
“As a graphics technologist and intense OpenGL ES developer, I can honestly say that if you buy only one book on OpenGL ES 3.0 programming, then this should be the book. Dan and Budirijanto have written a book clearly by programmers for programmers. It is simply required reading for anyone interested in OpenGL ES 3.0. It is informative, well organized, and comprehensive, but best of all practical. You will find yourself reaching for this book over and over again instead of the actual OpenGL ES specification during your programming sessions. I give it my highest recommendation.”

—Rick Tewell, Graphics Technology Architect, Freescale

“This book provides outstanding coverage of the latest version of OpenGL ES, with clear, comprehensive explanations and extensive examples. It belongs on the desk of anyone developing mobile applications.”

—Dave Astle, Graphics Tools Lead, Qualcomm Technologies, Inc., and Founder, GameDev.net

“The second edition of OpenGL® ES™ 3.0 Programming Guide provides a solid introduction to OpenGL ES 3.0 specifications, along with a wealth of practical information and examples to help any level of developer begin programming immediately. We’d recommend this guide as a primer on OpenGL ES 3.0 to any of the thousands of developers creating apps for the many mobile and embedded products using our PowerVR Rogue graphics.”

—Kristof Beets, Business Development, Imagination Technologies

“This is a solid OpenGL ES 3.0 reference book. It covers all aspects of the API and will help any developer get familiar with and understand the API, including specifically the new ES 3.0 functionality.”

—Jed Fisher, Managing Partner, 4D Pipeline

“This is a clear and thorough reference for OpenGL ES 3.0, and an excellent presentation of the concepts present in all modern OpenGL programming. This is the guide I’d want by my side when diving into embedded OpenGL.”

—Todd Furlong, President & Principal Engineer, Inv3rsion LLC
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.

Visit [informit.com/opengl](http://informit.com/opengl) for a complete list of available products.
## Contents

List of Figures ................................................................. xvii
List of Examples ............................................................... xxi
List of Tables .................................................................... xxv
Foreword ............................................................................ xxix
Preface .............................................................................. xxxi
  Intended Audience ............................................................ xxxi
  Organization of This Book ................................................... xxxii
  Example Code and Shaders ................................................ xxvii
  Errata ................................................................................ xxxvi
Acknowledgments ............................................................... xxxvