampling.....	153
Sample Shading	155
Testing and Operating on Fragments	156
Scissor Test	157
Multisample Fragment Operations	158
Stencil Test.....	159
Stencil Examples	161
Depth Test	163
Blending.....	166
Blending Factors	167
Controlling Blending Factors.....	167
The Blending Equation.....	170
Dithering	171
Logical Operations	171
Occlusion Query	173
Conditional Rendering.....	176
Per-Primitive Antialiasing.....	178

Antialiasing Lines	179
Antialiasing Polygons	180
Framebuffer Objects	180
Renderbuffers	183
Creating Renderbuffer Storage	185
Framebuffer Attachments	187
Framebuffer Completeness	190
Invalidating Framebuffers	192
Writing to Multiple Renderbuffers Simultaneously	193
Selecting Color Buffers for Writing and Reading	195
Dual-Source Blending	198
Reading and Copying Pixel Data	200
Copying Pixel Rectangles	203
5. Viewing Transformations, Clipping, and Feedback	205
Viewing	206
Viewing Model	207
Camera Model	207
Orthographic Viewing Model	212
User Transformations	212
Matrix Multiply Refresher	214
Homogeneous Coordinates	215
Linear Transformations and Matrices	219
Transforming Normals	231
OpenGL Matrices	232
OpenGL Transformations	236
Advanced: User Clipping	238
Transform Feedback	239
Transform Feedback Objects	239
Transform Feedback Buffers	241
Configuring Transform Feedback Varyings	244
Starting and Stopping Transform Feedback	250
Transform Feedback Example—Particle System	252
6. Textures	259
Texture Mapping	261
Basic Texture Types	262
Creating and Initializing Textures	263

Texture Formats	270
Proxy Textures.....	276
Specifying Texture Data	277
Explicitly Setting Texture Data.....	277
Using Pixel Unpack Buffers	280
Copying Data from the Framebuffer	281
Loading Images from Files	282
Retrieving Texture Data	287
Texture Data Layout	288
Sampler Objects.....	292
Sampler Parameters	294
Using Textures	295
Texture Coordinates.....	298
Arranging Texture Data	302
Using Multiple Textures.....	303
Complex Texture Types.....	306
3D Textures	307
Array Textures	309
Cube-Map Textures.....	309
Shadow Samplers	317
Depth-Stencil Textures	318
Buffer Textures.....	319
Texture Views.....	321
Compressed Textures.....	326
Filtering	329
Linear Filtering	330
Using and Generating Mipmaps.....	333
Calculating the Mipmap Level.....	338
Mipmap Level-of-Detail Control	339
Advanced Texture Lookup Functions.....	340
Explicit Level of Detail	340
Explicit Gradient Specification	340
Texture Fetch with Offsets	341
Projective Texturing	342
Texture Queries in Shaders	343
Gathering Texels	345
Combining Special Functions.....	345
Point Sprites	346

Textured Point Sprites	347
Controlling the Appearance of Points	350
Rendering to Texture Maps	351
Discarding Rendered Data	354
Chapter Summary	356
Texture Redux	356
Texture Best Practices	357
7. Light and Shadow	359
Lighting Introduction	360
Classic Lighting Model	361
Fragment Shaders for Different Light Styles	362
Moving Calculations to the Vertex Shader	373
Multiple Lights and Materials	376
Lighting Coordinate Systems	383
Limitations of the Classic Lighting Model	383
Advanced Lighting Models	384
Hemisphere Lighting	384
Image-Based Lighting	389
Lighting with Spherical Harmonics	395
Shadow Mapping	400
Creating a Shadow Map	401
8. Procedural Texturing	411
Procedural Texturing	412
Regular Patterns	414
Toy Ball	422
Lattice	431
Procedural Shading Summary	432
Bump Mapping	433
Application Setup	436
Vertex Shader	438
Fragment Shader	439
Normal Maps	441
Antialiasing Procedural Textures	442
Sources of Aliasing	442
Avoiding Aliasing	444

Increasing Resolution	445
Antialiasing High Frequencies	447
Frequency Clamping.....	457
Procedural Antialiasing Summary.....	459
Noise	460
Definition of Noise	461
Noise Textures	468
Trade-offs.....	471
A Simple Noise Shader	472
Turbulence	475
Marble.....	477
Granite.....	478
Wood	478
Noise Summary.....	483
Further Information	483
9. Tessellation Shaders.....	485
Tessellation Shaders.....	486
Tessellation Patches.....	487
Tessellation Control Shaders	488
Generating Output-Patch Vertices	489
Tessellation Control Shader Variables.....	490
Controlling Tessellation	491
Tessellation Evaluation Shaders	496
Specifying the Primitive Generation Domain	497
Specifying the Face Winding for Generated Primitives	497
Specifying the Spacing of Tessellation Coordinates.....	498
Additional Tessellation Evaluation Shader layout Options	498
Specifying a Vertex's Position	498
Tessellation Evaluation Shader Variables.....	499
A Tessellation Example: The Teapot	500
Processing Patch Input Vertices.....	501
Evaluating Tessellation Coordinates for the Teapot.....	501
Additional Tessellation Techniques	504
View-Dependent Tessellation.....	504
Shared Tessellated Edges and Cracking	506
Displacement Mapping	507

10. Geometry Shaders	509
Creating a Geometry Shader	510
Geometry Shader Inputs and Outputs	514
Geometry Shader Inputs	514
Special Geometry Shader Primitives	517
Geometry Shader Outputs	523
Producing Primitives	525
Culling Geometry	525
Geometry Amplification	527
Advanced Transform Feedback	532
Multiple Output Streams	533
Primitive Queries	537
Using Transform Feedback Results	539
Geometry Shader Instancing	549
Multiple Viewports and Layered Rendering	550
Viewport Index	550
Layered Rendering	556
Chapter Summary	559
Geometry Shader Redux	560
Geometry Shader Best Practices	561
11. Memory	563
Using Textures for Generic Data Storage	564
Binding Textures to Image Units	569
Reading from and Writing to Images	572
Shader Storage Buffer Objects	576
Writing Structured Data	577
Atomic Operations and Synchronization	578
Atomic Operations on Images	578
Atomic Operations on Buffers	587
Sync Objects	589
Image Qualifiers and Barriers	593
High Performance Atomic Counters	605
Example	609
Order-Independent Transparency	609

12. Compute Shaders	623
Overview.....	624
Workgroups and Dispatch	625
Knowing Where You Are	630
Communication and Synchronization.....	632
Communication	633
Synchronization	634
Examples.....	636
Physical Simulation	636
Image Processing.....	642
Chapter Summary.....	647
Compute Shader Redux.....	647
Compute Shader Best Practices	648
A. Basics of GLUT: The OpenGL Utility Toolkit	651
Initializing and Creating a Window	652
Accessing Functions	654
Handling Window and Input Events	655
Managing a Background Process.....	658
Running the Program	658
B. OpenGL ES and WebGL	659
OpenGL ES.....	660
WebGL	662
Setting up WebGL within an HTML5 page	662
Initializing Shaders in WebGL	664
Initializing Vertex Data in WebGL.....	667
Using Texture Maps in WebGL.....	668
C. Built-in GLSL Variables and Functions	673
Built-in Variables	674
Built-in Variable Declarations	674
Built-in Variable Descriptions	676
Built-in Constants.....	684
Built-in Functions.....	686
Angle and Trigonometry Functions	688
Exponential Functions	690
Common Functions.....	692
Floating-Point Pack and Unpack Functions	698

Geometric Functions	700
Matrix Functions.....	702
Vector Relational Functions	703
Integer Functions	705
Texture Functions.....	708
Atomic-Counter Functions.....	722
Atomic Memory Functions	723
Image Functions	725
Fragment Processing Functions.....	729
Noise Functions	731
Geometry Shader Functions	732
Shader Invocation Control Functions	734
Shader Memory Control Functions.....	734
D. State Variables.....	737
The Query Commands.....	738
OpenGL State Variables.....	745
Current Values and Associated Data.....	746
Vertex Array Object State	747
Vertex Array Data	749
Buffer Object State.....	750
Transformation State.....	751
Coloring State.....	752
Rasterization State	753
Multisampling	755
Textures.....	756
Textures.....	759
Textures.....	762
Textures.....	764
Texture Environment	766
Pixel Operations.....	767
Framebuffer Controls	770
Framebuffer State	771
Framebuffer State	772
Framebuffer State.....	773
Renderbuffer State	775
Renderbuffer State	776
Pixel State	778

Shader Object State.....	781
Shader Program Pipeline Object State	782
Shader Program Object State	783
Program Interface State	793
Program Object Resource State.....	794
Vertex and Geometry Shader State	797
Query Object State	797
Image State	798
Transform Feedback State	799
Atomic Counter State.....	800
Shader Storage Buffer State.....	801
Sync Object State	802
Hints.....	803
Compute Dispatch State	803
Implementation-Dependent Values	804
Tessellation Shader Implementation-Dependent Limits.....	810
Geometry Shader Implementation-Dependent Limits	813
Fragment Shader Implementation-Dependent Limits.....	815
Implementation-Dependent Compute Shader Limits.....	816
Implementation-Dependent Shader Limits	818
Implementation-Dependent Debug Output State	823
Implementation-Dependent Values	824
Internal Format-Dependent Values.....	826
Implementation-Dependent Transform Feedback Limits	826
Framebuffer-Dependent Values	827
Miscellaneous	827
E. Homogeneous Coordinates and Transformation Matrices	829
Homogeneous Coordinates.....	830
Transforming Vertices	830
Transforming Normals	831
Transformation Matrices	831
Translation.....	832
Scaling	832
Rotation	832
Perspective Projection	834
Orthographic Projection.....	834

F. OpenGL and Window Systems	835
Accessing New OpenGL Functions	836
GLEW: The OpenGL Extension Wrangler	837
GLX: OpenGL Extension for the X Window System	838
Initialization	839
Controlling Rendering	840
GLX Prototypes	842
WGL: OpenGL Extensions for Microsoft Windows	845
Initialization	846
Controlling Rendering	846
WGL Prototypes.....	848
OpenGL in Mac OS X: The Core OpenGL (CGL) API and the NSOpenGL Classes	850
Mac OS X's Core OpenGL Library	851
Initialization	851
Controlling Rendering	852
CGL Prototypes.....	852
The NSOpenGL Classes	854
Initialization	854
 G. Floating-Point Formats for Textures, Framebuffers, and Renderbuffers.....	 857
Reduced-Precision Floating-Point Values	858
16-bit Floating-Point Values.....	858
10- and 11-bit Unsigned Floating-Point Values.....	860
 H. Debugging and Profiling OpenGL	 865
Creating a Debug Context	866
Debug Output	868
Debug Messages	869
Filtering Messages	872
Application-Generated Messages	874
Debug Groups	875
Naming Objects	877
Profiling.....	879
Profiling Tools	879
In-Application Profiling.....	881

I. Buffer Object Layouts	885
Using Standard Layout Qualifiers.....	886
The <code>std140</code> Layout Rules.....	886
The <code>std430</code> Layout Rules.....	887
Glossary	889
Index	919

This page intentionally left blank

Figures

Figure 1.1	Image from our first OpenGL program: triangles.cpp	5
Figure 1.2	The OpenGL pipeline	10
Figure 2.1	Shader-compilation command sequence	71
Figure 3.1	Vertex layout for a triangle strip	89
Figure 3.2	Vertex layout for a triangle fan	90
Figure 3.3	Packing of elements in a BGRA-packed vertex attribute	112
Figure 3.4	Packing of elements in a RGBA-packed vertex attribute	112
Figure 3.5	Simple example of drawing commands	124
Figure 3.6	Using primitive restart to break a triangle strip	125
Figure 3.7	Two triangle strips forming a cube	127
Figure 3.8	Result of rendering with instanced vertex attributes	134
Figure 3.9	Result of instanced rendering using <code>gl_InstanceID</code>	139
Figure 4.1	Region occupied by a pixel	144
Figure 4.2	Polygons and their depth slopes	165
Figure 4.3	Aliased and antialiased lines	178
Figure 4.4	Close-up of RGB color elements in an LCD panel	199
Figure 5.1	Steps in configuring and positioning the viewing frustum	207
Figure 5.2	Coordinate systems required by OpenGL	209
Figure 5.3	User coordinate systems unseen by OpenGL	210
Figure 5.4	A view frustum	211
Figure 5.5	Pipeline subset for user/shader part of transforming coordinates	212
Figure 5.6	One-dimensional homogeneous space	217

Figure 5.7	Translating by skewing	218
Figure 5.8	Translating an object 2.5 in the x direction.....	220
Figure 5.9	Scaling an object to three times its size	221
Figure 5.10	Scaling an object in place	223
Figure 5.11	Rotation	225
Figure 5.12	Rotating in place	225
Figure 5.13	Frustum projection	228
Figure 5.14	Orthographic projection	230
Figure 5.15	z precision	237
Figure 5.16	Transform feedback varyings packed in a single buffer....	246
Figure 5.17	Transform feedback varyings packed in separate buffers	246
Figure 5.18	Transform feedback varyings packed into multiple buffers	250
Figure 5.19	Schematic of the particle system simulator	253
Figure 5.20	Result of the particle system simulator.....	258
Figure 6.1	Byte-swap effect on byte, short, and integer data	289
Figure 6.2	Subimage	290
Figure 6.3	*IMAGE_HEIGHT pixel storage mode	291
Figure 6.4	*SKIP_IMAGES pixel storage mode	292
Figure 6.5	Output of the simple textured quad example.....	299
Figure 6.6	Effect of different texture wrapping modes	301
Figure 6.7	Two textures used in the multitexture example	306
Figure 6.8	Output of the simple multitexture example.....	306
Figure 6.9	Output of the volume texture example	308
Figure 6.10	A sky box	312
Figure 6.11	A golden environment mapped torus	315
Figure 6.12	A visible seam in a cube map	316
Figure 6.13	The effect of seamless cube-map filtering	317
Figure 6.14	Effect of texture minification and magnification	330
Figure 6.15	Resampling of a signal in one dimension	330
Figure 6.16	Bilinear resampling	331
Figure 6.17	A pre-filtered mipmap pyramid	334
Figure 6.18	Effects of minification mipmap filters.....	335
Figure 6.19	Illustration of mipmaps using unrelated colors	336
Figure 6.20	Result of the simple textured point sprite example	348

Figure 6.21	Analytically calculated point sprites	349
Figure 6.22	Smooth edges of circular point sprites	349
Figure 7.1	Elements of the classic lighting model	361
Figure 7.2	A sphere illuminated using the hemisphere lighting model	386
Figure 7.3	Analytic hemisphere lighting function	387
Figure 7.4	Lighting model comparison	388
Figure 7.5	Light probe image	391
Figure 7.6	Lat-long map	391
Figure 7.7	Cube map	392
Figure 7.8	Effects of diffuse and specular environment maps	394
Figure 7.9	Spherical harmonics lighting	400
Figure 7.10	Depth rendering	405
Figure 7.11	Final rendering of shadow map	409
Figure 8.1	Procedurally striped torus	415
Figure 8.2	Stripes close-up	419
Figure 8.3	Brick patterns	420
Figure 8.4	Visualizing the results of the half-space distance calculations	427
Figure 8.5	Intermediate results from the toy ball shader	428
Figure 8.6	Intermediate results from “in” or “out” computation	429
Figure 8.7	The lattice shader applied to the cow model	432
Figure 8.8	Inconsistently defined tangents leading to large lighting errors	437
Figure 8.9	Simple box and torus with procedural bump mapping ...	441
Figure 8.10	Normal mapping	442
Figure 8.11	Aliasing artifacts caused by point sampling	444
Figure 8.12	Supersampling	446
Figure 8.13	Using the <i>s</i> texture coordinate to create stripes on a sphere	448
Figure 8.14	Antialiasing the stripe pattern	449
Figure 8.15	Visualizing the gradient	451
Figure 8.16	Effect of adaptive analytical antialiasing on striped teapots	452
Figure 8.17	Periodic step function	454
Figure 8.18	Periodic step function (pulse train) and its integral	454

Figure 8.19	Brick shader with and without antialiasing.....	456
Figure 8.20	Checkerboard pattern.....	458
Figure 8.21	A discrete 1D noise function.....	462
Figure 8.22	A continuous 1D noise function.....	463
Figure 8.23	Varying the frequency and the amplitude of the noise function.....	464
Figure 8.24	Summing noise functions.....	465
Figure 8.25	Basic 2D noise, at frequencies 4, 8, 16, and 32.....	467
Figure 8.26	Summed noise, at 1, 2, 3, and 4 octaves.....	467
Figure 8.27	Teapots rendered with noise shaders.....	475
Figure 8.28	Absolute value noise or “turbulence”.....	476
Figure 8.29	A bust of Beethoven rendered with the wood shader.....	482
Figure 9.1	Quad tessellation.....	492
Figure 9.2	Isoline tessellation.....	494
Figure 9.3	Triangle tessellation.....	495
Figure 9.4	Even and odd tessellation.....	496
Figure 9.5	The tessellated patches of the teapot.....	502
Figure 9.6	Tessellation cracking.....	507
Figure 10.1	Lines adjacency sequence.....	518
Figure 10.2	Line-strip adjacency sequence.....	519
Figure 10.3	Triangles adjacency sequence.....	520
Figure 10.4	Triangle-strip adjacency layout.....	521
Figure 10.5	Triangle-strip adjacency sequence.....	522
Figure 10.6	Texture used to represent hairs in the fur rendering example.....	530
Figure 10.7	The output of the fur rendering example.....	531
Figure 10.8	Schematic of geometry shader sorting example.....	546
Figure 10.9	Final output of geometry shader sorting example.....	548
Figure 10.10	Output of the viewport-array example.....	555
Figure 11.1	Output of the simple load-store shader.....	575
Figure 11.2	Timeline exhibited by the naïve overdraw counter shader.....	579
Figure 11.3	Output of the naïve overdraw counter shader.....	580
Figure 11.4	Output of the atomic overdraw counter shader.....	582
Figure 11.5	Cache hierarchy of a fictitious GPU.....	597

Figure 11.6	Data structures used for order-independent transparency	610
Figure 11.7	Inserting an item into the per-pixel linked lists	616
Figure 11.8	Result of order-independent transparency incorrect order on left; correct order on right.....	621
Figure 12.1	Schematic of a compute workload	626
Figure 12.2	Relationship of global and local invocation ID.....	632
Figure 12.3	Output of the physical simulation program as simple points	640
Figure 12.4	Output of the physical simulation program.....	642
Figure 12.5	Image processing	646
Figure 12.6	Image processing artifacts.....	647
Figure B.1	Our WebGL demo	671
Figure H.1	AMD's GPUPerfStudio2 profiling Unigine Heaven 3.0	880
Figure H.2	Screenshot of Unigine Heaven 3.0	880

This page intentionally left blank

Tables

Table 1.1	Command Suffixes and Argument Data Types	10
Table 1.2	Example of Determining Parameters for glVertexAttribPointer()	26
Table 1.3	Clearing Buffers	28
Table 2.1	Basic Data Types in GLSL	38
Table 2.2	Implicit Conversions in GLSL	39
Table 2.3	GLSL Vector and Matrix Types	40
Table 2.4	Vector Component Accessors	43
Table 2.5	GLSL Type Modifiers	46
Table 2.6	GLSL Operators and Their Precedence	50
Table 2.7	GLSL Flow-Control Statements	52
Table 2.8	GLSL Function Parameter Access Modifiers	54
Table 2.9	GLSL Preprocessor Directives	57
Table 2.10	GLSL Preprocessor Predefined Macros	58
Table 2.11	GLSL Extension Directive Modifiers	60
Table 2.12	Layout Qualifiers for Uniform	62
Table 3.1	OpenGL Primitive Mode Tokens	90
Table 3.2	Buffer Binding Targets	93
Table 3.3	Buffer Usage Tokens	96
Table 3.4	Access Modes for glMapBuffer()	101
Table 3.5	Flags for Use with glMapBufferRange()	104
Table 3.6	Values of <i>Type</i> for glVertexAttribPointer()	109
Table 4.1	Converting Data Values to Normalized Floating-Point Values	150
Table 4.2	Query Values for the Stencil Test	161
Table 4.3	Source and Destination Blending Factors	169

Table 4.4	Blending Equation Mathematical Operations.....	171
Table 4.5	Sixteen Logical Operations	172
Table 4.6	Values for Use with glHint()	179
Table 4.7	Framebuffer Attachments	187
Table 4.8	Errors Returned by glCheckFramebufferStatus()	191
Table 4.9	glReadPixels() Data Formats	201
Table 4.10	Data Types for glReadPixels()	202
Table 5.1	Drawing Modes Allowed During Transform Feedback.....	251
Table 6.1	Texture Targets and Corresponding Sampler Types.....	263
Table 6.2	Sized Internal Formats	271
Table 6.3	External Texture Formats.....	274
Table 6.4	Example Component Layouts for Packed Pixel Formats.....	276
Table 6.5	Texture Targets and Corresponding Proxy Targets	276
Table 6.6	Target Compatibility for Texture Views	322
Table 6.7	Internal Format Compatibility for Texture Views	323
Table 7.1	Spherical Harmonic Coefficients for Light Probe Images	397
Table 9.1	Tessellation Control Shader Input Variables.....	490
Table 9.2	Evaluation Shader Primitive Types	497
Table 9.3	Options for Controlling Tessellation Level Effects.....	498
Table 9.4	Tessellation Control Shader Input Variables.....	500
Table 10.1	Geometry Shader Primitive Types and Accepted Drawing Modes.....	513
Table 10.2	Geometry Shader Primitives and the Vertex Count for Each	515
Table 10.3	Provoking Vertex Selection by Primitive Mode	524
Table 10.4	Ordering of Cube-Map Face Indices.....	559
Table 11.1	Generic Image Types in GLSL.....	565
Table 11.2	Image Format Qualifiers.....	566
Table B.1	Type Strings for WebGL Shaders	664
Table B.2	WebGL Typed Arrays	667
Table C.1	Cube-Map Face Targets	679
Table C.2	Notation for Argument or Return Type	687
Table D.1	Current Values and Associated Data	746
Table D.2	State Variables for Vertex Array Objects.....	747

Table D.3	State Variables for Vertex Array Data (Not Stored in a Vertex Array Object)	749
Table D.4	State Variables for Buffer Objects	750
Table D.5	Transformation State Variables	751
Table D.6	State Variables for Controlling Coloring.....	752
Table D.7	State Variables for Controlling Rasterization	753
Table D.8	State Variables for Multisampling	755
Table D.9	State Variables for Texture Units	756
Table D.10	State Variables for Texture Objects	759
Table D.11	State Variables for Texture Images.....	762
Table D.12	State Variables Per Texture Sampler Object	764
Table D.13	State Variables for Texture Environment and Generation	766
Table D.14	State Variables for Pixel Operations	767
Table D.15	State Variables Controlling Framebuffer Access and Values.....	770
Table D.16	State Variables for Framebuffers per Target	771
Table D.17	State Variables for Framebuffer Objects	772
Table D.18	State Variables for Framebuffer Attachments.....	773
Table D.19	Renderbuffer State.....	775
Table D.20	State Variables per Renderbuffer Object	776
Table D.21	State Variables Controlling Pixel Transfers.....	778
Table D.22	State Variables for Shader Objects	781
Table D.23	State Variables for Program Pipeline Object State	782
Table D.24	State Variables for Shader Program Objects	783
Table D.25	State Variables for Program Interfaces	793
Table D.26	State Variables for Program Object Resources	794
Table D.27	State Variables for Vertex and Geometry Shader State	797
Table D.28	State Variables for Query Objects	797
Table D.29	State Variables per Image Unit.....	798
Table D.30	State Variables for Transform Feedback	799
Table D.31	State Variables for Atomic Counters	800
Table D.32	State Variables for Shader Storage Buffers	801
Table D.33	State Variables for Sync Objects	802
Table D.34	Hints	803
Table D.35	State Variables for Compute Shader Dispatch.....	803

Table D.36	State Variables Based on Implementation-Dependent Values	804
Table D.37	State Variables for Implementation-Dependent Tessellation Shader Values	810
Table D.38	State Variables for Implementation-Dependent Geometry Shader Values	813
Table D.39	State Variables for Implementation-Dependent Fragment Shader Values	815
Table D.40	State Variables for Implementation-Dependent Compute Shader Limits	816
Table D.41	State Variables for Implementation-Dependent Shader Limits	818
Table D.42	State Variables for Debug Output State	823
Table D.43	Implementation-Dependent Values	824
Table D.44	Internal Format-Dependent Values	826
Table D.45	Implementation-Dependent Transform Feedback Limits	826
Table D.46	Framebuffer-Dependent Values	827
Table D.47	Miscellaneous State Values	827
Table G.1	Reduced-Precision Floating-Point Formats	858
Table I.1	std140 Layout Rules	886
Table I.2	std430 Layout Rules	887

Examples

Example 1.1	triangles.cpp: Our First OpenGL Program	5
Example 1.2	Vertex Shader for triangles.cpp: triangles.vert.....	23
Example 1.3	Fragment Shader for triangles.cpp: triangles.frag.....	25
Example 2.1	A Simple Vertex Shader	36
Example 2.2	Obtaining a Uniform Variable's Index and Assigning Values	48
Example 2.3	Declaring a Uniform Block.....	61
Example 2.4	Initializing Uniform Variables in a Named Uniform Block.....	65
Example 2.5	Static Shader Control Flow	77
Example 2.6	Declaring a Set of Subroutines	78
Example 3.1	Initializing a Buffer Object with <code>glBufferSubData()</code>	98
Example 3.2	Initializing a Buffer Object with <code>glMapBuffer()</code>	103
Example 3.3	Declaration of the <code>DrawArraysIndirectCommand</code> Structure.....	118
Example 3.4	Declaration of the <code>DrawElementsIndirectCommand</code> Structure.....	119
Example 3.5	Setting up for the Drawing Command Example.....	122
Example 3.6	Drawing Commands Example	123
Example 3.7	Intializing Data for a Cube Made of Two Triangle Strips.....	125
Example 3.8	Drawing a Cube Made of Two Triangle Strips Using Primitive Restart.....	127
Example 3.9	Vertex Shader Attributes for the Instancing Example	130
Example 3.10	Example Setup for Instanced Vertex Attributes	130
Example 3.11	Instanced Attributes Example Vertex Shader.....	132

Example 3.12	Instancing Example Drawing Code	132
Example 3.13	gl_VertexID Example Vertex Shader	136
Example 3.14	Example Setup for Instanced Vertex Attributes	138
Example 4.1	Specifying Vertex Color and Position Data: gouraud.cpp	150
Example 4.2	A Simple Vertex Shader for Gouraud Shading	152
Example 4.3	A Simple Fragment Shader for Gouraud Shading	152
Example 4.4	A Multisample-Aware Fragment Shader	155
Example 4.5	Using the Stencil Test: stencil.c	161
Example 4.6	Rendering Geometry with Occlusion Query: occquery.c	174
Example 4.7	Retrieving the Results of an Occlusion Query	175
Example 4.8	Rendering Using Conditional Rendering	177
Example 4.9	Setting Up Blending for Antialiasing Lines: antilines.cpp	180
Example 4.10	Creating a 256 × 256 RGBA Color Renderbuffer	187
Example 4.11	Attaching a Renderbuffer for Rendering	188
Example 4.12	Specifying <code>layout</code> Qualifiers for MRT Rendering	194
Example 4.13	Layout Qualifiers Specifying the Index of Fragment Shader Outputs	198
Example 5.1	Multiplying Multiple Matrices in a Vertex Shader	233
Example 5.2	Simple Use of <code>gl_ClipDistance</code>	238
Example 5.3	Example Initialization of a Transform Feedback Buffer	243
Example 5.4	Example Specification of Transform Feedback Varyings	245
Example 5.5	Leaving Gaps in a Transform Feedback Buffer	247
Example 5.6	Assigning Transform Feedback Outputs to Different Buffers	248
Example 5.7	Assigning Transform Feedback Outputs to Different Buffers	249
Example 5.8	Vertex Shader Used in Geometry Pass of Particle System Simulator	254
Example 5.9	Configuring the Geometry Pass of the Particle System Simulator	254
Example 5.10	Vertex Shader Used in Simulation Pass of Particle System Simulator	255

Example 5.11	Configuring the Simulation Pass of the Particle System Simulator.....	257
Example 5.12	Main Rendering Loop of the Particle System Simulator	257
Example 6.1	Direct Specification of Image Data in C	278
Example 6.2	Loading Static Data into Texture Objects	279
Example 6.3	Loading Data into a Texture Using a Buffer Object	280
Example 6.4	Definition of the vglImageData Structure	283
Example 6.5	Simple Image Loading Example	284
Example 6.6	Loading a Texture Using loadImage	285
Example 6.7	Simple Texture Lookup Example (Fragment Shader)	297
Example 6.8	Simple Texture Lookup Example (Vertex Shader).....	297
Example 6.9	Simple Texturing Example	298
Example 6.10	Setting the Border Color of a Sampler	301
Example 6.11	Texture Swizzle Example.....	302
Example 6.12	Simple Multitexture Example (Vertex Shader).....	304
Example 6.13	Simple Multitexture Example (Fragment Shader).....	305
Example 6.14	Simple Multitexture Example	305
Example 6.15	Simple Volume Texture Vertex Shader	307
Example 6.16	Simple Volume Texture Fragment Shader.....	308
Example 6.17	Initializing a Cube-Map Texture	310
Example 6.18	Initializing a Cube-Map Array Texture.....	311
Example 6.19	Simple Skybox Example—Vertex Shader.....	313
Example 6.20	Simple Skybox Example—Fragment Shader	313
Example 6.21	Cube-Map Environment Mapping Example—Vertex Shader	314
Example 6.22	Cube-Map Environment Mapping Example—Fragment Shader	314
Example 6.23	Creating and Initializing a Buffer Texture	320
Example 6.24	Texel Lookups from a Buffer Texture	321
Example 6.25	Creating a Texture View with a New Format.....	324
Example 6.26	Creating a Texture View with a New Target	325
Example 6.27	Simple Point Sprite Vertex Shader.....	347
Example 6.28	Simple Point Sprite Fragment Shader	347
Example 6.29	Analytic Shape Fragment Shader	348

Example 6.30	Attaching a Texture Level as a Framebuffer Attachment: fbotexture.cpp	353
Example 7.1	Setting Final Color Values with No Lighting.....	363
Example 7.2	Ambient Lighting	364
Example 7.3	Directional Light Source Lighting.....	366
Example 7.4	Point-Light Source Lighting.....	369
Example 7.5	Spotlight Lighting	371
Example 7.6	Point-light Source Lighting in the Vertex Shader.....	374
Example 7.7	Structure for Holding Light Properties	376
Example 7.8	Multiple Mixed Light Sources	377
Example 7.9	Structure to Hold Material Properties	380
Example 7.10	Code Snippets for Using an Array of Material Properties.....	380
Example 7.11	Front and Back Material Properties	382
Example 7.12	Vertex Shader for Hemisphere Lighting	388
Example 7.13	Shaders for Image-based Lighting	394
Example 7.14	Shaders for Spherical Harmonics Lighting.....	398
Example 7.15	Creating a Framebuffer Object with a Depth Attachment	401
Example 7.16	Setting up the Matrices for Shadow Map Generation.....	402
Example 7.17	Simple Shader for Shadow Map Generation.....	403
Example 7.18	Rendering the Scene From the Light's Point of View	404
Example 7.19	Matrix Calculations for Shadow Map Rendering	406
Example 7.20	Vertex Shader for Rendering from Shadow Maps.....	406
Example 7.21	Fragment Shader for Rendering from Shadow Maps...	407
Example 8.1	Vertex Shader for Drawing Stripes	416
Example 8.2	Fragment Shader for Drawing Stripes	417
Example 8.3	Vertex Shader for Drawing Bricks	420
Example 8.4	Fragment Shader for Drawing Bricks	421
Example 8.5	Values for Uniform Variables Used by the Toy Ball Shader	423
Example 8.6	Vertex Shader for Drawing a Toy Ball	424
Example 8.7	Fragment Shader for Drawing a Toy Ball	429
Example 8.8	Fragment Shader for Procedurally Discarding Part of an Object.....	431

Example 8.9	Vertex Shader for Doing Procedural Bump Mapping	438
Example 8.10	Fragment Shader for Procedural Bump Mapping	440
Example 8.11	Fragment Shader for Adaptive Analytic Antialiasing	451
Example 8.12	Source Code for an Antialiased Brick Fragment Shader	456
Example 8.13	Source Code for an Antialiased Checkerboard Fragment Shader	458
Example 8.14	C function to Generate a 3D Noise Texture	469
Example 8.15	A Function for Activating the 3D Noise Texture	471
Example 8.16	Cloud Vertex Shader	473
Example 8.17	Fragment Shader for Cloudy Sky Effect.....	474
Example 8.18	Sun Surface Fragment Shader.....	477
Example 8.19	Fragment Shader for Marble	477
Example 8.20	Granite Fragment Shader	478
Example 8.21	Fragment Shader for Wood.....	480
Example 9.1	Specifying Tessellation Patches	488
Example 9.2	Passing Through Tessellation Control Shader Patch Vertices.....	490
Example 9.3	Tessellation Levels for Quad Domain Tessellation Illustrated in Figure 9.1	492
Example 9.4	Tessellation Levels for an Isoline Domain Tessellation Shown in Figure 9.2	493
Example 9.5	Tessellation Levels for a Triangular Domain Tessellation Shown in Figure 9.3	494
Example 9.6	A Sample Tessellation Evaluation Shader	499
Example 9.7	gl_in Parameters for Tessellation Evaluation Shaders.....	499
Example 9.8	Tessellation Control Shader for Teapot Example.....	501
Example 9.9	The Main Routine of the Teapot Tessellation Evaluation Shader.....	502
Example 9.10	Definition of $B(i, u)$ for the Teapot Tessellation Evaluation Shader.....	503
Example 9.11	Computing Tessellation Levels Based on View-Dependent Parameters.....	504
Example 9.12	Specifying Tessellation Level Factors Using Perimeter Edge Centers	506

Example 9.13	Displacement Mapping in main Routine of the Teapot Tessellation Evaluation Shader	508
Example 10.1	A Simple Pass-Through Geometry Shader	511
Example 10.2	Geometry Shader Layout Qualifiers	512
Example 10.3	Implicit Declaration of <code>gl_in[]</code>	514
Example 10.4	Implicit Declaration of Geometry Shader Outputs	523
Example 10.5	A Geometry Shader that Drops Everything.....	526
Example 10.6	Geometry Shader Passing Only Odd-Numbered Primitives.....	526
Example 10.7	Fur Rendering Geometry Shader.....	528
Example 10.8	Fur Rendering Fragment Shader	529
Example 10.9	Global Layout Qualifiers Used to Specify a Stream Map	533
Example 10.10	Example 10.9 Rewritten to Use Interface Blocks	534
Example 10.11	Incorrect Emission of Vertices into Multiple Streams.....	535
Example 10.12	Corrected Emission of Vertices into Multiple Streams.....	536
Example 10.13	Assigning Transform Feedback Outputs to Buffers	537
Example 10.14	Simple Vertex Shader for Geometry Sorting.....	541
Example 10.15	Geometry Shader for Geometry Sorting	542
Example 10.16	Configuring Transform Feedback for Geometry Sorting	543
Example 10.17	Pass-Through Vertex Shader used for Geometry Shader Sorting	544
Example 10.18	OpenGL Setup Code for Geometry Shader Sorting.....	545
Example 10.19	Rendering Loop for Geometry Shader Sorting.....	547
Example 10.20	Geometry Amplification Using Nested Instancing	550
Example 10.21	Directing Geometry to Different Viewports with a Geometry Shader	552
Example 10.22	Creation of Matrices for Viewport Array Example.....	553
Example 10.23	Specifying Four Viewports.....	554
Example 10.24	Example Code to Create an FBO with an Array Texture Attachment	556
Example 10.25	Geometry Shader for Rendering into an Array Texture	557
Example 11.1	Examples of Image Format Layout Qualifiers	568

Example 11.2	Creating, Allocating, and Binding a Texture to an Image Unit.....	571
Example 11.3	Creating and Binding a Buffer Texture to an Image Unit.....	572
Example 11.4	Simple Shader Demonstrating Loading and Storing into Images.....	574
Example 11.5	Simple Declaration of a Buffer Block.....	576
Example 11.6	Creating a Buffer and Using it for Shader Storage.....	577
Example 11.7	Declaration of Structured Data.....	577
Example 11.8	Naïvely Counting Overdraw in a Scene.....	578
Example 11.9	Counting Overdraw with Atomic Operations.....	581
Example 11.10	Possible Definitions for <code>IMAGE_PARAMS</code>	583
Example 11.11	Equivalent Code for <code>imageAtomicAdd</code>	584
Example 11.12	Equivalent Code for <code>imageAtomicExchange</code> and <code>imageAtomicComp</code>	585
Example 11.13	Simple Per-Pixel Mutex Using <code>imageAtomicCompSwap</code>	585
Example 11.14	Example Use of a Sync Object.....	592
Example 11.15	Basic Spin-Loop Waiting on Memory.....	594
Example 11.16	Result of Loop-Hoisting on Spin-Loop.....	594
Example 11.17	Examples of Using the <code>volatile</code> Keyword.....	595
Example 11.18	Examples of Using the <code>coherent</code> Keyword.....	598
Example 11.19	Example of Using the <code>memoryBarrier()</code> Function... ..	599
Example 11.20	Using the <code>early_fragment_tests</code> Layout Qualifier.....	604
Example 11.21	Counting Red and Green Fragments Using General Atomics.....	605
Example 11.22	Counting Red and Green Fragments Using Atomic Counters.....	606
Example 11.23	Initializing an Atomic Counter Buffer.....	608
Example 11.24	Initializing for Order-Independent Transparency.....	611
Example 11.25	Per-Frame Reset for Order-Independent Transparency.....	613
Example 11.26	Appending Fragments to Linked List for Later Sorting.....	614
Example 11.27	Main Body of Final Order-Independent Sorting Fragment Shader.....	617

Example 11.28	Traversing Linked-Lists in a Fragment Shader	618
Example 11.29	Sorting Fragments into Depth Order for OIT	619
Example 11.30	Blending Sorted Fragments for OIT	619
Example 12.1	Simple Local Workgroup Declaration	626
Example 12.2	Creating, Compiling, and Linking a Compute Shader	627
Example 12.3	Dispatching Compute Workloads	629
Example 12.4	Declaration of Compute Shader Built-in Variables	630
Example 12.5	Operating on Data	631
Example 12.6	Example of Shared Variable Declarations	633
Example 12.7	Particle Simulation Compute Shader	637
Example 12.8	Initializing Buffers for Particle Simulation	638
Example 12.9	Particle Simulation Fragment Shader	640
Example 12.10	Particle Simulation Rendering Loop	641
Example 12.11	Central Difference Edge Detection Compute Shader	643
Example 12.12	Dispatching the Image Processing Compute Shader ...	644
Example B.1	An Example of Creating an OpenGL ES Version 2.0 Rendering Context	661
Example B.2	Creating an HTML5 Canvas Element	662
Example B.3	Creating an HTML5 Canvas Element that Supports WebGL	663
Example B.4	Our WebGL Applications Main HTML Page	664
Example B.5	Our WebGL Shader Loader: InitShaders.js	666
Example B.6	Loading WebGL Shaders Using <code>InitShaders()</code>	667
Example B.7	Initializing Vertex Buffers in WebGL	668
Example B.8	Our demo.js WebGL Application	669
Example H.1	Creating a Debug Context Using WGL	866
Example H.2	Creating a Debug Context Using GLX	867
Example H.3	Prototype for the Debug Message Callback Function	868
Example H.4	Creating Debug Message Filters	873
Example H.5	Sending Application-Generated Debug Messages	875
Example H.6	Using an Elapsed Time Query	882

About This Guide

The OpenGL graphics system is a software interface to graphics hardware. (The 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. This guide explains how to program with the OpenGL graphics system to deliver the visual effect you want.

What This Guide Contains

This guide contains the following chapters:

- Chapter 1, “Introduction to OpenGL”, provides a glimpse into what OpenGL can do. It also presents a simple OpenGL program and explains the essential programming details you need to know for the subsequent chapters.
- Chapter 2, “Shader Fundamentals”, discusses the major feature of OpenGL, programmable shaders, demonstrating how to initialize and use them within an application.
- Chapter 3, “Drawing with OpenGL”, describes the various methods for rendering geometry using OpenGL, as well as some optimization techniques for making rendering more efficient.
- Chapter 4, “Color, Pixels, and Framebuffers”, explains OpenGL’s processing of color, including how pixels are processed, buffers are managed, and rendering techniques focused on pixel processing.
- Chapter 5, “Viewing Transformations, Clipping, and Feedback”, details the operations for presenting a three-dimensional scene on a two-dimensional computer screen, including the mathematics and shader operations for the various types of geometric projection.
- Chapter 6, “Textures”, discusses combining *geometric models* and imagery for creating realistic, high-detailed three-dimensional models.
- Chapter 7, “Light and Shadow”, describes simulating illumination effects for computer graphics, focusing on implementing those techniques in programmable shaders.

-
- Chapter 8, “Procedural Texturing”, details the generation of textures and other surface effects using programmable shaders for increased realism and other rendering effects.
 - Chapter 9, “Tessellation Shaders”, explains OpenGL’s shader facility for managing and tessellating geometric surfaces.
 - Chapter 10, “Geometry Shaders”, describe an additional technique for modifying geometric primitives within the OpenGL rendering pipeline using shaders.
 - Chapter 11, “Memory”, demonstrates techniques using OpenGL’s framebuffer and buffer memories for advanced rendering techniques and nongraphical uses.
 - Chapter 12, “Compute Shaders”, introduces the newest shader stage which integrates general computation into the OpenGL rendering pipeline.

Additionally, a number of appendices are available for reference.

- Appendix A, “Basics of GLUT: The OpenGL Utility Toolkit”, discusses the library that handles window system operations. GLUT is portable and it makes code examples shorter and more comprehensible.
- Appendix B, “OpenGL ES and WebGL”, details the other APIs in the OpenGL family, including OpenGL ES for embedded and mobile systems, and WebGL for interactive 3D applications within Web browsers.
- Appendix C, “Built-in GLSL Variables and Functions”, provides a detailed reference to OpenGL Shading Language.
- Appendix D, “State Variables”, lists the state variables that OpenGL maintains and describes how to obtain their values.
- Appendix E, “Homogeneous Coordinates and Transformation Matrices”, explains some of the mathematics behind *matrix* transformations.
- Appendix F, “OpenGL and Window Systems”, describes the various window–system-specific libraries that provide the binding routines used for allowing OpenGL to render with their native windows.
- Appendix G, “Floating-Point Formats for Textures, Framebuffers, and Renderbuffers”, provides an overview of the floating-point formats used within OpenGL.
- Appendix H, “Debugging and Profiling OpenGL”, discusses the latest debug features available within OpenGL.

-
- Appendix I, “Buffer Object Layouts”, provides a reference for use with uniform buffers using the standard memory layouts defined in OpenGL.

What’s New in This Edition

Virtually everything! For those familiar with previous versions of the *OpenGL Programming Guide*, this edition is a complete rewrite focusing on the latest methods and techniques for OpenGL application development. It combines the function-centric approach of the classic Red Book, with the *shading* techniques found in the *OpenGL Shading Language* (commonly called the “Orange Book”).

In this edition, the author team was expanded to include major contributors to OpenGL’s evolution, as well as the OpenGL Shading Language specification editor. As such, this edition covers the very latest version of OpenGL, Version 4.3, including compute shaders. It also describes every stage of the programmable rendering pipeline. We sincerely hope you find it useful and educational.

What You Should Know Before Reading This Guide

This guide assumes only that you know how to program in the C language (we do use a little bit of C++, but nothing you won’t be able to figure out easily) and that you have some background in mathematics (geometry, trigonometry, linear algebra, calculus, and differential geometry). Even if you have little or no experience with computer graphics technology, you should be able to follow most of the discussions in this book. Of course, computer graphics is an ever-expanding subject, so you may want to enrich your learning experience with supplemental reading:

- *Computer Graphics: Principles and Practice, Third Edition*, by John F. Hughes et al. (Addison-Wesley, forthcoming 2013)—This book is an encyclopedic treatment of the subject of computer graphics. It includes a wealth of information but is probably best read after you have some experience with the subject.
- *3D Computer Graphics* by Andrew S. Glassner (The Lyons Press, 1994)—This book is a nontechnical, gentle introduction to computer graphics. It focuses on the visual effects that can be achieved, rather than on the techniques needed to achieve them.

Another great place for all sorts of general information is the OpenGL Web site. This Web site contains software, sample programs, documentation,

FAQs, discussion boards, and news. It is always a good place to start any search for answers to your OpenGL questions:

<http://www.opengl.org/>

Additionally, full documentation of all the procedures and shading language syntax that compose the latest OpenGL version are documented and available at the official OpenGL Web site. These Web pages replace the *OpenGL Reference Manual* that was published by the OpenGL Architecture Review Board and Addison-Wesley.

OpenGL is really a hardware-independent specification of a programming interface, and you use a particular implementation of it on a particular kind of hardware. This guide explains how to program with any OpenGL implementation. However, since implementations may vary slightly—in performance and in providing additional, optional features, for example—you might want to investigate whether supplementary documentation is available for the particular implementation you’re using. In addition, the provider of your particular implementation might have OpenGL-related utilities, toolkits, programming and debugging support, widgets, sample programs, and demos available at its Web site.

How to Obtain the Sample Code

This guide contains many sample programs to illustrate the use of particular OpenGL programming techniques. As the audience for this guide has a wide range of experience, from novice to seasoned veteran, with both computer graphics and OpenGL, the examples published in these pages usually present the simplest approach to a particular rendering situation, demonstrated using the OpenGL Version 4.3 interface. This is done mainly to make the presentation straightforward and accessible to those readers just starting with OpenGL. For those of you with extensive experience looking for implementations using the latest features of the API, we first thank you for your patience with those following in your footsteps, and ask that you please visit our Web site:

<http://www.opengl-redbook.com/>

There, you will find the source code for all examples in this text, implementations using the latest features, and additional discussion describing the modifications required in moving from one version of OpenGL to another.

All of the programs contained within this book use the OpenGL Utility Toolkit (GLUT), originally authored by Mark Kilgard. For this edition, we

use the open-source version of the GLUT interface from the folks developing the fre glut project. They have enhanced Mark's original work (which is thoroughly documented in his book, *OpenGL Programming for the X Window System*, Addison-Wesley, 1997). You can find their open-source project page at the following address:

<http://fre glut.sourceforge.net/>

You can obtain code and binaries of their implementation at this site.

The section "OpenGL-Related Libraries" in Chapter 1 and Appendix A give more information about using GLUT. Additional resources to help accelerate your learning and programming of OpenGL and GLUT can be found at the OpenGL Web site's resource pages:

<http://www.opengl.org/resources/>

Many implementations of OpenGL might also include the code samples as part of the system. This source code is probably the best source for your implementation, because it might have been optimized for your system. Read your machine-specific OpenGL documentation to see where those code samples can be found.

Errata

Unfortunately, it is likely this book will have errors. Additionally, OpenGL is updated during the publication of this guide: errors are corrected and clarifications are made to the specification, and new specifications are released. We keep a list of bugs and updates at our Web site, <http://www.opengl-redbook.com/>, where we also offer facilities for reporting any new bugs you might find. If you find an error, please accept our apologies, and our thanks in advance for reporting it. We'll get it corrected as soon as possible.

Style Conventions

These style conventions are used in this guide:

- **Bold**—Command and routine names and matrices
- *Italics*—Variables, arguments, parameter names, spatial dimensions, matrix components, and first occurrences of key terms.
- Regular—Enumerated types and defined constants

Code examples are set off from the text in a monospace font, and command summaries are shaded with gray boxes.

In a command summary, we sometimes use braces to identify options among data types. In the following example, **glCommand()** has four possible suffixes: s, i, f, and d, which stand for the data types GLshort, GLint, GLfloat, and GLdouble. In the function prototype for **glCommand()**, *TYPE* is a wildcard that represents the data type indicated by the suffix.

```
void glCommand{sifd}(TYPE x1, TYPE y1, TYPE x2, TYPE y2);
```

We use this form when the number of permutations of the function becomes unruly.

This page intentionally left blank

Drawing with OpenGL



Chapter Objectives

After reading this chapter, you will be able to:

- Identify all of the rendering primitives available in OpenGL.
- Initialize and populate data buffers for use in rendering geometry.
- Optimize rendering using advanced techniques like *instanced rendering*.

The primary use of OpenGL is to render graphics into a framebuffer. To accomplish this, complex objects are broken up into *primitives*—points, lines, and triangles that when drawn at high enough density give the appearance of 2D and 3D objects. OpenGL includes many functions for rendering such primitives. These functions allow you to describe the layout of primitives in memory, how many primitives to render, and what form they take, and even to render many copies of the same set of primitives with one function call. These are arguably the most important functions in OpenGL, as without them, you wouldn't be able to do much but clear the screen.

This chapter contains the following major sections:

- “OpenGL Graphics Primitives” describes the available graphics primitives in OpenGL that you can use in your renderings.
- “Data in OpenGL Buffers” explains the mechanics of working with data in OpenGL.
- “Vertex Specification” outlines how to use vertex data for rendering, and processing it using vertex shaders.
- “OpenGL Drawing Commands” introduces the set of functions that cause OpenGL to draw.
- “Instanced Rendering” describes how to render multiple objects using the same vertex data efficiently.

OpenGL Graphics Primitives

OpenGL includes support for many primitive types. Eventually they all get rendered as one of three types—points, lines, or triangles. Line and triangle types can be combined together to form strips, loops (for lines), and fans (for triangles). Points, lines, and triangles are the *native* primitive types supported by most graphics hardware.¹ Other primitive types are supported by OpenGL, including patches, which are used as inputs to the tessellator and the *adjacency primitives* that are designed to be used as inputs to the geometry shader. Tessellation (and tessellation shaders) are introduced in Chapter 9, and geometry shaders are introduced in Chapter 10. The patch and adjacency primitive types will be covered in detail in each of those chapters. In this section, we cover only the point, line, and triangle primitive types.

1. In terms of hardware support, this means that the graphics processor likely includes direct hardware support for rasterizing these types of primitives. Other primitive types such as patches and adjacency primitives are never directly rasterized.

Points

Points are represented by a single vertex. The vertex represents a point in four-dimensional *homogeneous coordinates*. As such, a point really has no area, and so in OpenGL it is really an analogue for a square region of the display (or draw buffer). When rendering points, OpenGL determines which pixels are covered by the point using a set of rules called *rasterization rules*. The rules for rasterizing a point in OpenGL are quite straightforward—a sample is considered covered by a point if it falls within a square centered on the point's location in window coordinates. The side length of the square is equal to the point's size, which is fixed state (set with `glPointSize()`), or the value written to the `gl_PointSize` built-in variable in the vertex, tessellation, or geometry shader. The value written to `gl_PointSize` in the shader is used only if `GL_PROGRAM_POINT_SIZE` is enabled, otherwise it is ignored and the fixed state value set with `glPointSize()` is used.

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

Sets the fixed size, in pixels, that will be used for points when `GL_PROGRAM_POINT_SIZE` is not enabled.

The default point size is 1.0. Thus, when points are rendered, each vertex essentially becomes a single pixel on the screen (unless it's clipped, of course). If the point size is increased (either with `glPointSize()`, or by writing a value larger than 1.0 to `gl_PointSize`), then each point vertex may end up lighting more than one pixel. For example, if the point size is 1.2 pixels and the point's vertex lies exactly at a pixel center, then only that pixel will be lit. However, if the point's vertex lies exactly midway between two horizontally or vertically adjacent pixel centers, then both of those pixels will be lit (i.e., two pixels will be lit). If the point's vertex lies at the exact midpoint between four adjacent pixels, then all four pixels will be lit—for a total of four pixels being lit for one point!

Point Sprites

When you render points with OpenGL, the fragment shader is run for every fragment in the point. Each point is essentially a square area of the screen and each pixel can be shaded a different color. You can calculate that color analytically in the fragment shader or use a texture to shade the point. To assist in this, OpenGL fragment shaders include a special *built-in variable* called `gl_PointCoord` which contains the coordinate within the point where the current fragment is located. `gl_PointCoord` is available

only in the fragment shader (it doesn't make much sense to include it in other shaders) and has a defined value only when rendering points. By simply using `gl_PointCoord` as a source for texture coordinates, bitmaps and textures can be used instead of a simple square block. Combined with alpha blending or with discarding fragments (using the `discard` keyword), it's even possible to create point *sprites* with odd shapes.

We'll revisit point sprites with an example shortly. If you want to skip ahead, the example is shown in "Point Sprites" on Page 346.

Lines, Strips, and Loops

In OpenGL, the term *line* refers to a *line segment*, not the mathematician's version that extends to infinity in both directions. Individual lines are therefore represented by pairs of vertices, one for each endpoint of the line. Lines can also be joined together to represent a connected series of line segments, and optionally closed. The closed sequence is known as a *line loop*, whereas the open sequence (one that is not closed) is known as a *line strip*. As with points, lines technically have no area, and so special rasterization rules are used to determine which pixels should be lit when a line segment is rasterized. The rule for line rasterization is known as the *diamond exit rule*. It is covered in some detail in the OpenGL specification. However, we attempt to paraphrase it here. When rasterizing a line running from point A to point B, a pixel should be lit if the line passes through the imaginary edge of a diamond shape drawn inside the pixel's square area on the screen—unless that diamond contains point B (i.e., the end of the line is inside the diamond). That way, if another, second line is drawn from point B to point C, the pixel in which B resides is lit only once.

The diamond exit rule suffices for thin lines, but OpenGL allows you to specify wider sizes for lines using the `glLineWidth()` function (the equivalent for `glPointSize()` for lines).

```
void glLineWidth(GLfloat width);
```

Sets the fixed width of lines. The default value is 1.0. *width* is the new value of line width and must be greater than 0.0, otherwise an error is generated.

There is no equivalent to `glPointSize` for lines—lines are rendered at one fixed width until state is changed in OpenGL. When the line width is greater than 1, the line is simply replicated *width* times either horizontally or vertically. If the line is *y-major* (i.e., it extends further vertically than

horizontally), it is replicated horizontally. If it is *x-major* then it is replicated vertically.

The OpenGL specification is somewhat liberal on how ends of lines are represented and how wide lines are rasterized when antialiasing is turned off. When antialiasing is turned on, lines are treated as rectangles aligned along the line, with width equal to the current line width.

Triangles, Strips, and Fans

Triangles are made up of collections of three vertices. When separate triangles are rendered, each triangle is independent of all others. A triangle is rendered by projecting each of the three vertices into screen space and forming three edges running between the edges. A sample is considered covered if it lies on the positive side of all of the *half spaces* formed by the lines between the vertices. If two triangles share an edge (and therefore a pair of vertices), no single sample can be considered inside both triangles. This is important because, although some variation in rasterization algorithm is allowed by the OpenGL specification, the rules governing pixels that lie along a shared edge are quite strict:

- No pixel on a shared edge between two triangles that together would cover the pixel should be left unlit.
- No pixel on a shared edge between two triangles should be lit by more than one of them.

This means that OpenGL will reliably rasterize meshes with shared edges without gaps between the triangles, and without *overdraw*.² This is important when rasterizing triangle *strips* or *fans*. When a triangle strip is rendered, the first three vertices form the first triangle, then each subsequent vertex forms another triangle along with the last two vertices of the previous triangle. This is illustrated in Figure 3.1.

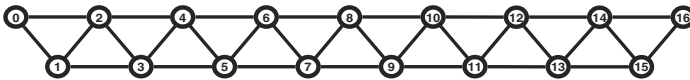


Figure 3.1 Vertex layout for a triangle strip

When rendering a triangle fan, the first vertex forms a shared point that is included in each subsequent triangle. Triangles are then formed using that

2. Overdraw is where the same pixel is lit more than once, and can cause artifacts when blending is enabled, for example.

shared point and the next two vertices. An arbitrarily complex *convex* polygon can be rendered as a triangle fan. Figure 3.2 shows the vertex layout of a triangle fan.

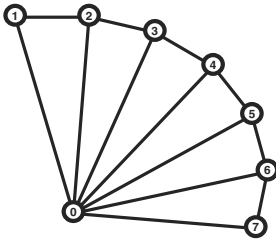


Figure 3.2 Vertex layout for a triangle fan

These primitive types are used by the drawing functions that will be introduced in the next section. They are represented by OpenGL tokens that are passed as arguments to functions used for rendering. Table 3.1 shows the mapping of primitive types to the OpenGL tokens used to represent them.

Table 3.1 OpenGL Primitive Mode Tokens

Primitive Type	OpenGL Token
Points	GL_POINTS
Lines	GL_LINES
Line Strips	GL_LINE_STRIP
Line Loops	GL_LINE_LOOP
Independent Triangles	GL_TRIANGLES
Triangle Strips	GL_TRIANGLE_STRIP
Triangle Fans	GL_TRIANGLE_FAN

Rendering Polygons As Points, Outlines, or Solids

A *polygon* has two sides—front and back—and might be rendered differently depending on which side is facing the viewer. This allows you to have cutaway views of solid objects in which there is an obvious distinction between the parts that are inside and those that are outside. By default, both front and back *faces* are drawn in the same way. To change this, or to draw only outlines or vertices, use `glPolygonMode()`.

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

Controls the drawing mode for a polygon's front and back faces. The parameter *face* must be `GL_FRONT_AND_BACK`; while *mode* can be `GL_POINT`, `GL_LINE`, `GL_FILL` to indicate whether the polygon should be drawn as points, outlined, or filled. By default, both the front and back faces are drawn filled.

Reversing and Culling Polygon Faces

By convention, polygons whose vertices appear in counterclockwise order on the screen are called *front facing*. You can construct the surface of any “reasonable” solid—a mathematician would call such a surface an orientable manifold (spheres, donuts, and teapots are orientable; Klein bottles and Möbius strips aren't)—from polygons of consistent orientation. In other words, you can use all clockwise polygons or all counterclockwise polygons.

Suppose you've consistently described a model of an orientable surface but happen to have the clockwise orientation on the outside. You can swap what OpenGL considers the back face by using the function `glFrontFace()`, supplying the desired orientation for front-facing polygons.

```
void glFrontFace(GLenum mode);
```

Controls how front-facing polygons are determined. By default, *mode* is `GL_CCW`, which corresponds to a counterclockwise orientation of the ordered vertices of a projected polygon in window coordinates. If *mode* is `GL_CW`, faces with a clockwise orientation are considered front-facing.

Note: The orientation (clockwise or counterclockwise) of the vertices is also known as its winding.

In a completely enclosed surface constructed from opaque polygons with a consistent orientation, none of the back-facing polygons are ever visible—they're always obscured by the front-facing polygons. If you are outside this surface, you might enable culling to discard polygons that OpenGL determines are back-facing. Similarly, if you are inside the object, only back-facing polygons are visible. To instruct OpenGL to discard front- or back-facing polygons, use the command `glCullFace()` and enable culling with `glEnable()`.

```
void glCullFace(GLenum mode);
```

Indicates which polygons should be discarded (culled) before they're converted to screen coordinates. The mode is either `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK` to indicate front-facing, back-facing, or all polygons. To take effect, culling must be enabled using `glEnable()` with `GL_CULL_FACE`; it can be disabled with `glDisable()` and the same argument.

Advanced

In more technical terms, deciding whether a face of a polygon is front- or back-facing depends on the sign of the polygon's area computed in window coordinates. One way to compute this area is

$$a = \frac{1}{2} \sum_{i=0}^{n-1} x_i y_{i \oplus 1} - x_{i \oplus 1} y_i$$

where x_i and y_i are the x and y window coordinates of the i^{th} vertex of the n -vertex polygon and where $i \oplus 1$ is shorthand for $(i + 1) \bmod n$, where \bmod is the modulus operator.

Assuming that `GL_CCW` has been specified, if $a > 0$, the polygon corresponding to that vertex is considered to be front-facing; otherwise, it's back-facing. If `GL_CW` is specified and if $a < 0$, then the corresponding polygon is front-facing; otherwise, it's back-facing.

Data in OpenGL Buffers

Almost everything you will ever do with OpenGL will involve buffers full of data. Buffers in OpenGL are represented as *buffer objects*. You've already had a brief introduction to buffer objects in Chapter 1. However, in this section we'll dig a little deeper into the specifics of how buffer objects are used; ways to create, manage, and destroy them; and the best practices associated with buffer objects.

Creating and Allocating Buffers

As with many things in OpenGL, buffer objects are named using `GLuint` values. Values are reserved using the `glGenBuffers()` command. This function has already been described in Chapter 1, but we include the prototype here again for handy reference.

```
void glGenBuffers(GLsizei n, GLuint *buffers);
```

Returns n currently unused names for buffer objects in the array *buffers*.

After calling `glGenBuffers()`, you will have an array of buffer object names in *buffers*, but at this time, they're just placeholders. They're not actually buffer objects—yet. The buffer objects themselves are not actually created until the name is first bound to one of the buffer binding points on the context. This is important because OpenGL may make decisions about the best way to allocate memory for the buffer object based on where it is bound. The buffer binding points (called *targets*) are described in Table 3.2.

Table 3.2 Buffer Binding Targets

Target	Uses
GL_ARRAY_BUFFER	This is the binding point that is used to set vertex array data pointers using <code>glVertexAttribPointer()</code> . This is the target that you will likely use most often.
GL_COPY_READ_BUFFER and GL_COPY_WRITE_BUFFER	Together, these targets form a pair of binding points that can be used to copy data between buffers without disturbing OpenGL state, or implying usage of any particular kind to OpenGL.
GL_DRAW_INDIRECT_BUFFER	A buffer target used to store the parameters for drawing commands when using <i>indirect drawing</i> , which will be explained in detail in the next section.
GL_ELEMENT_ARRAY_BUFFER	Buffers bound to this target can contain vertex indices which are used by <i>indexed draw commands</i> such as <code>glDrawElements()</code> .
GL_PIXEL_PACK_BUFFER	The pixel pack buffer is used as the destination for OpenGL commands that read data from image objects such as textures or the framebuffer. Examples of such commands include <code>glGetTexImage()</code> and <code>glReadPixels()</code> .

Table 3.2 (continued) Buffer Binding Targets

Target	Uses
GL_PIXEL_UNPACK_BUFFER	The pixel unpack buffer is the <i>opposite</i> of the pixel pack buffer—it is used as the <i>source</i> of data for commands like <code>glTexImage2D()</code> .
GL_TEXTURE_BUFFER	Texture buffers are buffers that are bound to texture objects so that their data can be directly read inside shaders. The <code>GL_TEXTURE_BUFFER</code> binding point provides a target for manipulating these buffers, although they must still be attached to textures to make them accessible to shaders.
GL_TRANSFORM_FEEDBACK_BUFFER	Transform feedback is a facility in OpenGL whereby transformed vertices can be captured as they exit the vertex processing part of the pipeline (after the vertex or geometry shader, if present) and some of their attributes written into buffer objects. This target provides a binding point for buffers that are used to record those attributes. Transform feedback will be covered in some detail in “Transform Feedback” on Page 239.
GL_UNIFORM_BUFFER	This target provides a binding point where buffers that will be used as <i>uniform buffer objects</i> may be bound. Uniform buffers are covered in Subsection 2, “Uniform Blocks”.

A buffer object actually is created by binding one of the names reserved by a call to `glGenBuffers()` to one of the targets in Table 3.2 using `glBindBuffer()`. As with `glGenBuffers()`, `glBindBuffer()` was introduced in Chapter 1, but we include its prototype here again for completeness.

```
void glBindBuffer(GLenum target, GLuint buffer);
```

Binds the buffer object named *buffer* to the buffer-binding point as specified by *target*. *target* must be one of the OpenGL buffer-binding targets, and *buffer* must be a name reserved by a call to `glGenBuffers()`. If this the first time the name *buffer* has been bound, a buffer object is created with that name.

Right, so we now have a buffer object bound to one of the targets listed in Table 3.2, now what? The default state of a newly created buffer object is a buffer with no data in it. Before it can be used productively, we must put some data into it.

Getting Data into and out of Buffers

There are many ways to get data into and out of buffers in OpenGL. These range from explicitly providing the data, to replacing parts of the data in a buffer object with new data, to generating the data with OpenGL and recording it into the buffer object. The simplest way to get data into a buffer object is to load data into the buffer at time of allocation. This is accomplished through the use of the `glBufferData()` function. Here's the prototype of `glBufferData()` again.

```
void glBufferData(GLenum target, GLsizeiptr size,  
                 const GLvoid *data, GLenum usage);
```

Allocates *size* bytes of storage for the buffer object bound to *target*. If *data* is non-NULL, that space is initialized with the contents of memory addressed by *data*. *usage* is provided to allow the application to supply OpenGL with a hint as to the intended usage for the data in the buffer.

It's important to note that `glBufferData()` actually allocates (or reallocates) storage for the buffer object. That is, if the size of the new data is greater than the current storage space allocated for the buffer object, the buffer object will be resized to make room. Likewise, if the new data is smaller than what has been allocated for the buffer, the buffer object will shrink to match the new size. The fact that it is possible to specify the initial data to be placed into the buffer object is merely a convenience and is not necessarily the best way to do it (or even the most convenient, for that matter).

The *target* of the initial binding is not the only information OpenGL uses to decide how to best allocate the buffer object's data store. The other important parameter to `glBufferData()` is the *usage* parameter. *usage* must be one of the standard usage tokens such as `GL_STATIC_DRAW` or `GL_DYNAMIC_COPY`. Notice how the token name is made of two parts—the first being one of `STATIC`, `DYNAMIC`, or `STREAM` and the second being one of `DRAW`, `READ`, or `COPY`.

The meanings of these “subtokens” are shown in Table 3.3.

Table 3.3 Buffer Usage Tokens

Token Fragment	Meaning
<code>_STATIC_</code>	The data store contents will be modified once and used many times.
<code>_DYNAMIC_</code>	The data store contents will be modified repeatedly and used many times.
<code>_STREAM_</code>	The data store contents will be modified once and used at most a few times.
<code>_DRAW</code>	The data store contents are modified by the application and used as the source for OpenGL drawing and image specification commands.
<code>_READ</code>	The data store contents are modified by reading data from OpenGL and used to return that data when queried by the application.
<code>_COPY</code>	The data store contents are modified by reading data from OpenGL and used as the source for OpenGL drawing and image specification commands.

Accurate specification of the *usage* parameter is important to achieve optimal performance. This parameter conveys useful information to OpenGL about how you plan to use the buffer. Consider the first part of the accepted tokens first. When the token starts with `_STATIC_`, this indicates that the data will change very rarely, if at all—it is essentially static. This should be used for data that will be specified once and never modified again. When *usage* includes `_STATIC_`, OpenGL may decide to shuffle the data around internally in order to make it fit in memory better, or be a more optimal data format. This may be an expensive operation, but since the data is static, it needs to be performed only once and so the payoff may be great.

Including `_DYNAMIC_` in *usage* indicates that you're going to change the data from time to time but will probably use it many times between modifications. You might use this, for example, in a modeling program where the data is essentially static—until the user edits it. In this case, it'll probably be used for many frames, then be modified, and then used for many more frames, and so on. This is in contrast to the `GL_STREAM_` subtoken. This indicates that you're planning on regularly modifying the data in the buffer and using it only a few times (maybe only once) between each modification. In this case, OpenGL might not even copy your data to fast graphics memory if it can access it in place. This should be used for applications such as physical simulations running on the CPU where a new set of data is presented in each frame.

Now turn your attention to the second part of the *usage* tokens. This part of the token indicates *who* is responsible for updating and using the data. When the token includes `_DRAW`, this infers that the buffer will be used as a source of data during regular OpenGL drawing operations. It will be *read* a lot, compared to data whose *usage* token includes `_READ`, which is likely to be *written* often. Including `_READ` indicates that the application will read back from the buffer (see “Accessing the Content of Buffers”), which in turn infers that the data is likely to be written to often by OpenGL. *usage* parameters including `_DRAW` should be used for buffers containing vertex data, for example, whereas parameters including `_READ` should be used for pixel buffer objects and other buffers that will be used to retrieve information from OpenGL. Finally, including `_COPY` in *usage* indicates that the application will use OpenGL to generate data to be placed in the buffer, which will then be used as a source for subsequent drawing operations. An example of an appropriate use of `_COPY` is transform feedback buffers—buffers that will be written by OpenGL and then be used as vertex buffers in later drawing commands.

Initializing Part of a Buffer

Suppose you have an array containing some vertex data, another containing some color information, and yet another containing texture coordinates or some other data. You’d like to *pack* the data back to back into one big buffer object so that OpenGL can use it. The arrays may or may not be contiguous in memory, so you can’t use `glBufferData()` to upload all of it in one go. Further, if you use `glBufferData()` to upload, say, the vertex data first, then the buffer will be sized to exactly match the vertex data and there won’t be room for the color or texture coordinate information. That’s where `glBufferSubData()` comes in.

```
void glBufferSubData(GLenum target, GLintptr offset,  
                    GLsizei size, const GLvoid *data);
```

Replaces a subset of a buffer object’s data store with new data. The section of the buffer object bound to *target* starting at *offset* bytes is updated with the *size* bytes of data addressed by *data*. An error is thrown if *offset* and *size* together specify a range that is beyond the bounds of the buffer object’s data store.

By using a combination of `glBufferData()` and `glBufferSubData()`, we can allocate and initialize a buffer object and upload data into several separate sections of it. An example is shown in Example 3.1.

Example 3.1 Initializing a Buffer Object with `glBufferSubData()`

```
// Vertex positions
static const GLfloat positions[] =
{
    -1.0f, -1.0f, 0.0f, 1.0f,
     1.0f, -1.0f, 0.0f, 1.0f,
     1.0f,  1.0f, 0.0f, 1.0f,
    -1.0f,  1.0f, 0.0f, 1.0f
};

// Vertex colors
static const GLfloat colors[] =
{
    1.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 1.0f,
    1.0f, 1.0f, 1.0f,
};

// The buffer object
GLuint buffer;

// Reserve a name for the buffer object.
glGenBuffers(1, &buffer);
// Bind it to the GL_ARRAY_BUFFER target.
glBindBuffer(GL_ARRAY_BUFFER, buffer);
// Allocate space for it (sizeof(positions) + sizeof(colors)).
glBufferData(GL_ARRAY_BUFFER,           // target
             sizeof(positions) + sizeof(colors), // total size
             NULL,                       // no data
             GL_STATIC_DRAW);           // usage
// Put "positions" at offset zero in the buffer.
glBufferSubData(GL_ARRAY_BUFFER,       // target
               0,                      // offset
               sizeof(positions),      // size
               positions);             // data
// Put "colors" at an offset in the buffer equal to the filled size of
// the buffer so far - i.e., sizeof(positions).
glBufferSubData(GL_ARRAY_BUFFER,       // target
               sizeof(positions),      // offset
               sizeof(colors),         // size
               colors);                // data
// Now "positions" is at offset 0 and "colors" is directly after it
// in the same buffer.
```

If you simply wish to clear a buffer object's data store to a known value, you can use the `glClearBufferData()` or `glClearBufferSubData()` functions. Their prototypes are as follows:

```
void glClearBufferData(GLenum target, GLenum internalformat,
                      GLenum format, GLenum type,
                      const void * data);
void glClearBufferSubData(GLenum target,
                          GLenum internalformat,
                          GLintptr offset, GLintptr size,
                          GLenum format, GLenum type,
                          const void * data);
```

Clear all or part of a buffer object's data store. The data store of the buffer bound to *target* is filled with the data stored in *data*. *format* and *type* specify the format and type of the data pointed to by *data*, respectively. The data is first converted into the format specified by *internalformat*, and then that data is used to fill the specified range of the buffer's data store. In the case of `glClearBufferData()`, the entire store is filled with the specified data. For `glClearBufferSubData()`, the range is specified by *offset* and *size*, which give the starting offset and size, in bytes of the range, respectively.

Using `glClearBufferData()` or `glClearBufferSubData()` allows you to initialize the data store of a buffer object without necessarily reserving and clearing a region of system memory to do it.

Data can also be copied between buffer objects using the `glCopyBufferSubData()` function. Rather than assembling chunks of data in one large buffer object using `glBufferSubData()`, it is possible to upload the data into separate buffers using `glBufferData()` and then copy from those buffers into the larger buffer using `glCopyBufferSubData()`. Depending on the OpenGL implementation, it may be able to overlap these copies because each time you call `glBufferData()` on a buffer object, it invalidates whatever contents may have been there before. Therefore, OpenGL can sometimes just allocate a whole new data store for your data, even though a copy operation from the previous store has not completed yet. It will then release the old storage at a later opportunity.

The prototype of `glCopyBufferSubData()` is as follows:

```
void glCopyBufferSubData(GLenum readtarget,
                        GLenum writetarget,
                        GLintptr readoffset,
                        GLintptr writeoffset, GLsizei size);
```

Copies part of the data store of the buffer object bound to *readtarget* into the data store of the buffer object bound to *writetarget*. The *size* bytes of data at *readoffset* within *readtarget* are copied into *writetarget* at *writeoffset*. If *readoffset* or *writeoffset* together with *size* would cause either OpenGL to access any area outside the bound buffer objects, a `GL_INVALID_VALUE` error is generated.

Whilst `glCopyBufferSubData()` can be used to copy data between buffers bound to any two targets, the targets `GL_COPY_READ_BUFFER` and `GL_COPY_WRITE_BUFFER` are provided specifically for this purpose. Neither target is used for anything else by OpenGL, and so you can safely bind buffers to them for the purposes of copying or staging data without disturbing OpenGL state or needing to keep track of what was bound to the target before your copy.

Reading the Contents of a Buffer

Data can be read back from a buffer object in a couple of different ways. The first is to use the `glGetBufferSubData()` function. This function reads data from the buffer object bound to one of the targets and places it into a chunk of memory owned by your applications. The prototype of `glGetBufferSubData()` is as follows:

```
void glGetBufferSubData(GLenum target, GLintptr offset,
                       GLsizeiptr size, GLvoid * data);
```

Returns some or all of the data from the buffer object currently bound to *target*. Data starting at byte-offset *offset* and extending for *size* bytes is copied from the data store to the memory pointed to by *data*. An error is thrown if the buffer object is currently mapped, or if *offset* and *size* together define a range beyond the bounds of the buffer object's data store.

`glGetBufferSubData()` is useful when you have generated data using OpenGL and wish to retrieve it. Examples include using transform feedback to process vertices using a GPU, or reading framebuffer or texture data into a Pixel Buffer Object. Both of these topics will be covered later. Of course, it's also possible to use `glGetBufferSubData()` to simply read back data that you previously put into the buffer object.

Accessing the Content of Buffers

The issue with all of the functions covered in this section so far (`glBufferData()`, `glBufferSubData()`, `glCopyBufferSubData()`, and

`glGetBufferSubData()` is that they all cause OpenGL to make a copy of your data. `glBufferData()` and `glBufferSubData()` both copy data from your application's memory into memory owned by OpenGL. Obviously, `glCopyBufferSubData()` causes a copy of previously buffered data to be made. `glGetBufferSubData()` copies data from memory owned by OpenGL into memory provided by your application. Depending on the hardware configuration, it's very possible that the memory owned by OpenGL would be accessible to your application if only you had a pointer to it. Well, you can get that pointer using `glMapBuffer()`.

```
void * glMapBuffer(GLenum target, GLenum access);
```

Maps to the client's address space the entire data store of the buffer object currently bound to *target*. The data can then be directly read or written relative to the returned pointer, depending on the specified *access* policy. If OpenGL is unable to map the buffer object's data store, `glMapBuffer()` generates an error and returns NULL. This may occur for system-specific reasons, such as low virtual memory availability.

When you call `glMapBuffer()`, the function returns a pointer to memory that represents the data store of the buffer object attached to *target*. Note that this memory represents only this buffer—it is not necessarily the memory that the graphics processor will use. The *access* parameter specifies how the application intends to use the memory once it is mapped. It must be one of the tokens shown in Table 3.4.

Table 3.4 Access Modes for `glMapBuffer()`

Token	Meaning
GL_READ_ONLY	The application will only read from the memory mapped by OpenGL.
GL_WRITE_ONLY	The application will only write to the memory mapped by OpenGL.
GL_READ_WRITE	The application may read from or write to the memory mapped by OpenGL.

If `glMapBuffer()` fails to map the buffer object's data store, it returns NULL. The *access* parameter forms a contract between you and OpenGL that specifies how you will access the memory. If you violate that contract,

bad things will happen, which may include ignoring writes to the buffer, corrupting your data or even crashing your program.³

Note: When you map a buffer whose data store is in memory that will not be accessible to your application, OpenGL may need to move the data around so that when you use the pointer it gives you, you get what you expect. Likewise, when you're done with the data and have modified it, OpenGL may need to move it back to a place where the graphics processor can see it. This can be expensive in terms of performance, so great care should be taken when doing this.

When the buffer is mapped with the `GL_READ_ONLY` or `GL_READ_WRITE` *access* mode, the data that was in the buffer object becomes visible to your application. You can read it back, write it to a file, and even modify it in place (so long as you used `GL_READ_WRITE` as the *access* mode). If *access* is `GL_READ_WRITE` or `GL_WRITE_ONLY`, you can write data into memory using the pointer OpenGL gave you. Once you are done using the data or writing data into the buffer object, you must unmap it using `glUnmapBuffer()`, whose prototype is as follows:

```
GLboolean glUnmapBuffer(GLenum target);
```

Releases the mapping created by `glMapBuffer()`. `glUnmapBuffer()` returns `GL_TRUE` unless the data store contents have become corrupt during the time the data store was mapped. This can occur for system-specific reasons that affect the availability of graphics memory, such as screen mode changes. In such situations, `GL_FALSE` is returned and the data store contents are undefined. An application must detect this rare condition and reinitialize the data store.

When you unmap the buffer, any data you wrote into the memory given to you by OpenGL becomes visible in the buffer object. This means that you can place data into buffer objects by allocating space for them using `glBufferData()` and passing `NULL` as the *data* parameter, mapping them, writing data into them directly, and then unmapping them again. Example 3.2 contains an example of loading the contents of a file into a buffer object.

3. The unfortunate thing is that so many applications *do* violate this contract that most OpenGL implementations will assume you don't know what you're doing and will treat all calls to `glMapBuffer()` as if you specified `GL_READ_WRITE` as the *access* parameter, just so these other applications will work.

Example 3.2 Initializing a Buffer Object with `glMapBuffer()`

```
GLuint buffer;
FILE * f;
size_t filesize;

// Open a file and find its size
f = fopen("data.dat", "rb");
fseek(f, 0, SEEK_END);
filesize = ftell(f);
fseek(f, 0, SEEK_SET);

// Create a buffer by generating a name and binding it to a buffer
// binding point - GL_COPY_WRITE_BUFFER here (because the binding means
// nothing in this example).
glGenBuffers(1, &buffer);
glBindBuffer(GL_COPY_WRITE_BUFFER, buffer);

// Allocate the data store for the buffer by passing NULL for the
// data parameter.
glBufferData(GL_COPY_WRITE_BUFFER, (GLsizei)filesize, NULL,
            GL_STATIC_DRAW);
// Map the buffer...
void * data = glMapBuffer(GL_COPY_WRITE_BUFFER, GL_WRITE_ONLY);

// Read the file into the buffer.
fread(data, 1, filesize, f);

// Okay, done, unmap the buffer and close the file.
glUnmapBuffer(GL_COPY_WRITE_BUFFER);
fclose(f);
```

In Example 3.2, the entire contents of a file are read into a buffer object in a single operation. The buffer object is created and allocated to the same size as the file. Once the buffer is mapped, the file can be read directly into the buffer object's data store. No copies are made by the application, and, if the data store is visible to both the application and the graphics processor, no copies will be made by OpenGL.

There may be significant performance advantages to initializing buffer objects in this manner. The logic is this; when you call `glBufferData()` or `glBufferSubData()`, once those functions return, you are free to do whatever you want with the memory you gave them—free it, use it for something else—it doesn't matter. This means that those functions *must* be done with that memory by the time they return, and so they need to make a copy of your data. However, when you call `glMapBuffer()`, the pointer you get points at memory owned by OpenGL. When you call `glUnmapBuffer()`, OpenGL still owns that memory—it's the application that has to be done with it. This means that if the data needs to be moved

or copied, OpenGL can start that process when you call `glUnmapBuffer()` and return immediately, content in the knowledge that it can finish the operation at its leisure without your application interfering in any way. Thus the copy that OpenGL needs to perform can overlap whatever your application does next (making more buffers, reading more files, and so on). If it doesn't need to make a copy, then great! The unmap operation essentially becomes free in that case.

Asynchronous and Explicit Mapping

To address many of the issues involved with mapping buffers using `glMapBuffer()` (such as applications incorrectly specifying the *access* parameter or always using `GL_READ_WRITE`), `glMapBufferRange()` uses flags to specify *access* more precisely. The prototype for `glMapBufferRange()` is as follows:

```
void * glMapBufferRange(GLenum target, GLintptr offset,
                       GLsizeiptr length, GLbitfield access);
```

Maps all or part of a buffer object's data store into the application's address space. *target* specifies the target to which the buffer object is currently bound. *offset* and *length* together indicate the range of the data (in bytes) that is to be mapped. *access* is a bitfield containing flags that describe the mapping.

For `glMapBufferRange()`, *access* is a bitfield that must contain one or both of the `GL_MAP_READ_BIT` and the `GL_MAP_WRITE_BIT` indicating whether the application plans to read from the mapped data store, write to it, or do both. In addition, *access* may contain one or more of the flags shown in Table 3.5.

Table 3.5 Flags for Use with `glMapBufferRange()`

Flag	Meaning
<code>GL_MAP_INVALIDATE_RANGE_BIT</code>	If specified, any data in the specified range of the buffer may be discarded and considered invalid. Any data within the specified range that is not subsequently written by the application becomes undefined. This flag may not be used with <code>GL_MAP_READ_BIT</code> .

Table 3.5 (continued) Flags for Use with `glMapBufferRange()`

Flag	Meaning
<code>GL_MAP_INVALIDATE_BUFFER_BIT</code>	If specified, the <i>entire contents</i> of the buffer may be discarded and considered invalid, regardless of the specified range. Any data lying outside the mapped range of the buffer object becomes undefined, as does any data within the range but not subsequently written by the application. This flag may not be used with <code>GL_MAP_READ_BIT</code> .
<code>GL_MAP_FLUSH_EXPLICIT_BIT</code>	The application will take responsibility to signal to OpenGL which parts of the mapped range contain valid data by calling <code>glFlushMappedBufferRange()</code> prior to calling <code>glUnmapBuffer()</code> . Use this flag if a larger range of the buffer will be mapped and not all of it will be written by the application. This bit must be used in conjunction with <code>GL_MAP_WRITE_BIT</code> . If <code>GL_MAP_FLUSH_EXPLICIT_BIT</code> is not specified, <code>glUnmapBuffer()</code> will automatically flush the entirety of the mapped range.
<code>GL_MAP_UNSYNCHRONIZED_BIT</code>	If this bit is not specified, OpenGL will wait until all pending operations that may access the buffer have completed before returning the mapped range. If this flag is set, OpenGL will not attempt to synchronize operations on the buffer.

As you can see from the flags listed in Table 3.5, the command provides a significant level of control over how OpenGL uses the data in the buffer and how it synchronizes operations that may access that data.

When you specify that you want to invalidate the data in the buffer object by specifying either the `GL_MAP_INVALIDATE_RANGE_BIT` or `GL_MAP_INVALIDATE_BUFFER_BIT`, this indicates to OpenGL that it is free to dispose of any previously stored data in the buffer object. Either of the flags can be set only if you also specify that you're going to write to the buffer by also setting the `GL_MAP_WRITE_BIT` flag. If you specify `GL_MAP_INVALIDATE_RANGE_BIT`, it indicates that you will update the entire range (or at least all the parts of it that you care about). If you set the `GL_MAP_INVALIDATE_BUFFER_BIT`, it means that you don't care what

ends up in the parts of the buffer that you didn't map. Either way, setting the flags indicates that you're planning to update the rest of the buffer with subsequent maps.⁴ When OpenGL is allowed to throw away the rest of the buffer's data, it doesn't have to make any effort to merge your modified data back into the rest of the original buffer. It's probably a good idea to use `GL_MAP_INVALIDATE_BUFFER_BIT` for the first section of the buffer that you map, and then `GL_MAP_INVALIDATE_RANGE_BIT` for the rest of the buffer.

The `GL_MAP_UNSYNCHRONIZED_BIT` flag is used to disengage OpenGL's automatic synchronization between data transfer and use. Without this bit, OpenGL will finish up any in-flight commands that might be using the buffer object. This can *stall* the OpenGL pipeline, causing a bubble and a loss of performance. If you can guarantee that all pending commands will be complete before you actually modify the contents of the buffer (but not necessarily before you call `glMapBufferRange()`) through a method such as calling `glFinish()` or using a *sync object* (which are described in "Atomic Operations and Synchronization" on Page 578 in Chapter 11), then OpenGL doesn't need to do this synchronization for you.

Finally, the `GL_MAP_FLUSH_EXPLICIT_BIT` flag indicates that the application will take on the responsibility of letting OpenGL know which parts of the buffer it has modified before calling `glUnmapBuffer()`. It does this through a call to `glFlushMappedBufferRange()`, whose prototype is as follows:

```
void glFlushMappedBufferRange(GLenum target, GLintptr offset,  
                             GLsizeiptr length);
```

Indicates to OpenGL that the range specified by *offset* and *length* in the mapped buffer bound to *target* has been modified and should be incorporated back into the buffer object's data store.

It is possible to call `glFlushMappedBufferRange()` multiple times on separate or even overlapping ranges of a mapped buffer object. The range of the buffer object specified by *offset* and *length* must lie within the range of buffer object that has been mapped, and that range must have been mapped by a call to `glMapBufferRange()` with *access* including the `GL_MAP_FLUSH_EXPLICIT_BIT` flag set. When this call is made, OpenGL assumes that you're done modifying the specified range of the mapped buffer object, and can begin any operations it needs to perform in order to

4. Don't specify the `GL_MAP_INVALIDATE_BUFFER_BIT` for every section, otherwise only the last section you mapped will have valid data in it!

make that data usable such as copying it to graphics processor visible memory, or flushing, or invalidating data caches. It can do these things even though some or all of the buffer is still mapped. This is a useful way to parallelize OpenGL with other operations that your application might perform. For example, if you need to load a very large piece of data from a file into a buffer, map a range of the buffer large enough to hold the whole file, then read chunks of the file, and after each chunk call **glFlushMappedBufferRange()**. OpenGL will then operate *in parallel* to your application, reading more data from the file for the next chunk.

By combining these flags in various ways, it is possible to optimize data transfer between the application and OpenGL or to use advanced techniques such as *multithreading* or *asynchronous file operations*.

Discarding Buffer Data

Advanced

When you are done with the data in a buffer, it can be advantageous to tell OpenGL that you don't plan to use it any more. For example, consider the case where you write data into a buffer using transform feedback, and then draw using that data. If that drawing command is the last one that is going to access the data, then you can tell OpenGL that it is free to discard the data and use the memory for something else. This allows an OpenGL implementation to make optimizations such as tightly packing memory allocations or avoiding expensive copies in systems with more than one GPU.

To discard some or all of the data in a buffer object, you can call **glInvalidateBufferData()** or **glInvalidateBufferSubData()**, respectively. The prototypes of these functions are as follows:

```
void glInvalidateBufferData(GLuint buffer);  
void glInvalidateBufferSubData(GLuint buffer, GLintptr offset,  
                               GLsizeiptr length);
```

Tell OpenGL that the application is done with the contents of the buffer object in the specified range and that it is free to discard the data if it believes it is advantageous to do so. **glInvalidateBufferSubData()** discards the data in the region of the buffer object whose name is *buffer* starting at *offset* bytes and continuing for *length* bytes. **glInvalidateBufferData()** discards the entire contents of the buffer's data store.

Note that semantically, calling `glBufferData()` with a NULL pointer does a very similar thing to calling `glInvalidateBufferData()`. Both methods will tell the OpenGL implementation that it is safe to discard the data in the buffer. However, `glBufferData()` logically recreates the underlying memory allocation, whereas `glInvalidateBufferData()` does not. Depending on the OpenGL implementation, it may be more optimal to call `glInvalidateBufferData()`. Further, `glInvalidateBufferSubData()` is really the only way to discard a *region* of a buffer object's data store.

Vertex Specification

Now that you have data in buffers, and you know how to write a basic vertex shader, it's time to hook the data up to the shader. You've already read about *vertex array objects*, which contain information about where data is located and how it is laid out, and functions like `glVertexAttribPointer()`. It's time to take a deeper dive into vertex specifications, other variants of `glVertexAttribPointer()`, and how to specify data for vertex attributes that aren't floating point or aren't enabled.

VertexAttribPointer in Depth

The `glVertexAttribPointer()` command was briefly introduced in Chapter 1. The prototype is as follows:

```
void glVertexAttribPointer(GLuint index, GLint size,
                           GLenum type, GLboolean normalized,
                           GLsizei stride, const GLvoid *pointer);
```

Specifies where the data values for the vertex attribute with location *index* can be accessed. *pointer* is the offset in basic-machine units (i.e., bytes) from the start of the buffer object currently bound to the `GL_ARRAY_BUFFER` target for the first set of values in the array. *size* represents the number of components to be updated per vertex. *type* specifies the data type of each element in the array. *normalized* indicates that the vertex data should be normalized before being presented to the vertex shader. *stride* is the byte offset between consecutive elements in the array. If *stride* is zero, the data is assumed to be tightly packed.

The state set by `glVertexAttribPointer()` is stored in the currently bound vertex array object (VAO). *size* is the number of elements in the attribute's vector (1, 2, 3, or 4), or the special token `GL_BGRA`, which should be

specified when *packed* vertex data is used. The *type* parameter is a token that specifies the type of the data that is contained in the buffer object. Table 3.6 describes the token names that may be specified for *type* and the OpenGL data type that they correspond to:

Table 3.6 Values of *Type* for `glVertexAttribPointer()`

Token Value	OpenGL Type
GL_BYTE	GLbyte (signed 8-bit bytes)
GL_UNSIGNED_BYTE	GLubyte (unsigned 8-bit bytes)
GL_SHORT	GLshort (signed 16-bit words)
GL_UNSIGNED_SHORT	GLushort (unsigned 16-bit words)
GL_INT	GLint (signed 32-bit integers)
GL_UNSIGNED_INT	GLuint (unsigned 32-bit integers)
GL_FIXED	GLfixed (16.16 signed fixed point)
GL_FLOAT	GLfloat (32-bit IEEE single-precision floating point)
GL_HALF_FLOAT	GLhalf (16-bit S1E5M10 half-precision floating point)
GL_DOUBLE	GLdouble (64-bit IEEE double-precision floating point)
GL_INT_2_10_10_10_REV	GLuint (packed data)
GL_UNSIGNED_INT_2_10_10_10_REV	GLuint (packed data)

Note that while integer types such as `GL_SHORT` or `GL_UNSIGNED_INT` can be passed to the *type* argument, this tells OpenGL only what data type is stored in memory in the buffer object. OpenGL will convert this data to floating point in order to load it into floating-point vertex attributes. The way this conversion is performed is controlled by the *normalize* parameter. When *normalize* is `GL_FALSE`, integer data is simply typecast into floating-point format before being passed to the vertex shader. This means that if you place the integer value 4 into a buffer and use the `GL_INT` token for the *type* when *normalize* is `GL_FALSE`, the value 4.0 will be placed into the shader. When *normalize* is `GL_TRUE`, the data is normalized before being passed to the vertex shader. To do this, OpenGL divides each element by a fixed constant that depends on the incoming data type. When the data type is signed, the following formula is used:

$$f = \frac{c}{2^b - 1}$$

Whereas, if the data type is unsigned, the following formula is used:

$$f = \frac{2c + 1}{2^b - 1}$$

In both cases, f is the resulting floating-point value, c is the incoming integer component, and b is the number of bits in the data type (i.e., 8 for `GL_UNSIGNED_BYTE`, 16 for `GL_SHORT`, and so on). Note that unsigned data types are also scaled and biased before being divided by the type-dependent constant. To return to our example of putting 4 into an integer vertex attribute, we get:

$$f = \frac{4}{2^{32} - 1}$$

which works out to about 0.000000009313—a pretty small number!

Integer Vertex Attributes

If you are familiar with the way floating-point numbers work, you'll also realize that precision is lost as numbers become very large, and so the full range of integer values cannot be passed into a vertex shader using floating-point attributes. For this reason, we have *integer vertex attributes*. These are represented in vertex shaders by the `int`, `ivec2`, `ivec3`, or `ivec4` types or their unsigned counterparts—`uint`, `uvec2`, `uvec3`, and `uvec4`.

A second vertex-attribute function is needed in order to pass raw integers into these vertex attributes—one that doesn't automatically convert everything to floating point. This is `glVertexAttribIPointer()`—the I stands for integer.

```
void glVertexAttribIPointer(GLuint index, GLint size,  
                           GLenum type, GLsizei stride,  
                           const GLvoid *pointer);
```

Behaves similarly to `glVertexAttribPointer()`, but for vertex attributes declared as integers in the vertex shader. *type* must be one of the integer data type tokens `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, or `GL_UNSIGNED_INT`.

Notice that the parameters to `glVertexAttribIPointer()` are identical to the parameters to `glVertexAttribPointer()`, except for the omission of the

normalize parameter. *normalize* is missing because it's not relevant to integer vertex attributes. Only the integer data type tokens, `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, and `GL_UNSIGNED_INT` may be used for the *type* parameter.

Double-Precision Vertex Attributes

The third variant of `glVertexAttribPointer()` is `glVertexAttribLPointer()`—here the L stands for “long”. This version of the function is specifically for loading attribute data into 64-bit *double-precision* floating-point vertex attributes.

```
void glVertexAttribLPointer(GLuint index, GLint size,  
                           GLenum type, GLsizei stride,  
                           const GLvoid *pointer);
```

Behaves similarly to `glVertexAttribPointer()`, but for vertex attributes declared as 64-bit double-precision floating-point types in the vertex shader. *type* must be `GL_DOUBLE`.

Again, notice the lack of the *normalize* parameter. In `glVertexAttribPointer()`, *normalize* was used only for integer data types that aren't legal here, and so the parameter is not needed. If `GL_DOUBLE` is used with `glVertexAttribPointer()`, the data is automatically down-converted to 32-bit single-precision floating-point representation before being passed to the vertex shader—even if the target vertex attribute was declared using one of the double-precision types `double`, `dvec2`, `dvec3`, or `dvec4`, or one of the double-precision matrix types such as `dmat4`. However, with `glVertexAttribLPointer()`, the full precision of the input data is kept and passed to the vertex shader.

Packed Data Formats for Vertex Attributes

Going back to the `glVertexAttribPointer()` command, you will notice that the allowed values for the *size* parameter are 1, 2, 3, 4, and the special token `GL_BGRA`. Also, the *type* parameter may take one of the special values `GL_INT_2_10_10_10_REV` or `GL_UNSIGNED_INT_2_10_10_10_REV`, both of which correspond to the `GLuint` data type. These special tokens are used to represent *packed* data that can be consumed by OpenGL. The `GL_INT_2_10_10_10_REV` and `GL_UNSIGNED_INT_2_10_10_10_REV` tokens represent four-component data represented as ten bits for each of the first three components and two for the last, packed in reverse order into a single 32-bit quantity (a `GLuint`). `GL_BGRA` could just have easily

been called `GL_ZYXW`.⁵ Looking at the data layout within the 32-bit word, you would see the bits divided up as shown in Figure 3.3.

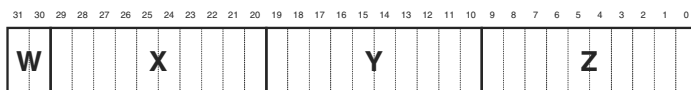


Figure 3.3 Packing of elements in a BGRA-packed vertex attribute

In Figure 3.3, the elements of the vertex are packed into a single 32-bit integer in the order w, x, y, z —which when reversed is z, y, x, w , or b, g, r, a when using color conventions. In Figure 3.4, the coordinates are packed in the order w, z, y, x , which reversed and written in color conventions is r, g, b, a .

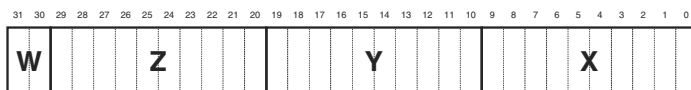


Figure 3.4 Packing of elements in a RGBA-packed vertex attribute

Vertex data may be specified only in the first of these two formats by using the `GL_INT_2_10_10_10_REV` or `GL_UNSIGNED_INT_2_10_10_10_REV` tokens. When one of these tokens is used as the *type* parameter to `glVertexAttribPointer()`, each vertex consumes one 32-bit word in the vertex array. The word is unpacked into its components and then optionally normalized (depending on the value of the *normalize* parameter before being loaded into the appropriate vertex attribute. This data arrangement is particularly well suited to normals or other types of attributes that can benefit from the additional precision afforded by the 10-bit components but perhaps don't require the full precision offered by half-float data (which would take 16-bits per component). This allows the conservation of memory space and bandwidth, which helps improve performance.

Static Vertex-Attribute Specification

Remember from Chapter 1 where you were introduced to `glEnableVertexAttribArray()` and `glDisableVertexAttribArray()`.

⁵ Not a valid OpenGL token; just to be clear.

These functions are used to tell OpenGL which vertex attributes are backed by vertex buffers. Before OpenGL will read any data from your vertex buffers, you must enable the corresponding vertex attribute arrays with `glEnableVertexAttribArray()`. You may wonder what happens if you don't enable the attribute array for one of your vertex attributes. In that case, the *static vertex attribute* is used. The static vertex attribute for each vertex is the default value that will be used for the attribute when there is no enabled attribute array for it. For example, imagine you had a vertex shader that would read the vertex color from one of the vertex attributes. Now suppose that all of the vertices in a particular mesh or part of that mesh had the same color. It would be a waste of memory and potentially of performance to fill a buffer full of that constant value for all the vertices in the mesh. Instead, you can just disable the vertex attribute array and use the static vertex attribute to specify color for all of the vertices.

The static vertex attribute for each attribute may be specified using one of `glVertexAttrib*()` functions. When the vertex attribute is declared as a floating-point quantity in the vertex shader (i.e., it is of type `float`, `vec2`, `vec3`, `vec4`, or one of the floating-point matrix types such as `mat4`), the following `glVertexAttrib*()` commands can be used to set its value.

```
void glVertexAttrib{1234}{fdfs}(GLuint index, TYPE values);
void glVertexAttrib{1234}{fdfs}v(GLuint index,
                                const TYPE *values);
void glVertexAttrib4{bsifd ub us ui}v(GLuint index,
                                       const TYPE *values);
```

Specifies the static value for the vertex attribute with index *index*. For the non-*v* versions, up to four values are specified in the *x*, *y*, *z*, and *w* parameters. For the *v* versions, up to four components are sourced from the array whose address is specified in *v* and used in place of the *x*, *y*, *z*, and *w* components in that order.

All of these functions implicitly convert the supplied parameters to floating-point before passing them to the vertex shader (unless they're already floating-point). This conversion is a simple typecast. That is, the values are converted exactly as specified as if they had been specified in a buffer and associated with a vertex attribute by calling `glVertexAttribPointer()` with the *normalize* parameter set to `GL_FALSE`. For the integer variants of the functions, versions exist that normalize the

parameters to the range $[0, 1]$ or $[-1, 1]$ depending on whether the parameters are signed or unsigned. These are:

```
void glVertexAttrib4Nub(GLuint index, GLubyte x, GLubyte y,  
                       GLubyte z, GLubyte w);  
void glVertexAttrib4Nv{bsi ub us ui}v(GLuint index,  
                                       const TYPE *v);
```

Specifies a single or multiple vertex-attribute values for attribute *index*, normalizing the parameters to the range $[0, 1]$ during the conversion process for the unsigned variants and to the range $[-1, 1]$ for the signed variants.

Even with these commands, the parameters are still converted to floating-point before being passed to the vertex shader. Thus, they are suitable only for setting the static values of attributes declared with one of the single-precision floating-point data types. If you have vertex attributes that are declared as integers or double-precision floating-point variables, you should use one of the following functions:

```
void glVertexAttribI{1234}{i ui}(GLuint index, TYPE values);  
void glVertexAttribI{123}{i ui}v(GLuint index,  
                                 const TYPE *values);  
void glVertexAttribI4{bsi ub us ui}v(GLuint index,  
                                     const TYPE *values);
```

Specifies a single or multiple static integer vertex-attribute values for integer vertex attribute *index*.

Furthermore, if you have vertex attributes that are declared as one of the double-precision floating-point types, you should use one of the L variants of `glVertexAttrib*()`, which are:

```
void glVertexAttribL{1234}(GLuint index, TYPE values);  
void glVertexAttribL{1234}v(GLuint index, const TYPE *values);
```

Specifies a single or multiple static vertex-attribute values for double-precision vertex attribute *index*.

Both the `glVertexAttribI*()` and `glVertexAttribL*()` variants of `glVertexAttrib*()` pass their parameters through to the underlying vertex attribute just as the I versions of `glVertexAttribIPointer()` do.

If you use one of the `glVertexAttrib*()` functions with less components than there are in the underlying vertex attribute (e.g., you use `glVertexAttrib*() 2f` to set the value of a vertex attribute declared as a `vec4`), default values are filled in for the missing components. For w , 1.0 is used as the default value, and for y and z , 0.0 is used.⁶ If you use a function that takes more components than are present in the vertex attribute in the shader, the additional components are simply discarded.

Note: The static vertex attribute values are stored in the current VAO, not the program object. That means that if the current vertex shader has, for example, a `vec3` input and you use `glVertexAttrib*() 4fv` to specify a four-component vector for that attribute, the fourth component will be ignored *but still stored*. If you change the vertex shader to one that has a `vec4` input at that attribute location, the fourth component specified earlier will appear in that attribute's w component.

OpenGL Drawing Commands

Most OpenGL drawing commands start with the word *Draw*.⁷ The drawing commands are roughly broken into two subsets—indexed and nonindexed draws. Indexed draws use an array of indices stored in a buffer object bound to the `GL_ELEMENT_ARRAY_BUFFER` binding that is used to indirectly index into the enabled vertex arrays. On the other hand, nonindexed draws do not use the `GL_ELEMENT_ARRAY_BUFFER` at all, and simply read the vertex data sequentially. The most basic, nonindexed drawing command in OpenGL is `glDrawArrays()`.

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

Constructs a sequence of geometric primitives using array elements starting at *first* and ending at *first + count - 1* of each enabled array. *mode* specifies what kinds of primitives are constructed and is one of the primitive mode tokens such as `GL_TRIANGLES`, `GL_LINE_LOOP`, `GL_LINES`, and `GL_POINTS`.

Similarly, the most basic *indexed* drawing command is `glDrawElements()`.

6. The lack of a default for x is intentional—you can't specify values for y , z , or w without also specifying a value for x .

7. In fact, the only two commands in OpenGL that start with *Draw* but don't draw anything are `glDrawBuffer()` and `glDrawBuffers()`.

```
void glDrawElements(GLenum mode, GLsizei count,
                   GLenum type, const GLvoid *indices);
```

Defines a sequence of geometric primitives using *count* number of elements, whose indices are stored in the buffer bound to the `GL_ELEMENT_ARRAY_BUFFER` buffer binding point (the *element array buffer*). *indices* represents an offset, in bytes, into the element array buffer where the indices begin. *type* must be one of `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, or `GL_UNSIGNED_INT`, indicating the data type of the indices the element array buffer. *mode* specifies what kind of primitives are constructed and is one of the primitive mode tokens, such as `GL_TRIANGLES`, `GL_LINE_LOOP`, `GL_LINES`, and `GL_POINTS`.

Each of these functions causes vertices to be read from the enabled vertex-attribute arrays and used to construct primitives of the type specified by *mode*. Vertex-attribute arrays are enabled using `glEnableVertexAttribArray()` as described in Chapter 1. `glDrawArrays()` just uses the vertices in the buffer objects associated with the enabled vertex attributes in the order they appear. `glDrawElements()` uses the indices in the element array buffer to index into the vertex attribute arrays. Each of the more complex OpenGL drawing functions essentially builds functionality on top of these two functions. For example, `glDrawElementsBaseVertex()` allows the indices in the element array buffer to be offset by a fixed amount.

```
void glDrawElementsBaseVertex(GLenum mode, GLsizei count,
                              GLenum type,
                              const GLvoid *indices,
                              GLint basevertex);
```

Behaves identically to `glDrawElements()` except that the *i*th element transferred by the corresponding draw command will be taken from element `indices[i] + basevertex` of each enabled vertex attribute array.

`glDrawElementsBaseVertex()` allows the indices in the element array buffer to be interpreted relative to some base index. For example, multiple versions of a model (say, frames of an animation) can be stored in a single set of vertex buffers at different offsets within the buffer. `glDrawElementsBaseVertex()` can then be used to draw any frame of that animation by simply specifying the first index that corresponds to that frame. The same set of indices can be used to reference every frame.

Another command that behaves similarly to `glDrawElements()` is `glDrawRangeElements()`.

```
void glDrawRangeElements(GLenum mode, GLuint start,
                        GLuint end, GLsizei count,
                        GLenum type,
                        const GLvoid *indices);
```

This is a restricted form of `glDrawElements()` in that it forms a contract between the application (i.e., *you*) and OpenGL that guarantees that any index contained in the section of the element array buffer referenced by *indices* and *count* will fall within the range specified by *start* and *end*.

Various combinations of functionality are available through even more advanced commands—for example, `glDrawRangeElementsBaseVertex()` combines the features of `glDrawElementsBaseVertex()` with the contractual arrangement of `glDrawRangeElements()`.

```
void glDrawRangeElementsBaseVertex(GLenum mode,
                                   GLuint start, GLuint end,
                                   GLsizei count,
                                   GLenum type,
                                   const GLvoid *indices,
                                   GLint basevertex);
```

Forms a contractual agreement between the application similar to that of `glDrawRangeElements()`, while allowing the base vertex to be specified in *basevertex*. In this case, the contract states that the values stored in the element array buffer will fall between *start* and *end* before *basevertex* is added.

Instanced versions of both of these functions are also available. Instancing will be covered in “Instanced Rendering” on Page 128. The instancing commands include `glDrawArraysInstanced()`, `glDrawElementsInstanced()`, and even `glDrawElementsInstancedBaseVertex()`. Finally, there are two commands that take their parameters not from your program directly, but from a *buffer object*. These are the draw-indirect functions, and to use them, a buffer object must be bound to the `GL_DRAW_INDIRECT_BUFFER` binding. The first is the indirect version of `glDrawArrays()`, `glDrawArraysIndirect()`.

```
void glDrawArraysIndirect(GLenum mode,
                          const GLvoid *indirect);
```

Behaves exactly as **glDrawArraysInstanced()**, except that the parameters for the drawing command are taken from a structure stored in the buffer bound to the `GL_DRAW_INDIRECT_BUFFER` binding point (the *draw indirect buffer*). *indirect* represents an offset into the draw indirect buffer. *mode* is one of the primitive types that is accepted by **glDrawArrays()**.

In **glDrawArraysIndirect()**, the parameters for the actual draw command are taken from a structure stored at offset *indirect* into the *draw indirect* buffer. The structure's declaration in "C" is presented in Example 3.3:

Example 3.3 Declaration of the DrawArraysIndirectCommand Structure

```
typedef struct DrawArraysIndirectCommand_t
{
    GLuint count;
    GLuint primCount;
    GLuint first;
    GLuint baseInstance;
} DrawArraysIndirectCommand;
```

The fields of the `DrawArraysIndirectCommand` structure are interpreted as if they were parameters to a call to **glDrawArraysInstanced()**. *first* and *count* are passed directly to the internal function. The *primCount* field is the instance count, and the *baseInstance* field becomes the *baseInstance* offset to any instanced vertex attributes (don't worry, the instanced rendering commands will be described shortly).

The indirect version of **glDrawElements()** is **glDrawElementsIndirect()** and its prototype is as follows:

```
void glDrawElementsIndirect(GLenum mode, GLenum type,
                            const GLvoid * indirect);
```

Behaves exactly as **glDrawElements()**, except that the parameters for the drawing command are taken from a structure stored in the buffer bound to the `GL_DRAW_INDIRECT_BUFFER` binding point. *indirect* represents an offset into the draw indirect buffer. *mode* is one of the primitive types that is accepted by **glDrawElements()**, and *type* specifies the type of the indices stored in the element array buffer at the time the draw command is called.

As with `glDrawArraysIndirect()`, the parameters for the draw command in `glDrawElementsIndirect()` come from a structure stored at offset *indirect* stored in the element array buffer. The structure's declaration in "C" is presented in Example 3.4:

Example 3.4 Declaration of the DrawElementsIndirectCommand Structure

```
typedef struct DrawElementsIndirectCommand_t
{
    GLuint    count;
    GLuint    primCount;
    GLuint    firstIndex;
    GLuint    baseVertex;
    GLuint    baseInstance;
} DrawElementsIndirectCommand;
```

As with the `DrawArraysIndirectCommand` structure, the fields of the `DrawElementsIndirectCommand` structure are also interpreted as calls to the `glDrawElementsInstancedBaseVertex()` command. *count* and *baseVertex* are passed directly to the internal function. As in `glDrawArraysIndirect()`, *primCount* is the instance count. *firstVertex* is used, along with the size of the indices implied by the *type* parameter to calculate the value of *indices* that would have been passed to `glDrawElementsInstancedBaseVertex()`. Again, *baseInstance* becomes the instance offset to any instanced vertex attributes used by the resulting drawing commands.

Now, we come to the drawing commands that do not start with *Draw*. These are the multivariants of the drawing commands, `glMultiDrawArrays()`, `glMultiDrawElements()`, and `glMultiDrawElementsBaseVertex()`. Each one takes an array of *first* parameters, and an array of *count* parameters acts as if the nonmultiversion of the function had been called once for each element of the array. For example, look at the prototype for `glMultiDrawArrays()`.

```
void glMultiDrawArrays(GLenum mode, const GLint * first,
                      const GLint * count, GLsizei primcount);
```

Draws multiple sets of geometric primitives with a single OpenGL function call. *first* and *count* are arrays of *primcount* parameters that would be valid for a call to `glDrawArrays()`.

Calling `glMultiDrawArrays()` is equivalent to the following OpenGL code sequence:

```
void glMultiDrawArrays(GLenum mode,
                      const GLint * first,
                      const GLint * count,
                      GLsizei primcount)
{
    GLsizei i;

    for (i = 0; i < primcount; i++)
    {
        glDrawArrays(mode, first[i], count[i]);
    }
}
```

Similarly, the multiversion of `glDrawElements()` is `glMultiDrawElements()`, and its prototype is as follows:

```
void glMultiDrawElements(GLenum mode, const GLint * count,
                        GLenum type,
                        const GLvoid * const * indices,
                        GLsizei primcount);
```

Draws multiple sets of geometric primitives with a single OpenGL function call. *first* and *indices* are arrays of *primcount* parameters that would be valid for a call to `glDrawElements()`.

Calling `glMultiDrawElements()` is equivalent to the following OpenGL code sequence:

```
void glMultiDrawElements(GLenum mode,
                        const GLsizei * count,
                        GLenum type,
                        const GLvoid * const * indices,
                        GLsizei primcount)
{
    GLsizei i;

    for (i = 0; i < primcount; i++)
    {
        glDrawElements(mode, count[i], type, indices[i]);
    }
}
```

An extension of `glMultiDrawElements()` to include a *baseVertex* parameter is `glMultiDrawElementsBaseVertex()`. Its prototype is as follows:

```
void glMultiDrawElementsBaseVertex(GLenum mode,
                                   const GLint * count,
                                   GLenum type,
                                   const GLvoid * const * indices,
                                   GLsizei primcount,
                                   const GLint * baseVertex);
```

Draws multiple sets of geometric primitives with a single OpenGL function call. *first*, *indices*, and *baseVertex* are arrays of *primcount* parameters that would be valid for a call to `glDrawElementsBaseVertex()`.

As with the previously described OpenGL multidrawing commands, `glMultiDrawElementsBaseVertex()` is equivalent to another code sequence that ends up calling the nonmultiversion of the function.

```
void glMultiDrawElementsBaseVertex(GLenum mode,
                                   const GLsizei * count,
                                   GLenum type,
                                   const GLvoid * const * indices,
                                   GLsizei primcount,
                                   const \GLint * baseVertex);
{
    GLsizei i;

    for (i = 0; i < primcount; i++)
    {
        glDrawElements(mode, count[i], type,
                      indices[i], baseVertex[i]);
    }
}
```

Finally, if you have a large number of draws to perform and the parameters are already in a buffer object suitable for use by `glDrawArraysIndirect()` or `glDrawElementsIndirect()`, it is possible to use the *multi* versions of these two functions, `glMultiDrawArraysIndirect()` and `glMultiDrawElementsIndirect()`.

```
void glMultiDrawArraysIndirect(GLenum mode,
                               const void * indirect,
                               GLsizei drawcount,
                               GLsizei stride);
```

Draws multiple sets of primitives, the parameters for which are stored in a buffer object. *drawcount* independent draw commands are dispatched as a result of a call to **glMultiDrawArraysIndirect()**, and parameters are sourced from these commands as they would be for **glDrawArraysIndirect()**. Each `DrawArraysIndirectCommand` structure is separated by *stride* bytes. If *stride* is zero, then the data structures are assumed to form a tightly packed array.

```
void glMultiDrawElementsIndirect(GLenum mode,
                                  GLenum type,
                                  const void * indirect,
                                  GLsizei drawcount,
                                  GLsizei stride);
```

Draws multiple sets of primitives, the parameters for which are stored in a buffer object. *drawcount* independent draw commands are dispatched as a result of a call to **glMultiDrawElementsIndirect()**, and parameters are sourced from these commands as they would be for **glDrawElementsIndirect()**. Each `DrawElementsIndirectCommand` structure is separated by *stride* bytes. If *stride* is zero, then the data structures are assumed to form a tightly packed array.

OpenGL Drawing Exercises

This is a relatively simple example of using a few of the OpenGL drawing commands covered so far in this chapter. Example 3.5 shows how the data is loaded into the buffers required to use the draw commands in the example. Example 3.6 shows how the drawing commands are called.

Example 3.5 Setting up for the Drawing Command Example

```
// A four vertices
static const GLfloat vertex_positions[] =
{
    -1.0f, -1.0f, 0.0f, 1.0f,
    1.0f, -1.0f, 0.0f, 1.0f,
    -1.0f, 1.0f, 0.0f, 1.0f,
    -1.0f, -1.0f, 0.0f, 1.0f,
};
```

```

// Color for each vertex
static const GLfloat vertex_colors[] =
{
    1.0f, 1.0f, 1.0f, 1.0f,
    1.0f, 1.0f, 0.0f, 1.0f,
    1.0f, 0.0f, 1.0f, 1.0f,
    0.0f, 1.0f, 1.0f, 1.0f
};

// Three indices (we're going to draw one triangle at a time)
static const GLushort vertex_indices[] =
{
    0, 1, 2
};

// Set up the element array buffer
glGenBuffers(1, ebo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo[0]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER,
             sizeof(vertex_indices), vertex_indices, GL_STATIC_DRAW);

// Set up the vertex attributes
glGenVertexArrays(1, vao);
glBindVertexArray(vao[0]);

glGenBuffers(1, vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
glBufferData(GL_ARRAY_BUFFER,
             sizeof(vertex_positions) + sizeof(vertex_colors),
             NULL, GL_STATIC_DRAW);
glBufferSubData(GL_ARRAY_BUFFER, 0,
               sizeof(vertex_positions), vertex_positions);
glBufferSubData(GL_ARRAY_BUFFER,
               sizeof(vertex_positions), sizeof(vertex_colors),
               vertex_colors);

```

Example 3.6 Drawing Commands Example

```

// DrawArrays
model_matrix = vmath::translation(-3.0f, 0.0f, -5.0f);
glUniformMatrix4fv(render_model_matrix_loc, 4, GL_FALSE, model_matrix);
glDrawArrays(GL_TRIANGLES, 0, 3);

// DrawElements
model_matrix = vmath::translation(-1.0f, 0.0f, -5.0f);
glUniformMatrix4fv(render_model_matrix_loc, 4, GL_FALSE, model_matrix);
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_SHORT, NULL);

// DrawElementsBaseVertex
model_matrix = vmath::translation(1.0f, 0.0f, -5.0f);
glUniformMatrix4fv(render_model_matrix_loc, 4, GL_FALSE, model_matrix);
glDrawElementsBaseVertex(GL_TRIANGLES, 3, GL_UNSIGNED_SHORT, NULL, 1);

```

```
// DrawArraysInstanced
model_matrix = vmath::translation(3.0f, 0.0f, -5.0f);
glUniformMatrix4fv(render_model_matrix_loc, 4, GL_FALSE, model_matrix);
glDrawArraysInstanced(GL_TRIANGLES, 0, 3, 1);
```

The result of the program in Examples 3.5 and 3.6 is shown in Figure 3.5. It's not terribly exciting, but you can see four similar triangles, each rendered using a different drawing command.

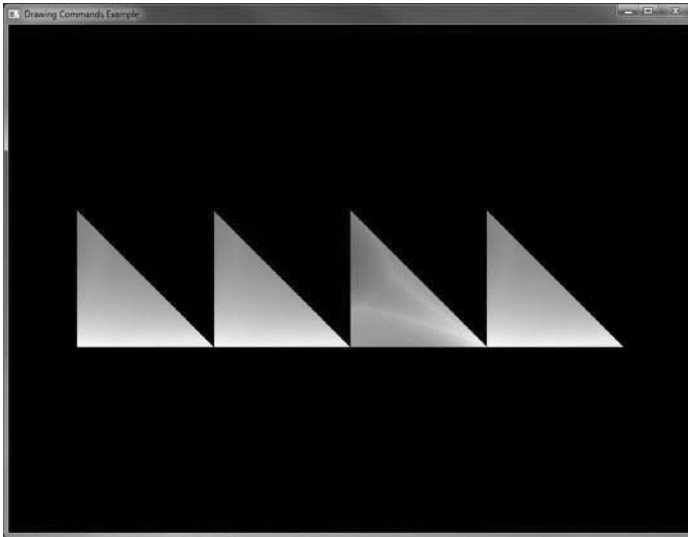


Figure 3.5 Simple example of drawing commands

Restarting Primitives

As you start working with larger sets of vertex data, you are likely to find that you need to make numerous calls to the OpenGL drawing routines, usually rendering the same type of primitive (such as `GL_TRIANGLE_STRIP`) that you used in the previous drawing call. Of course, you can use the `glMultiDraw*()` routines, but they require the overhead of maintaining the arrays for the starting index and length of each primitive.

OpenGL has the ability to restart primitives within the same drawing command by specifying a special value, the *primitive restart index*, which is specially processed by OpenGL. When the primitive restart index is encountered in a draw call, a new rendering primitive of the same type is

started with the vertex following the index. The primitive restart index is specified by the `glPrimitiveRestartIndex()` function.

```
void glPrimitiveRestartIndex(GLuint index);
```

Specifies the vertex array element index used to indicate that a new primitive should be started during rendering. When processing of vertex-array element indices encounters a value that matches `index`, no vertex data is processed, the current graphics primitive is terminated, and a new one of the identical type is started from the next vertex.

As vertices are rendered with one of the `glDrawElements()` derived function calls, it can watch for the index specified by `glPrimitiveRestartIndex()` to appear in the element array buffer. However, it watches only for this index to appear if primitive restating is enabled. Primitive restarting is controlled by calling `glEnable()` or `glDisable()` with the `GL_PRIMITIVE_RESTART` parameter.

To illustrate, consider the layout of vertices in Figure 3.6, which shows how a triangle strip would be broken in two by using primitive restarting. In this figure, the primitive restart index has been set to 8. As the triangles are rendered, OpenGL watches for the index 8 to be read from the element array buffer, and when it sees it go by, rather than creating a vertex, it ends the current triangle strip. The next vertex (vertex 9) becomes the first vertex of a new triangle strip, and so in this case two triangle strips are created.



Figure 3.6 Using primitive restart to break a triangle strip

The following example demonstrates a simple use of primitive restart—it draws a cube as a pair of triangle strips separated by a primitive restart index. Examples 3.7 and 3.8 demonstrate how the data for the cube is specified and then drawn.

Example 3.7 Initializing Data for a Cube Made of Two Triangle Strips

```
// 8 corners of a cube, side length 2, centered on the origin
static const GLfloat cube_positions[] =
{
    -1.0f, -1.0f, -1.0f, 1.0f,
    -1.0f, -1.0f, 1.0f, 1.0f,
    -1.0f, 1.0f, -1.0f, 1.0f,
```

```

        -1.0f, 1.0f, 1.0f, 1.0f,
        1.0f, -1.0f, -1.0f, 1.0f,
        1.0f, -1.0f, 1.0f, 1.0f,
        1.0f, 1.0f, -1.0f, 1.0f,
        1.0f, 1.0f, 1.0f, 1.0f
    };

    // Color for each vertex
    static const GLfloat cube_colors[] =
    {
        1.0f, 1.0f, 1.0f, 1.0f,
        1.0f, 1.0f, 0.0f, 1.0f,
        1.0f, 0.0f, 1.0f, 1.0f,
        1.0f, 0.0f, 0.0f, 1.0f,
        0.0f, 1.0f, 1.0f, 1.0f,
        0.0f, 1.0f, 0.0f, 1.0f,
        0.0f, 0.0f, 1.0f, 1.0f,
        0.5f, 0.5f, 0.5f, 1.0f
    };

    // Indices for the triangle strips
    static const GLushort cube_indices[] =
    {
        0, 1, 2, 3, 6, 7, 4, 5,          // First strip
        0xFFFF,                          // <<-- This is the restart index
        2, 6, 0, 4, 1, 5, 3, 7          // Second strip
    };

    // Set up the element array buffer
    glGenBuffers(1, ebo);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo[0]);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER,
                 sizeof(cube_indices),
                 cube_indices, GL_STATIC_DRAW);

    // Set up the vertex attributes
    glGenVertexArrays(1, vao);
    glBindVertexArray(vao[0]);

    glGenBuffers(1, vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    glBufferData(GL_ARRAY_BUFFER,
                 sizeof(cube_positions) + sizeof(cube_colors),
                 NULL, GL_STATIC_DRAW);
    glBufferSubData(GL_ARRAY_BUFFER, 0,
                   sizeof(cube_positions), cube_positions);
    glBufferSubData(GL_ARRAY_BUFFER, sizeof(cube_positions),
                   sizeof(cube_colors), cube_colors);

    glVertexAttribPointer(0, 4, GL_FLOAT,
                          GL_FALSE, 0, NULL);
    glVertexAttribPointer(1, 4, GL_FLOAT,
                          GL_FALSE, 0,

```

```

        (const GLvoid *)sizeof(cube_positions));
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);

```

Figure 3.7 shows how the vertex data given in Example 3.7 represents the cube as two independent triangle strips.

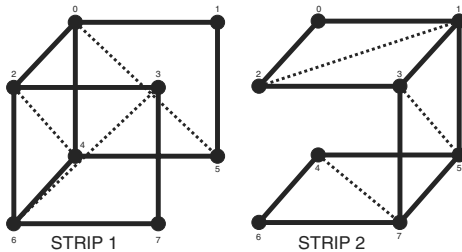


Figure 3.7 Two triangle strips forming a cube

Example 3.8 Drawing a Cube Made of Two Triangle Strips Using Primitive Restart

```

// Set up for a glDrawElements call
glBindVertexArray(vao[0]);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo[0]);

#if USE_PRIMITIVE_RESTART
// When primitive restart is on, we can call one draw command
glEnable(GL_PRIMITIVE_RESTART);
glPrimitiveRestartIndex(0xFFFF);
glDrawElements(GL_TRIANGLE_STRIP, 17, GL_UNSIGNED_SHORT, NULL);
#else
// Without primitive restart, we need to call two draw commands
glDrawElements(GL_TRIANGLE_STRIP, 8, GL_UNSIGNED_SHORT, NULL);
glDrawElements(GL_TRIANGLE_STRIP, 8, GL_UNSIGNED_SHORT,
               (const GLvoid *) (9 * sizeof(GLushort)));
#endif
#endif

```

Note: OpenGL will restart primitives whenever it comes across the current restart index in the element array buffer. Therefore, it's a good idea to set the restart index to a value that will not be used in your code. The default restart index is zero, which is very likely to appear in your element array buffer. A good value to choose is $2^n - 1$, where n is the number of bits in your indices (i.e., 16 for `GL_UNSIGNED_SHORT` indices and 32 for `GL_UNSIGNED_INT` indices). This is very unlikely to be used as a real index. Sticking with such a standard also means that you don't need to figure out the index for every model in your program.

Instanced Rendering

Instancing, or instanced rendering, is a way of executing the same drawing commands many times in a row, with each producing a slightly different result. This can be a very efficient method of rendering a large amount of geometry with very few API calls. Several variants of already-familiar drawing functions exist to instruct OpenGL to execute the command multiple times. Further, various mechanisms are available in OpenGL to allow the shader to use the instance of the draw as an input, and to be given new values for vertex attributes *per-instance* rather than per-vertex. The simplest instanced rendering call is:

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

Draws *primCount* instances of the geometric primitives specified by *mode*, *first*, and *count* as if specified by individual calls to **glDrawArrays()**. The built-in variable `gl_InstanceID` is incremented for each instance, and new values are presented to the vertex shader for each *instanced* vertex attribute.

This is the instanced version of **glDrawArrays()**; note similarity of the two functions. The parameters of **glDrawArraysInstanced()** are identical to those of **glDrawArrays()**, with the addition of the *primCount* argument. This parameter specifies the count of the number of instances that are to be rendered. When this function is executed, OpenGL will essentially execute *primCount* copies of **glDrawArrays()**, with the *mode*, *first*, and *count* parameters passed through. There are ***Instanced** versions of several of the OpenGL drawing commands, including **glDrawElementsInstanced()** (for **glDrawElements()**) and **glDrawElementsInstancedBaseVertex()** (for **glDrawElementsBaseVertex()**). The **glDrawElementsInstanced()** function is defined as:

```
void glDrawElementsInstanced(GLenum mode, GLsizei count,  
                             GLenum type,  
                             const void* indices,  
                             GLsizei primCount);
```

Draws *primCount* instances of the geometric primitives specified by *mode*, *count* and *indices* as if specified by individual calls to **glDrawElements()**. As with **glDrawArraysInstanced()**, the built-in variable `gl_InstanceID` is incremented for each instance, and new values are presented to the vertex shader for each *instanced* vertex attribute.

Again, note that the parameters to `glDrawElementsInstanced()` are identical to `glDrawElements()`, with the addition of *primCount*. Each time one of the instanced functions is called, OpenGL essentially runs the whole command as many times as is specified by the *primCount* parameter. This on its own is not terribly useful. However, there are two mechanisms provided by OpenGL that allow vertex attributes to be specified as *instanced* and to provide the vertex shader with the index of the current instance.

```
void glDrawElementsInstancedBaseVertex(GLenum mode,
                                       GLsizei count,
                                       GLenum type,
                                       const void* indices,
                                       GLsizei instanceCount,
                                       GLuint baseVertex);
```

Draws *instanceCount* instances of the geometric primitives specified by *mode*, *count*, *indices*, and *baseVertex* as if specified by individual calls to `glDrawElementsBaseVertex()`. As with `glDrawArraysInstanced()`, the built-in variable `gl_InstanceID` is incremented for each instance, and new values are presented to the vertex shader for each *instanced* vertex attribute.

Instanced Vertex Attributes

Instanced vertex attributes behave similarly to regular vertex attributes. They are declared and used in exactly the same way inside the vertex shader. On the application side, they are also configured in the same way as regular vertex attributes. That is, they are backed by buffer objects, can be queried with `glGetAttribLocation()`, set up using `glVertexAttribPointer()`, and enabled and disabled using `glEnableVertexAttribArray()` and `glDisableVertexAttribArray()`. The important new function that allows a vertex attribute to become instanced is as follows:

```
void glVertexAttribDivisor(GLuint index, GLuint divisor);
```

Specifies the rate at which new values of the instanced the vertex attribute at *index* are presented to the vertex shader during instanced rendering. A *divisor* value of 0 turns off instancing for the specified attribute, whereas any other value of *divisor* indicates that a new value should be presented to the vertex shader each *divisor* instances.

The `glVertexAttribDivisor()` function controls the *rate* at which the vertex attribute is updated. *index* is the index of the vertex attribute whose divisor is to be set, and is the same as you would pass into `glVertexAttribPointer()` or `glEnableVertexAttribArray()`. By default, a new value of each enabled attribute is delivered to each vertex. Setting *divisor* to zero resets the attribute to this behavior and makes it a regular, noninstanced attribute. A nonzero value of *divisor* makes the attribute instanced and causes a new value to be fetched from the attribute array once every *divisor* instances rather than for every vertex. The index within the enabled vertex attribute array from which the attribute is taken is then $\frac{\text{instance}}{\text{divisor}}$, where *instance* is the current instance number and *divisor* is the value of *divisor* for the current attribute. For each of the instanced vertex attributes, the same value is delivered to the vertex shader for all vertices in the instance. If *divisor* is two, the value of the attribute is updated every second instance; if it is three then the attribute is updated every third instance, and so on. Consider the vertex attributes declared in Example 3.9, some of which will be configured as instanced.

Example 3.9 Vertex Shader Attributes for the Instancing Example

```
#version 410 core

// "position" and "normal" are regular vertex attributes
layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;

// Color is a per-instance attribute
layout (location = 2) in vec4 color;

// model_matrix will be used as a per-instance transformation
// matrix. Note that a mat4 consumes 4 consecutive locations, so
// this will actually sit in locations, 3, 4, 5, and 6.
layout (location = 3) in mat4 model_matrix;
```

Note that in Example 3.9, there is nothing special about the declaration of the instanced vertex attributes `color` and `model_matrix`. Now consider the code shown in Example 3.10, which configures a subset of vertex attributes declared in Example 3.9 as instanced.

Example 3.10 Example Setup for Instanced Vertex Attributes

```
// Get the locations of the vertex attributes in "prog", which is
// the (linked) program object that we're going to be rendering
// with. Note that this isn't really necessary because we specified
// locations for all the attributes in our vertex shader. This code
// could be made more concise by assuming the vertex attributes are
// where we asked the compiler to put them.
int position_loc = glGetAttribLocation(prog, "position");
int normal_loc = glGetAttribLocation(prog, "normal");
int color_loc = glGetAttribLocation(prog, "color");
```

```

int matrix_loc      = glGetAttribLocation(prog, "model_matrix");

// Configure the regular vertex attribute arrays -
// position and normal.
glBindBuffer(GL_ARRAY_BUFFER, position_buffer);
glVertexAttribPointer(position_loc, 4, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(position_loc);
glBindBuffer(GL_ARRAY_BUFFER, normal_buffer);
glVertexAttribPointer(normal_loc, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(normal_loc);

// Now we set up the color array. We want each instance of our
// geometry to assume a different color, so we'll just pack colors
// into a buffer object and make an instanced vertex attribute out
// of it.
glBindBuffer(GL_ARRAY_BUFFER, color_buffer);
glVertexAttribPointer(color_loc, 4, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(color_loc);
// This is the important bit... set the divisor for the color array
// to 1 to get OpenGL to give us a new value of "color" per-instance
// rather than per-vertex.
glVertexAttribDivisor(color_loc, 1);

// Likewise, we can do the same with the model matrix. Note that a
// matrix input to the vertex shader consumes N consecutive input
// locations, where N is the number of columns in the matrix. So...
// we have four vertex attributes to set up.
glBindBuffer(GL_ARRAY_BUFFER, model_matrix_buffer);
// Loop over each column of the matrix...
for (int i = 0; i < 4; i++)
{
    // Set up the vertex attribute
    glVertexAttribPointer(matrix_loc + i,           // Location
                          4, GL_FLOAT, GL_FALSE,   // vec4
                          sizeof(mat4),           // Stride
                          (void *) (sizeof(vec4) * i)); // Start offset

    // Enable it
    glEnableVertexAttribArray(matrix_loc + i);
    // Make it instanced
    glVertexAttribDivisor(matrix_loc + i, 1);
}

```

In Example 3.10, `position` and `normal` are regular, noninstanced vertex attributes. However, `color` is configured as an instanced vertex attribute with a divisor of one. This means that each instance will have a new value for the `color` attribute (which will be constant across all vertices in the instance). Further, the `model_matrix` attribute will also be made instanced to provide a new model transformation matrix for each instance. A `mat4` attribute is consuming a consecutive location. Therefore, we loop over each column in the matrix and configure it separately. The remainder of the vertex shader is shown in Example 3.11.

Example 3.11 Instanced Attributes Example Vertex Shader

```
// The view matrix and the projection matrix are constant
// across a draw
uniform mat4 view_matrix;
uniform mat4 projection_matrix;

// The output of the vertex shader (matched to the
// fragment shader)
out VERTEX
{
    vec3    normal;
    vec4    color;
} vertex;

// Ok, go!
void main(void)
{
    // Construct a model-view matrix from the uniform view matrix
    // and the per-instance model matrix.
    mat4 model_view_matrix = view_matrix * model_matrix;

    // Transform position by the model-view matrix, then by the
    // projection matrix.
    gl_Position = projection_matrix * (model_view_matrix *
                                     position);

    // Transform the normal by the upper-left-3x3-submatrix of the
    // model-view matrix
    vertex.normal = mat3(model_view_matrix) * normal;
    // Pass the per-instance color through to the fragment shader.
    vertex.color = color;
}
```

The code to set the model matrices for the instances and then draw the instanced geometry using these shaders is shown in Example 3.12. Each instance has its own model matrix, whereas the view matrix (consisting of a rotation around the y axis followed by a translation in z) is common to all instances. The model matrices are written directly into the buffer by mapping it using `glMapBuffer()`. Each model matrix translates the object away from the origin and then rotates the translated model around the origin. The view and projection matrices are simply placed in uniform variables. Then, a single call to `glDrawArraysInstanced()` is used to draw all instances of the model.

Example 3.12 Instancing Example Drawing Code

```
// Map the buffer
mat4 * matrices = (mat4 *)glMapBuffer(GL_ARRAY_BUFFER,
                                     GL_WRITE_ONLY);

// Set model matrices for each instance
```

```

for (n = 0; n < INSTANCE_COUNT; n++)
{
    float a = 50.0f * float(n) / 4.0f;
    float b = 50.0f * float(n) / 5.0f;
    float c = 50.0f * float(n) / 6.0f;

    matrices[n] = rotation(a + t * 360.0f, 1.0f, 0.0f, 0.0f) *
        rotation(b + t * 360.0f, 0.0f, 1.0f, 0.0f) *
        rotation(c + t * 360.0f, 0.0f, 0.0f, 1.0f) *
        translation(10.0f + a, 40.0f + b, 50.0f + c);
}

// Done. Unmap the buffer.
glUnmapBuffer(GL_ARRAY_BUFFER);

// Activate instancing program
glUseProgram(render_prog);

// Set up the view and projection matrices
mat4 view_matrix(translation(0.0f, 0.0f, -1500.0f) *
    rotation(t * 360.0f * 2.0f, 0.0f, 1.0f, 0.0f));
mat4 projection_matrix(frustum(-1.0f, 1.0f,
    -aspect, aspect, 1.0f, 5000.0f));

glUniformMatrix4fv(view_matrix_loc, 1,
    GL_FALSE, view_matrix);
glUniformMatrix4fv(projection_matrix_loc, 1,
    GL_FALSE, projection_matrix);

// Render INSTANCE_COUNT objects
glDrawArraysInstanced(GL_TRIANGLES, 0, object_size, INSTANCE_COUNT);

```

The result of the program is shown in Figure 3.8. In this example, the constant `INSTANCE_COUNT` (which is referenced in the code of Examples 3.10 and 3.12) is 100. One hundred copies of the model are drawn, each with a different position and a different color. These models could very easily be trees in a forest, space ships in a fleet, or buildings in a city.

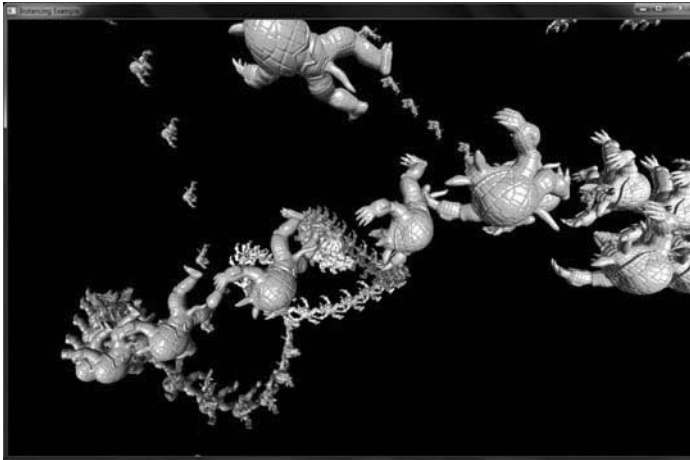


Figure 3.8 Result of rendering with instanced vertex attributes

There are some inefficiencies in the example shown in Examples 3.9 through 3.12. Work that will produce the same result across all of the vertices in an instance will still be performed per-vertex. Sometimes there are ways to get around this. For example, the computation of `model_view_matrix` will evaluate to the same matrix for all vertices within a single instance. Here, we could avoid this work by using a second instanced `mat4` attribute to carry the per-instance model-view matrix. In other cases, it may not be possible to avoid this work, but it may be possible to move it into a geometry shader so that work is performed once per-primitive rather than once per-vertex, or perhaps use geometry shader instancing instead. Both of these techniques will be explained in Chapter 10.

Note: Remember that calling an instanced drawing command is mostly equivalent to calling its noninstanced counterpart many times before executing any other OpenGL commands. Therefore, converting a sequence of OpenGL functions called inside a loop to a sequence of instanced draw calls will not produce identical results.

Another example of a way to use instanced vertex attributes is to pack a set of textures into a 2D array texture and then pass the array slice to be used for each instance in an instanced vertex attribute. The vertex shader can then pass the instance's slice into the fragment shader, which can then render each instance of the geometry with a different texture.

It is possible to internally add an offset to the indices used to fetch instanced vertex attributes from vertex buffers. Similar to the *baseVertex* parameter that is available through `glDrawElementsBaseVertex()`, the instance offset is exposed through an additional *baseInstance* parameter in some versions of the instanced drawing functions. The functions that take a *baseInstance* parameter are `glDrawArraysInstancedBaseInstance()`, `glDrawElementsInstancedBaseInstance()`, and `glDrawElementsInstancedBaseVertexBaseInstance()`. Their prototypes are as follows:

```
void glDrawArraysInstancedBaseInstance(GLenum mode,
                                       GLint first,
                                       GLsizei count,
                                       GLsizei instanceCount,
                                       GLuint baseInstance);
```

Draws *primCount* instances of the geometric primitives specified by *mode*, *first*, and *count* as if specified by individual calls to `glDrawArrays()`. The built-in variable `gl_InstanceID` is incremented for each instance, and new values are presented to the vertex shader for each *instanced* vertex attribute. Furthermore, the implied index used to fetch any instanced vertex attributes is offset by the value of *baseInstance* by OpenGL.

```
void glDrawElementsInstancedBaseInstance(GLenum mode,
                                       GLsizei count,
                                       GLenum type,
                                       const GLvoid * indices,
                                       GLsizei instanceCount,
                                       GLuint baseInstance);
```

Draws *primCount* instances of the geometric primitives specified by *mode*, *count*, and *indices* as if specified by individual calls to `glDrawElements()`. As with `glDrawArraysInstanced()`, the built-in variable `gl_InstanceID` is incremented for each instance, and new values are presented to the vertex shader for each *instanced* vertex attribute. Furthermore, the implied index used to fetch any instanced vertex attributes is offset by the value of *baseInstance* by OpenGL.

```
void glDrawElementsInstancedBaseVertexBaseInstance(GLenum mode,
                                                    GLsizei count,
                                                    GLenum type,
                                                    const GLvoid * indices,
                                                    GLsizei instanceCount,
                                                    GLuint baseVertex,
                                                    GLuint baseInstance);
```

Draws *instanceCount* instances of the geometric primitives specified by *mode*, *count*, *indices*, and *baseVertex* as if specified by individual calls to `glDrawElementsBaseVertex()`. As with `glDrawArraysInstanced()`, the built-in variable `gl_InstanceID` is incremented for each instance, and new values are presented to the vertex shader for each *instanced* vertex attribute. Furthermore, the implied index used to fetch any instanced vertex attributes is offset by the value of *baseInstance* by OpenGL.

Using the Instance Counter in Shaders

In addition to instanced vertex attributes, the index of the current instance is available to the vertex shader in the built-in variable `gl_InstanceID`. This variable is implicitly declared as an integer. It starts counting from zero and counts up one each time an instance is rendered.

`gl_InstanceID` is always present in the vertex shader, even when the current drawing command is not one of the instanced ones. In those cases, it will just be zero. The value in `gl_InstanceID` may be used to index into uniform arrays, perform texture lookups, as the input to an analytic function, or for any other purpose.

In the following example, the functionality of Examples 3.9 through 3.12 is replicated by using `gl_InstanceID` to index into texture buffer objects (TBOs) rather than through the use of instanced vertex attributes. Here, the vertex attributes of Example 3.9 are replaced with TBO lookups, and so are removed from the vertex attribute setup code. Instead, a first TBO containing color of each instance, and a second TBO containing the model matrices are created. The vertex attribute declaration and setup code are the same as in Examples 3.9 and 3.10 (with the omission of the `color` and `model_matrix` attributes, of course). As the instance's color and model matrix is now explicitly fetched in the vertex shader, more code is added to the body of the vertex shader, which is shown in Example 3.13.

Example 3.13 `gl_VertexID` Example Vertex Shader

```
// The view matrix and the projection matrix are constant across a draw
uniform mat4 view_matrix;
uniform mat4 projection_matrix;
```

```

// These are the TBOs that hold per-instance colors and per-instance
// model matrices
uniform samplerBuffer color_tbo;
uniform samplerBuffer model_matrix_tbo;

// The output of the vertex shader (matched to the fragment shader)
out VERTEX
{
    vec3    normal;
    vec4    color;
} vertex;

// Ok, go!
void main(void)
{
    // Use gl_InstanceID to obtain the instance color from the color TBO
    vec4 color = texelFetch(color_tbo, gl_InstanceID);

    // Generating the model matrix is more complex because you can't
    // store mat4 data in a TBO. Instead, we need to store each
    // matrix as four vec4 variables and assemble the matrix in the
    // shader. First, fetch the four columns of the matrix
    // (remember, matrices are stored in memory in column-major
    // order).
    vec4 col1 = texelFetch(model_matrix_tbo, gl_InstanceID * 4);
    vec4 col2 = texelFetch(model_matrix_tbo, gl_InstanceID * 4 + 1);
    vec4 col3 = texelFetch(model_matrix_tbo, gl_InstanceID * 4 + 2);
    vec4 col4 = texelFetch(model_matrix_tbo, gl_InstanceID * 4 + 3);

    // Now assemble the four columns into a matrix.
    mat4 model_matrix = mat4(col1, col2, col3, col4);

    // Construct a model-view matrix from the uniform view matrix
    // and the per-instance model matrix.
    mat4 model_view_matrix = view_matrix * model_matrix;

    // Transform position by the model-view matrix, then by the
    // projection matrix.
    gl_Position = projection_matrix * (model_view_matrix *
                                     position);

    // Transform the normal by the upper-left-3x3-submatrix of the
    // model-view matrix
    vertex.normal = mat3(model_view_matrix) * normal;
    // Pass the per-instance color through to the fragment shader.
    vertex.color = color;
}

```

To drive the shader of Example 3.13, we need to create and initialize TBOs to back the `color_tbo` and `model_matrix_tbo` samplers rather than initializing the instanced vertex attributes. However, aside from the differences in setup code, the program is essentially unchanged.

Example 3.14 contains the code to set up the TBOs for use with the shader of Example 3.13.

Example 3.14 Example Setup for Instanced Vertex Attributes

```
// Get the locations of the vertex attributes in "prog", which is
// the (linked) program object that we're going to be rendering
// with. Note that this isn't really necessary because we specified
// locations for all the attributes in our vertex shader. This code
// could be made more concise by assuming the vertex attributes are
// where we asked the compiler to put them.
int position_loc    = glGetAttribLocation(prog, "position");
int normal_loc      = glGetAttribLocation(prog, "normal");

// Configure the regular vertex attribute arrays - position and normal.
glBindBuffer(GL_ARRAY_BUFFER, position_buffer);
glVertexAttribPointer(position_loc, 4, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(position_loc);
glBindBuffer(GL_ARRAY_BUFFER, normal_buffer);
glVertexAttribPointer(normal_loc, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(normal_loc);

// Now set up the TBOs for the instance colors and model matrices...

// First, create the TBO to store colors, bind a buffer to it and
// initialize its format. The buffer has previously been created
// and sized to store one vec4 per-instance.
glGenTextures(1, &color_tbo);
glBindTexture(GL_TEXTURE_BUFFER, color_tbo);
glTexBuffer(GL_TEXTURE_BUFFER, GL_RGBA32F, color_buffer);

// Now do the same thing with a TBO for the model matrices. The
// buffer object (model_matrix_buffer) has been created and sized
// to store one mat4 per-instance.
glGenTextures(1, &model_matrix_tbo);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_BUFFER, model_matrix_tbo);
glTexBuffer(GL_TEXTURE_BUFFER, GL_RGBA32F, model_matrix_buffer);
```

Note that the code in Example 3.14 is actually shorter and simpler than that in Example 3.10. This is because we have shifted the responsibility for fetching per-instance data from built-in OpenGL functionality to the shader writer. This can be seen in the increased complexity of Example 3.13 relative to Example 3.11. With this responsibility comes additional power and flexibility. For example, if the number of instances is small, it may be preferable to use a uniform array rather than a TBO for data storage, which may increase performance. Regardless, there are very few other changes that need to be made to the original example to move to using explicit fetches driven by `gl_InstanceID`. In fact, the rendering

code of Example 3.12 is used intact to produce an identical result to the original program. The proof is in the screenshot (Figure 3.9).

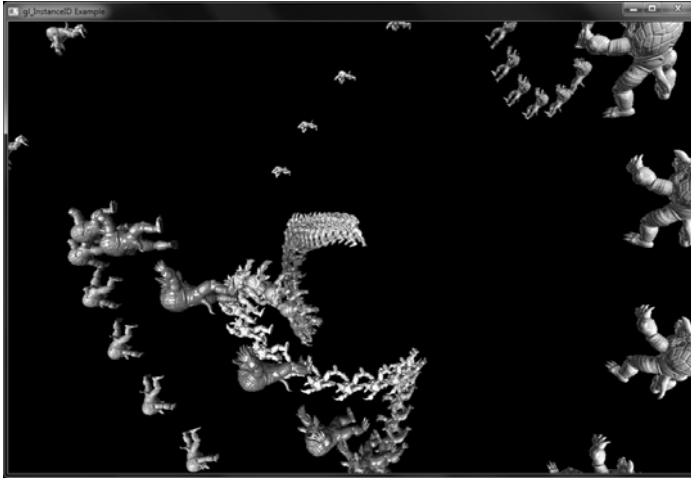


Figure 3.9 Result of instanced rendering using `gl_InstanceID`

Instancing Redux

To use a instancing in your program

- Create some vertex shader inputs that you intend to be instanced.
- Set the vertex attribute divisors with `glVertexAttribDivisor()`.
- Use the `gl_InstanceID` built-in variable in the vertex shader.
- Use the instanced versions of the rendering functions such as `glDrawArraysInstanced()`/`glDrawElementsInstanced()`, or `glDrawElementsInstancedBaseVertex()`.

This page intentionally left blank

- abs(), 692
- acos(), 689
- acosh(), 690
- adjacency primitives, 511, 516–523
- aliasing, 178, 442
- all(), 705
- alpha, 25, 143, 166
- alpha value, 166
- ambient light, 361, 363
- amplification
 - geometry, 527
- analytic integration, 452
- anisotropic filtering, 330
- antialiasing, 153, 442–459
- any(), 705
- application programming interface, 2
- area sampling, 453
- array textures, 262
- arrays, 44
- asin(), 689
- asinh(), 690
- atan(), 689
- atanh(), 690
- atomic counter, 604–608, 624, 629
- atomic operation, 577
 - on image variables, 578
- atomicAdd(), 588, 724
- atomicAnd(), 588, 724
- atomicCompSwap(), 588, 725
- atomicCounter(), 723
- atomicCounterDecrement(), 723
- atomicCounterIncrement(), 723
- atomicExchange(), 588, 725
- atomicMax(), 588, 724
- atomicMin(), 588, 724
- atomicOr(), 588, 725
- atomicXor(), 588, 725
- attenuation, 368
- barrier, 599
 - memory, 599
- barrier(), 734
- barycentric coordinates, 493
- Bernstein polynomials, 503
- Bézier patches, 500
- billboard, 525
- binding an object, 17
- binomial coefficient, 503
- binormal, 435
- bit depth, 143
- bitCount(), 707
- bitfieldExtract(), 706
- bitfieldInsert(), 706
- bitfieldReverse(), 707
- bitplane, 148
- blending, 14, 166, 616
- buffer
 - pixel unpack, 280
 - shader storage, 575

buffer objects, 11
buffer ping-ponging, 181
built-in variables
 compute shader, 630
 geometry shader, 561
bump map, 441
bump mapping, 433–442
byte swapping, 289

cache, 596
 coherency, 596
 hierarchy, 596
callback
 function, 868
callback(), 869
cascading style sheet, 662
ceil(), 693
CGL
 CGLChoosePixelFormat(),
 851
 CGLCreateContext(), 852
 CGLDescribeRenderer(), 851
 CGLDestroyPixelFormat(), 851
 CGLDestroyRendererInfo(), 851
 CGLQueryRendererInfo(), 851
 CGLSetCurrentContext(), 852
ChoosePixelFormat(), 846
clamp(), 694
client, 3
clip, 211
clip coordinates, 213
clipping, 13, 206
 frustum, 211
 user, 237, 238
clipping region, 11
coherent, 598
communication, 632
compatibility profile, 842
components, 42
compressed texture, 260
compression, 179
compression ratio, 326
compute shader, 36, 623–649
conditional rendering, 176

constructor, 40
constructors, 39
context, 15
 debug, 866
control texture, 414
controlling polygon rendering, 90
convex, 90
convolution, 453
convolution filter, 453
convolution kernel, 453
coordinate system, 205
coordinate systems, 208
core profile, 15
cos(), 688
cosh(), 689
cracking, 505
CreateDIBitmap(), 847
CreateDIBSection(), 847
cross(), 700
cube map, 262, 559
culling
 frustum, 211
 in a geometry shader, 527
current, 18

deadlock, 601
Debug
 Groups, 875
debug context, 865, 866
debug message, 869
debug output, 868
default framebuffer, 145
degrees(), 688
DeleteObject(), 847
deprecated, 658
depth buffer, 146, 375, 400
depth coordinate, 209
depth fighting, 404
depth range, 208, 236
depth testing, 13
depth texture, 400
depth value, 13
DescribePixelFormat(), 846
determinant(), 703

dFdx(), 729
dFdy(), 729
diffuse light, 361
directional light, 365
dispatch, 627
 indirect, 628
displacement mapping, 487, 507
display, 4
display callback, 15
display(), 8, 9, 15, 18, 28–30,
 651
distance(), 700
dithering, 171
dot(), 700
double buffering, 146
dual-source blending, 168
dynamically uniform, 297

edge detection, 643
emission, 384
emissive lighting, 380
EmitStreamVertex(), 733
EmitVertex(), 733
end
 of the universe, 882
EndPrimitive(), 733
EndStreamPrimitive(), 733
environment map, 313, 559
environment mapping, 313
equal(), 704
event loop, 8
exp(), 690
exp2(), 691
exponent, 274
eye coordinates, 209, 232, 382
eye space, 208, 209, 382

faceforward(), 701
faces, 90
factorial, 503
far plane, 211, 227, 236
feedback, 206, 239
feedback buffer objects, 239
fence, 589

filtering, 203
 debug messages, 872
 linear, 330
findLSB(), 707
findMSB(), 707
fixed-function pipeline, 34
flat shading, 153
floatBitsToInt(), 696
floatBitsToUint(), 696
floor(), 692
fma(), 697
fonts, 842
fract(), 693
fractional brownian motion, 463
fragment, 3, 144
fragment discard, 13
fragment shader, 3, 4, 35
fragment shading, 13
fragment tests
 early, 604
framebuffer, 4, 145
framebuffer attachment, 183
framebuffer object, 145
framebuffer rendering loop, 351
frequency clamping, 457
frexp(), 697
front facing, 91
frustum, 210
 clipping, 211
 culling, 211
function overloading, 296
fwidth(), 729

gamut, 143
GdiFlush(), 847
geometric model, xli
geometric object, 173
geometric primitive, 2
geometry shader, 35, 509
geometry shaders, 562
GetVersion(), 846
GetVersionEx(), 846
gl.h, 836
gl3.h, 836

glActiveSampler(), 294
 glActiveShaderProgram(), 83
 glActiveTexture(), 265, 294,
 303–305, 571
 glAttachShader(), 74, 625, 647
 glBeginConditionalRender(),
 176, 177
 glBeginQuery(), 173, 174, 881, 882
 glBeginQueryIndexed(), 538
 glBeginTransformFeedback(),
 250–252
 glBind*(), 18, 181
 glBindAttribLocation(), 436
 glBindBuffer(), 18–20, 63, 69, 94,
 242, 891
 glBindBufferBase(), 63, 64, 241, 242
 glBindBufferRange(), 63, 64,
 242–244
 glBindFragDataLocation(), 194, 195
 glBindFragDataLocationIndexed(),
 194, 195
 glBindFramebuffer(), 181, 182, 191
 glBindImageTexture(), 570, 571
 glBindProgramPipeline(), 82
 glBindRenderbuffer(), 184, 185
 glBindSampler(), 293, 891
 glBindTexture(), 264–266, 293,
 303–305, 309, 356, 571,
 891, 894
 glBindTransformFeedback(), 240
 glBindVertexArray(), 8, 17, 18, 29
 glBlendColor(), 168, 169
 glBlendEquation(), 170, 609
 glBlendEquationi(), 170
 glBlendEquationSeparate(), 170
 glBlendEquationSeparatei(), 170
 glBlendFunc(), 167, 168, 171, 198,
 200, 609
 glBlendFunci(), 167,
 168, 198
 glBlendFuncSeparate(), 167, 168,
 171, 198
 glBlendFuncSeparatei(), 167, 168,
 198
 glBlitFramebuffer(), 203
 glBufferData(), 11, 21, 22, 63, 69,
 95, 97, 99–103, 108, 242, 243,
 357, 602
 glBufferSubData(), 97–101, 103,
 242, 602
 glCheckFramebufferStatus(), 190,
 191, 557
 glClampColor(), 203
 glClear(), 8, 28, 29, 147, 190
 glClearBuffer(), 192
 glClearBuffer*(), 190
 glClearBufferData(), 98, 99
 glClearBufferfi(), 190, 192
 glClearBufferfv(), 192
 glClearBufferiv(), 192
 glClearBufferSubData(), 98, 99
 glClearColor(), 29, 147, 190
 glClearDepth(), 147, 190
 glClearDepthf(), 147
 glClearStencil(), 147
 glClientWaitSync(), 590, 591, 593
 glColorMask(), 147, 148
 glColorMaski(), 148
 glCommand{suffix}(), xlv
 glCompileShader(), 73, 511,
 624, 647
 glCompressedTexImage1D(), 327
 glCompressedTexImage2D(), 327
 glCompressedTexImage3D(), 327,
 328
 glCompressedTexSubImage1D(),
 328
 glCompressedTexSubImage2D(),
 328
 glCompressedTexSubImage3D(),
 329
 glCopyBufferSubData(), 99–101,
 602
 glCopyTexImage*(), 196, 197
 glCopyTexImage1D(), 281, 282
 glCopyTexImage2D(), 281, 282
 glCopyTexSubImage*(), 196, 197
 glCopyTexSubImage1D(), 282

glCopyTexSubImage2D(), 282
 glCopyTexSubImage3D(), 282
 glcorearb.h, 9, 836
 glCreateProgram(), 73, 76, 878
 glCreateShader(), 72, 76, 510, 511, 624, 627, 647, 878
 glCreateShaderProgramv(), 81
 glCullFace(), 91, 92, 160
 glDebugMessageCallback(), 868, 869
 glDebugMessageControl(), 872
 glDebugMessageInsert(), 874
 glDeleteBuffers(), 20
 glDeleteFramebuffers(), 182, 183
 glDeleteProgram(), 75, 76
 glDeleteProgramPipelines(), 82
 glDeleteQueries(), 174, 176
 glDeleteRenderbuffers(), 184
 glDeleteSamplers(), 295
 glDeleteShader(), 74, 75
 glDeleteSync(), 591
 glDeleteTextures(), 266
 glDeleteTransformFeedbacks(), 241
 glDeleteVertexArrays(), 17, 18
 glDepthFunc(), 163
 glDepthMask(), 148
 glDepthRange(), 236
 glDepthRangeArrayv(), 551, 555
 glDepthRangef(), 236
 glDepthRangeIndexed(), 550, 551
 glDetachShader(), 74
 glDisable(), 31, 92, 125, 156, 157, 160, 169–172, 871
 glDisablei(), 170, 197
 glDisableVertexArray(), 28, 112, 129
 glDispatchCompute(), 627, 628, 630, 632, 648, 899
 glDispatchComputeIndirect(), 628, 629, 632, 648
 glDrawArrays(), 11, 30, 115–119, 128, 135, 487, 518, 539, 540, 628, 639
 glDrawArraysIndirect(), 117–119, 121, 122, 628
 glDrawArraysInstanced(), 117, 118, 128, 129, 132, 135, 136, 139, 541, 549
 glDrawArraysInstancedBase Instance(), 135
 glDrawBuffer(), 115, 196
 glDrawBuffers(), 115, 191, 192, 196
 glDrawElements(), 93, 115–118, 120, 125, 128, 129, 135, 518, 522
 glDrawElementsBaseVertex(), 116, 117, 121, 128, 129, 135, 136
 glDrawElementsIndirect(), 118, 119, 121, 122
 glDrawElementsInstanced(), 117, 128, 129, 139, 549
 glDrawElementsInstancedBase Instance(), 135
 glDrawElementsInstancedBase Vertex(), 117, 119, 128, 129, 139
 glDrawElementsInstancedBaseVertex BaseInstance(), 135, 136
 glDrawRangeElements(), 117, 806
 glDrawRangeElementsBaseVertex(), 117
 glDrawTransformFeedback(), 540, 548
 glDrawTransformFeedback Instance(), 540, 541
 glDrawTransformFeedbackStream(), 540, 541, 546, 547
 glDrawTransformFeedbackStream Instance(), 541
 glEnable(), 31, 91, 92, 125, 154–158, 160, 163, 164, 166, 170–172, 179, 180, 316, 511, 871, 872
 glEnablei(), 170, 197
 glEnableVertexArray(), 27, 28, 112, 113, 116, 129, 130
 glEndConditionalRender(), 176, 177
 glEndQuery(), 173–175, 881, 882
 glEndQueryIndexed(), 539

glEndTransformFeedback(), 252, 540
 GLEW
 glwInit(), 9, 15, 837
 glew.h, 837
 glExt.h, 9, 836, 837
 glFenceSync(), 589, 591–593
 glFinish(), 31, 106, 841, 847
 glFlush(), 30, 31, 590, 842
 glFlushMappedBufferRange(),
 105–107
 glFramebufferParameteri(), 183
 glFramebufferRenderbuffer(), 187,
 188
 glFramebufferTexture(), 351, 352
 glFramebufferTexture1D(), 351, 352
 glFramebufferTexture2D(), 351, 352
 glFramebufferTexture3D(), 351, 352,
 354
 glFramebufferTextureLayer(), 354,
 557
 glFrontFace(), 91, 899
 glGenBuffers(), 19, 20, 92–94, 356,
 877, 878
 glGenMipmaps(), 337
 glGenFramebuffers(), 181–183
 glGenProgramPipelines(), 82, 878
 glGenQueries(), 173, 174, 176, 539,
 878, 881
 glGenRenderbuffers(), 183, 184, 188
 glGenSamplers(), 292, 293, 295, 878
 glGenTextures(), 264–266, 319, 352,
 354, 356, 570, 877, 878
 glGenTransformFeedbacks(), 240
 glGenVertexArrays(), 17, 18
 glGet*(), 745
 glGetActiveAtomicCounter
 Bufferiv(), 738
 glGetActiveAttrib(), 738, 784, 785
 glGetActiveSubroutineName(), 738
 glGetActiveSubroutineUniformiv(),
 738
 glGetActiveSubroutineUniform
 Name(), 738
 glGetActiveUniform(), 304, 738, 784
 glGetActiveUniformBlockiv(), 63,
 739
 glGetActiveUniformBlockName(),
 739
 glGetActiveUniformName(), 739
 glGetActiveUniformsiv(), 65, 739,
 786, 787, 792
 glGetAttachedShaders(), 739, 783
 glGetAttribLocation(), 129, 739, 784
 glGetBooleani_v(), 739, 770, 798
 glGetBooleanv(), 739, 745, 755, 770,
 778, 779, 799, 807, 827
 glGetBufferParameteri64v(), 739
 glGetBufferParameteriv(), 739
 glGetBufferPointerv(), 739, 750
 glGetBufferSubData(), 100, 101,
 242, 532, 602, 739
 glGetCompressedTexImage(), 739
 glGetDebugMessageLog(), 739
 glGetDoublei_v(), 740, 751
 glGetDoublev(), 740, 745
 glGetError(), 591, 738, 740, 827
 glGetFloati_v(), 740, 751
 glGetFloatv(), 740, 745, 746,
 753–755, 769, 770, 804–806,
 825
 glGetFragDataIndex(), 195, 740
 glGetFragDataLocation(), 195, 740
 glGetFramebufferAttachment
 Parameteriv(), 740
 glGetFramebufferParameteriv(), 740
 glGetInteger64i_v(), 740, 748, 786,
 799–801
 glGetInteger64v(), 740, 820, 825
 glGetIntegeri_v(), 740, 748, 755,
 767, 769, 786, 798–801, 816
 glGetIntegerv(), 154, 157, 160, 527,
 533, 549, 555, 559, 593, 634,
 738, 740, 745, 746, 748, 749,
 751–754, 756, 757, 766–772,
 775, 778–780, 783, 786,
 799–801, 803–827, 847, 876
 glGetInternalformati64v(), 740
 glGetInternalformativ(), 740, 826

glGetMultisamplefv(), 154, 741, 827
 glGetObjectLabel(), 741, 748, 750,
 761, 765, 772, 777, 781, 782,
 784, 797, 799, 802, 877, 878
 glGetObjectPtrLabel(), 741, 877, 878
 glGetPointerv(), 741
 glGetProgramBinary(), 741, 784
 glGetProgramInfoLog(), 75, 741,
 783
 glGetProgramInterfaceiv(), 741
 glGetProgramiv(), 74, 629, 630, 741,
 783–786, 788–790
 glGetProgramPipelineInfoLog(), 741
 glGetProgramPipelineiv(), 741
 glGetProgramResourceIndex(), 741
 glGetProgramResourceiv(), 742
 glGetProgramResourceLocation(),
 741
 glGetProgramResourceLocation
 Index(), 742
 glGetProgramResourceName(), 742
 glGetProgramStageiv(), 742, 789,
 790
 glGetQuery*(), 176
 glGetQueryIndexediv(), 742
 glGetQueryiv(), 742, 825, 827
 glGetQueryObjecti64v(), 742
 glGetQueryObjectiv(), 175, 742, 797
 glGetQueryObjectui64v(), 742, 882
 glGetQueryObjectuiv(), 175, 539,
 742, 797, 881, 883
 glGetRenderbufferParameteriv(),
 742
 glGetSamplerParameterfv(), 742
 glGetSamplerParameteriv(), 743
 glGetSamplerParameteruiv(), 743
 glGetSamplerParameteriv(), 742
 glGetShaderInfoLog(), 73, 743, 781
 glGetShaderiv(), 73, 743, 781
 glGetShaderPrecisionFormat(), 743
 glGetShaderSource(), 743, 781
 glGetString(), 738, 743, 808
 glGetStringi(), 743, 808, 847
 glGetSubroutineIndex(), 79, 743
 glGetSubroutineUniformLocation(),
 79, 743
 glGetSynciv(), 589, 590, 743, 802
 glGetTexImage(), 93, 287, 288, 291,
 602, 743, 757, 758
 glGetTexLevelParameterfv(), 743
 glGetTexLevelParameteriv(), 743
 glGetTexParameter*(), 759–761
 glGetTexParameterfv(), 744, 760
 glGetTexParameteriv(), 744
 glGetTexParameterLuiV(), 744
 glGetTexParameteriv(), 744, 760,
 761
 glGetTransformFeedbackVarying(),
 744
 glGetUniform*(), 784
 glGetUniformBlockIndex(), 63, 744
 glGetUniformdv(), 744
 glGetUniformfv(), 744
 glGetUniformIndices(), 65, 744
 glGetUniformiv(), 744
 glGetUniformLocation(), 47, 569,
 744, 784
 glGetUniformLocationSubroutineuiv(), 744
 glGetUniformLocationiv(), 744
 glGetVertexAttribdv(), 745
 glGetVertexAttribfv(), 745, 797
 glGetVertexAttribiv(), 745
 glGetVertexAttribLuiV(), 745
 glGetVertexAttribiv(), 745
 glGetVertexAttribLdv(), 745
 glGetVertexAttribPointerv(), 745
 glHint(), 178, 179
 glInvalidateBufferData(), 107, 108
 glInvalidateBufferSubData(), 107,
 108
 glInvalidateFramebuffer(), 192, 193,
 354, 355
 glInvalidateSubFramebuffer(), 192,
 193, 354, 355
 glInvalidateTexImage(), 355
 glInvalidateTexSubImage(), 355
 glIsBuffer(), 20
 glIsEnabled(), 32, 157, 161, 738,

745, 749, 751, 753–756,
 767–769, 797, 827
 glIsEnabledi(), 197, 767, 769
 glIsFramebuffer(), 182, 183
 glIsProgram(), 76
 glIsQuery(), 174
 glIsRenderbuffer(), 184
 glIsSampler(), 293
 glIsShader(), 76
 glIsSync(), 592
 glIsTexture(), 265, 266
 glIsTransformFeedback(), 240
 glIsVertexArray(), 18
 glLineWidth(), 88
 glLinkProgram(), 47, 61, 64, 74,
 245, 537, 625, 627, 647
 glLogicOp(), 172
 glMapBuffer(), 61, 101–104, 132,
 532, 807
 glMapBufferRange(), 104–106, 591
 glMemoryBarrier(), 601, 603
 glMinSampleShading(), 155, 156
 glMultiDrawArrays(), 119, 120
 glMultiDrawArraysIndirect(), 121,
 122
 glMultiDrawElements(), 119–121
 glMultiDrawElementsBaseVertex(),
 119, 121
 glMultiDrawElementsIndirect(),
 121, 122
 global illumination, 384
 glObjectLabel(), 877, 878
 glObjectPtrLabel(), 877, 878
 glPatchParameterfv(), 496, 501
 glPatchParameteri(), 487–489, 500
 glPauseTransformFeedback(), 251,
 252
 glPixelStore*(), 669
 glPixelStoref(), 288
 glPixelStorei(), 288
 glPointParameter(), 350
 glPointParameterf(), 350
 glPointParameteri(), 350
 glPointSize(), 87, 88
 glPolygonMode(), 90, 91, 164
 glPolygonOffset(), 164–166
 glPopDebugGroup(), 870, 876, 877
 glPrimitiveRestartIndex(), 125
 glProgramParameteri(), 81
 glProgramUniform(), 83
 glProgramUniform*(), 83
 glProgramUniformMatrix(), 83
 glProgramUniformMatrix*(), 83
 glProvokingVertex(), 524
 glPushDebugGroup(), 870, 876, 877
 glQueryCounter(), 882, 883
 glReadBuffer(), 191, 196, 197, 201
 glReadPixels(), 93, 196, 197,
 200–202, 282
 glRenderbufferStorage(), 185, 186
 glRenderbufferStorageMultisample(),
 185, 186
 glResumeTransformFeedback(), 251,
 252
 glSampleCoverage(), 158
 glSampleMaski(), 158, 159
 glSamplerParameterf(), 294, 338
 glSamplerParameterfv(), 294
 glSamplerParameteri(), 294, 318,
 338, 339
 glSamplerParameteriv(), 294
 glSamplerParameterI{*i ui*}v(), 294
 glSamplerParameter{*fi*}(), 294
 glSamplerParameter{*fi*}v(), 294
 glScissor(), 157
 glScissorArrayv(), 555
 glScissorIndexed(), 554, 555
 glScissorIndexedv(), 554
 glShaderSource(), 72, 511, 627, 647
 GLSL, 23, 34–84
 .length(), 44
 #version, 23, 36
 arrays, 44
 barrier(), 489, 634, 635, 644
 blocks, 60
 bool, 40
 boolean types, 38
 buffer block, 49

GLSL *continued*

- buffer blocks, 69
- buffer, 46, 49, 60, 62, 636
- centroid, 730
- clamp(), 445
- column_major, 62
- const, 46, 54
- constructors, 39
- control flow, 52
- conversions, 39
- defined, 57, 58
- dFdx(), 450, 459
- dFdy(), 450, 459
- discard, 88
- dmat4, 111
- do-while loop, 52
- double, 39, 40, 49, 111
- dvec2, 111
- dvec3, 111
- dvec4, 111
- extensions, 59
- false, 39, 695, 696
- flat, 424, 730
- float, 39, 40, 49, 113, 426
- floating-point types, 38
- for loop, 52
- functions, 52
- fwidth(), 450, 453, 458, 459
- groupMemoryBarrier(), 636
- if-else statement, 51
- image, 564
- implicit conversions, 39
- in blocks, 70
- in, 37, 46, 54, 60, 155, 451, 491
- inout, 54
- int, 40, 49, 110, 426
- integer types, 38
- interface block, 60
- invariant, 55
- isolines, 497
- ivec2, 110
- ivec3, 110
- ivec4, 110
- layout, 194, 489, 490, 497, 498, 500, 502, 886
- length(), 44, 45, 69
- location, 194
- mat4, 113
- matrix, 40, 42
- memoryBarrier(), 635, 636
- memoryBarrierAtomicCounter(), 635
- memoryBarrierBuffer(), 635
- memoryBarrierImage(), 635
- memoryBarrierShared(), 635
- mix(), 445, 481
- noperspective, 730
- operator precedence, 49
- operators, 49
- out blocks, 70
- out, 37, 46, 54, 60, 194, 417, 424, 490, 491, 497
- packed, 62
- parameter qualifiers, 53
- precise qualifier, 55
- precise, 507
- preprocessor, 56–59
 - #extension, 59
 - #if,#else,#endif, 57
 - #version, 23, 36
 - __FILE__, 58
 - __LINE__, 58
 - __VERSION__, 58
- quads, 497, 500, 502, 505
- row_major, 62
- sample, 154, 155
- shared storage, 49
- shared, 46, 49, 62
- smoothstep(), 418, 426, 427, 445, 449, 451, 459
- std140, 62
- std430, 62
- step(), 448
- storage qualifiers, 45
- structures, 43
- subroutines, 76
- switch statement, 51

GLSL *continued*
 triangles, 497
 true, 39, 695, 696, 705
 uint, 40, 49, 110
 uniform block, 61–69
 uniform, 37, 46, 47, 49, 60, 62,
 235, 363, 366, 368, 380
 uvec2, 110
 uvec3, 110
 uvec4, 110
 vec2, 113
 vec3, 113, 115, 452, 480
 vec4, 37, 113, 115, 425, 426, 502,
 577, 893
 component names, 42
 vectors, 40, 42
 while loop, 52
 glStencilFunc(), 148, 159
 glStencilFuncSeparate(), 159, 160
 glStencilMask(), 148
 glStencilMaskSeparate(), 148
 glStencilOp(), 159, 160
 glStencilOpSeparate(), 160
 glTexBuffer(), 319, 357
 glTexBufferRange(), 320
 glTexImage*D(), 602
 glTexImage1D(), 268–270, 279, 282
 glTexImage2D(), 94, 268–270, 282,
 357, 570
 glTexImage2DMultisample(),
 268–270
 glTexImage3D(), 268–270, 279, 291,
 292, 307
 glTexImage3DMultisample(),
 268–270
 glTexParameterf(), 295, 338
 glTexParameterfv(), 295
 glTexParameteri(), 295, 302, 318,
 319, 408
 glTexParameterIiv(), 295
 glTexParameterIuiv(), 295
 glTexParameteriv(), 295, 302
 glTexParameterI{f|u|v}(), 295
 glTexParameter{f|i}(), 295
 glTexParameter{f|i}v(), 295
 glTexStorage1D(), 266, 267, 270,
 327
 glTexStorage2D(), 266, 267, 270,
 285, 310, 322, 327, 337, 339,
 356, 357
 glTexStorage2DMultisample(), 267,
 268
 glTexStorage3D(), 266–268, 270,
 307, 327, 570
 glTexStorage3DMultisample(), 267,
 268
 glTexSubImage*D(), 602
 glTexSubImage1D(), 278
 glTexSubImage2D(), 267, 270, 274,
 278–281, 285, 288, 290, 310,
 337, 356
 glTexSubImage3D(), 278, 291, 292
 glTextureView(), 322
 glTransformFeedbackVaryings(),
 244, 245, 536, 537
 glUniform(), 48
 glUniform*(), 9, 47, 48, 83
 glUniform1i(), 304, 305, 356, 564,
 569
 glUniform2f(), 9
 glUniform2fv(), 9
 glUniform3fv(), 9
 glUniformBlockBinding(), 64
 glUniformMatrix(), 48
 glUniformMatrix*(), 47, 48, 83
 glUniformMatrix4fv(), 554
 glUniformSubroutinesuiv(), 80
 glUnmapBuffer(), 102–106
 glUseProgram(), 75, 81–83, 627,
 629, 648, 667
 glUseProgramStages(), 82
 GLUT, 8
 glutCreateWindow(), 15, 181,
 652, 653, 654, 837
 glutDisplayFunc(), 15, 16, 655,
 658
 glutGetProcAddress(), 654, 655,
 836

GLUT *continued*
 glutIdleFunc(), 658
 glutInit(), 15, 652
 glutInitContextFlags(), 652, 866
 glutInitContextProfile(), 15, 23, 652
 glutInitContextVersion(), 15, 652
 glutInitDisplayMode(), 15, 181, 652
 glutInitWindowPosition(), 652
 glutInitWindowSize(), 15, 652
 glutKeyboardFunc(), 656
 glutKeyboardUpFunc(), 656
 glutMainLoop(), 16, 654, 658
 glutPassiveMotionEvent(), 657
 glutPostRedisplay(), 655, 657
 glutReshapeFunc(), 656
 glutSwapBuffers(), 146, 896
 glVertexAttrib(), 113
 glVertexAttrib*(), 113–115
 glVertexAttrib4(), 113
 glVertexAttrib4N(), 114
 glVertexAttrib4N*(), 26
 glVertexAttrib4Nub(), 114
 glVertexAttribDivisor(), 129, 130, 139
 glVertexAttribI(), 114
 glVertexAttribI*(), 114
 glVertexAttribI4(), 114
 glVertexAttribIPointer(), 110, 114
 glVertexAttribL(), 114
 glVertexAttribL*(), 114
 glVertexAttribLPointer(), 111
 glVertexAttribN*(), 149
 glVertexAttribPointer(), 26, 27, 30, 93, 108–113, 129, 130, 149
 glVertexport(), 236, 656
 glVertexportArrayv(), 551, 555
 glVertexportIndexedf(), 550, 551, 554
 glVertexportIndexedfv(), 550, 551
 glWaitSync(), 593, 825
 GLX
 glXChooseFBConfig(), 839
 glXChooseVisual(), 840
 glXCopyContext(), 841
 glXCreateContext(), 841
 glXCreateContextAttribsARB(), 840, 867
 glXCreateGLXPixmap(), 840
 glXCreateNewContext(), 840
 glXCreatePbuffer(), 840
 glXCreatePixmap(), 840
 glXCreateWindow(), 840
 glXDestroyContext(), 841
 glXDestroyGLXPixmap(), 842
 glXDestroyPbuffer(), 842
 glXDestroyPixmap(), 842
 glXDestroyWindow(), 842
 glXGetClientString(), 839
 glXGetConfig(), 840
 glXGetCurrentContext(), 841
 glXGetCurrentDisplay(), 841
 glXGetCurrentDrawable(), 841
 glXGetCurrentReadDrawable(), 841
 glXGetFBConfigAttrib(), 839
 glXGetProcAddress(), 840
 glXGetSelectedEvent(), 841
 glXGetVisualFromFBConfig(), 839
 glXIsDirect(), 840
 glXMakeContextCurrent(), 840
 glXMakeCurrent(), 841
 glXQueryContext(), 841
 glXQueryExtension(), 839
 glXQueryExtensionsString(), 839
 glXQueryServerString(), 839
 glXQueryVersion(), 839
 glXSelectEvent(), 841
 glXSwapBuffers(), 842
 glXUseXFont(), 842
 glXWaitGL(), 841, 847
 glXWaitX(), 841, 842, 847
 glxext.h, 840
 gl_ClipDistance, 238
 GL_CLIP_PLANE0, 239
 Gouraud shading, 153
 GPGPU, 181, 194
 GPU, 4

gradient vector, 450
gradient, 450
graphics processing unit, 4
greaterThan(), 704
greaterThanEqual(), 704
groupMemoryBarrier(), 735

half space, 89
halo, 384
hemispherical lighting, 384
hidden-line removal, 164
hidden-surface removal, 145
homogeneous clip coordinates, 209
homogeneous coordinate, 206
homogeneous coordinates, 209,
215, 216

image processing, 642
image, xli
image-based lighting, 389
imageAtomicAdd(), 582, 727
imageAtomicAnd(), 582, 728
imageAtomicCompSwap(), 583, 729
imageAtomicExchange(), 582, 583,
728
imageAtomicMax(), 582, 728
imageAtomicMin(), 582, 728
imageAtomicOr(), 582, 728
imageAtomicXor(), 582, 728
imageLoad(), 572, 727
imageSize(), 573, 574, 727
imageStore(), 573, 727
immutable, 267
impostor, 558
imulExtended(), 706
init(), 7, 8, 15–17, 24, 25, 27, 29,
147, 663, 664, 837
input-patch vertex, 489
instanced rendering, 85
instancing
 geometry shader, 548–550
intBitsToFloat(), 696
interface block, 60, 514, 516
interpolateAtCentroid(), 730
interpolateAtOffset(), 730, 731
interpolateAtSample(), 730
interpolation, 153
invariance, 54
inverse(), 703
inversesqrt(), 691
invocation, 55
invocation
 compute shader, 625
isinf(), 696
isnan(), 696

jaggies, 178

lacunarity, 466
layered rendering, 525, 550–559
layout qualifier, 24, 511, 566, 626
ldexp(), 697
length(), 700
lens flare, 384
lessThan(), 704
lessThanEqual(), 704
level of detail, 333, 340
light probe, 389
lighting, 70
 ambient, 363
 directional, 365
 emissive, 380
 hemispherical, 384
 image based, 389
 material properties, 379
 multiple lights, 376
 point lights, 368
 spherical harmonics, 395
 spotlight, 370
 two sided, 381
linked list, 610
LoadImage.h, 283
LoadShaders(), 8, 22, 23, 76, 664,
665, 667
LoadShaders.h, 22
local viewer, 383
log(), 691
log2(), 691

logical operation, 171
lossless compression, 326
lossy compression, 326
low-pass filtering, 449
luminance, 43

Mac OS X
 NSOpenGLContext(), 854
 NSOpenGLPixelFormat(), 854
 NSOpenGLView(), 854

magnification, 329
mantissa, 274
material, 70
matrix, xlii, 214
matrix
 column major, 234
 multiplication, 214
 OpenGL, 232
 row major, 234
matrixCompMult(), 702
max(), 694
memory
 read only, 598
memoryBarrier(), 735
memoryBarrierAtomicCounter(),
 735
memoryBarrierBuffer(), 735
memoryBarrierImage(), 735
memoryBarrierShared(), 735
min(), 693, 694
minification, 329
 magnification, 335
mipmap, 262
mipmaps, 332
mix(), 695
mod(), 693
model coordinates, 209
modeling transformation, 207
models, 4
modf(), 693
modulate, 362
monitor, 181
multisampling, 146, 153, 262
mutex, 585
nanosecond, 881
near plane, 211, 227, 236
network, 3
noise, 459–483
 gradient noise, 464
 granite, 478
 marble, 477
 octave, 463
 turbulence, 475
 value noise, 462
 wood, 478
noise1(), 732
noise2(), 732
noise3(), 732
noise4(), 732
normal, 362
normal vectors
 transforming, 831
normal map, 441
normal maps, 441
normalize(), 701
normalized device coordinates, 209
normalized homogeneous
 coordinates, 208
normalized value, 149
normalized-device coordinates, 22
not(), 705
notEqual(), 705

object, xli
 label, 877
object coordinates, 209
occlusion query, 173
octave, 463
off-screen rendering, 181
opaque types, 38
optimization
 loop hoisting, 594
orthographic projection, 230
orthographic viewing model, 212
orthographic, 212
outerProduct(), 702
overloading, 40

pack, 97
packDouble2x32(), 699
packHalf2x16(), 699
packSnorm2x16(), 698
packSnorm4x8(), 698
packUnorm2x16(), 698
packUnorm4x8(), 698
pass-through shader, 12, 501
patch, 12, 485, 486
performance profiling, 879
Perlin noise, 460
perspective correction, 730
perspective division, 213
perspective projection, 207, 210, 227
Phong reflection model, 376
Phong shading, 376
pixel, 4
point fade threshold, 350
point lighting, 368
point sampling, 444
point sprites, 346
point, 4
polygon culling, 91
polygon faces, 91
polygon offset, 404
polygon, 90
pow(), 690
primitive generator, 487
procedural shading, 412
procedural texture shader, 412
procedural texturing, 412–433
 brick, 419
 lattice, 431
 regular patterns, 414
 toy ball, 422
programmable blending, 609
projective texturing, 342, 408
protocol, 3
proxy texture, 260, 276

quadrilateral, 299
queries
 timer, 881

race condition, 645
radians(), 688
rasterization, 3
rasterization
 disabling, 511
rasterizer, 152, 153
ray tracing, 4
readonly, 598
reflect(), 701
refract(), 701
renderbuffer, 145, 181
renderer, 851
rendering pipeline, 10
rendering, 4
resolved, 154
restrict, 595
restricted pointer, 595
RGB color space, 25, 143
RGBA mode, 29
RGBA, 143
rotation, 224
round(), 692
roundEven(), 693

sample shader, 155
sample, 153
sampler, 629
 object, 292
sampler variables, 262
samples, 35
scaling, 221
scissor box, 157
scissoring, 157
selector, 265
separate shader objects, 81
server, 3
SetPixelFormat(), 846
shader, 3
 compiling, 627
 compute, 36, 623–649
 fragment, 35
 geometry, 35, 510
 subroutines, 76
 tessellation, 35, 510

vertex, 35
shader plumbing, 8
shader program, 8
shader stage, 4
shader storage buffer object, 69
shader storage buffer, 46
shader variable, 23
shading, xliii
shadow coordinates, 406
shadow map, 400
shadow mapping, 400
shadow sampler, 317
shadow texture, 402
shared exponent, 274
shared variables, 633
sign(), 692
sin(), 688
sinh(), 689
sky box, 312
slice, 261, 262
smoothstep(), 695
sorting, 616
spaces
 clip space, 209
 eye space, 209
 eye, 383
 model space, 209
 object space, 209
specular light, 362
spherical harmonic lighting, 395
spotlight, 370
sprite, 88
sqrt(), 691
sRGB color space, 143, 274
state, 4, 29
stencil buffer, 146
stencil testing, 13
step(), 695
stereo, 146
stipple, 163
storage qualifiers, 45
structures, 43
subpixel, 146
subroutines, 76
supersampling, 445
surface-local coordinate space, 434
surface-local coordinates, 434
SwapBuffers(), 847
sync object, 589
synchronization, 577, 634
tan(), 688
tangent space, 435
tanh(), 690
temporal aliasing, 444
tessellated, 12
tessellation
 control shaders, 488–491
 bypassing, 495
 gl_in variable, 490
 gl_out variable, 490
 other variables, 491
 pass-through, 495
 cracking along shared edges, 506
 displacement mapping, 507
 domains
 isolines, 493
 quads, 491–493
 selecting, 497
 triangles, 493–495
 evaluation shaders, 496–510
 coordinate spacing options, 498
 gl_in variable, 499
 gl_out variable, 500
 other variables, 500
 patches, 487, 488
 primitive winding, 497
 tessellation coordinates, 498
 view-dependent, 504–506
tessellation control shader, 486
tessellation coordinates, 487
tessellation domain, 485
tessellation evaluation shader, 486
tessellation level factor, 489
tessellation output patch vertices,
 488
tessellation shader, 35, 514
tessellation shaders, 486

texelFetch(), 321, 714
texelFetchOffset(), 714
texels, 261
texture
 array, 262
 buffer, 319–321, 572
 compressed, 326–329
 cube map, 559
 gathering texels, 345
 immutable storage, 357
 proxy, 276, 277
 rectangle, 263
 target, 263
 unit, 262
 view, 321–325
 writing to, 574
texture comparison mode, 402
texture coordinates, 153, 261
texture map, 14, 149
texture mapping, 8, 149
texture object, 261
texture sampler, 262
texture streaming, 339
texture swizzle, 302
texture targets, 262
texture unit, 262
texture view, 322
texture(), 296, 309, 317, 318, 711
textureGather(), 345, 721
textureGatherOffset(), 721, 722
textureGatherOffsets(), 722
textureGrad(), 341, 717
textureGradOffset(), 346, 718
textureLod(), 340, 712
textureLodOffset(), 715
textureOffset(), 341, 342, 713
textureProj(), 342, 343, 712
textureProjGrad(), 346, 719
textureProjGradOffset(), 346, 719, 720
textureProjLod(), 346, 716
textureProjLodOffset(), 346, 716
textureProjOffset(), 346, 715
textureQueryLevels(), 344, 711
textureQueryLod(), 343, 710
textures
 binding to image units, 569
textureSize(), 344, 709, 710
timeout, 590
transform feedback, 239, 532–548
 objects, 239
 particle system example, 252
 starting and stopping, 250
 varyings, 244
transform feedback object, 239
transformation matrices, 12, 831
transformation matrix
 projection
 orthographic, 230
 perspective, 227
 rotation, 224
 scaling, 221
 translation, 219
transformations
 model-view, 209
 modeling, 207
 normals, 231
 orthographic projection, 230
 perspective projection, 207, 210, 227
 rotation, 224
 scaling, 221
 translation, 219
 viewing, 207
 viewport, 209
translation, 219
transparency
 order independent, 609
transparent types, 38
transpose(), 702, 703
trunc(), 692
turbulence, 475
txtureProj(), 342
typed array, 667
uaddCarry(), 705
uintBitsToFloat(), 696
umulExtended(), 706

uniform block, 61–69, 629
uniform buffer object, 61
uniform variable, 46
unit square, 491
universe
 end of, 882
unpackDouble2x32(), 699
unpackHalf2x16(), 699
unpackSnorm2x16(), 698
unpackSnorm4x8(), 698
unpackUnorm2x16(), 698
unpackUnorm4x8(), 698
user clipping, 238
usubBorrow(), 706
Utah teapot, 500

vector, 7, 214
vertex shader, 4, 35
vertex winding, 497
vertex, 11
vertex-array object, 17
vertex-attribute array, 26
vglLoadImage(), 277, 283–286
vglLoadTexture(), 286
vglUnloadImage(), 284, 286
viewing frustum, 210
viewing model, 206
viewing transformation, 207
viewport, 156
viewport, 13
 index, 555
 multiple, 550–559
 transform, 209
vmath::frustum(), 229
vmath::lookAt(), 229
vmath::ortho(), 231
vmath::rotate(), 227
vmath::scale(), 224
vmath::translate(), 220
volatile, 595
voxel, 307

WebGL
 InitShaders(), 666, 667
 onload(), 668
 setupWebGL(), 663

WGL
 wglCopyContext(), 847
 wglCreateContext(), 846, 867
 wglCreateContextAttribsARB(),
 846, 866, 867
 wglCreateLayerContext(), 846
 wglDescribeLayerPlane(), 846
 wglDestroyContext(), 847
 wglGetCurrentContext(), 847
 wglGetCurrentDC(), 847
 wglGetLayerPaletteEntries(), 848
 wglGetProcAddress(), 846, 847,
 867
 wglMakeCurrent(), 847
 wglRealizeLayerPalette(), 848
 wglSetLayerPaletteEntries(), 848
 wglShareLists(), 847
 wglSwapLayerBuffers(), 847
 wglUseFontBitmaps(), 848
 wglUseFontOutlines(), 848

winding, 91
window coordinates, 209
window system, 866, 867
windows.h, 846
wingdi.h, 846
wireframe, 165, 525
workgroup, 625, 630
world coordinates, 209
writeonly, 598

X Window System, 3

z precision, 237
z-buffer, 146
z-buffering, 13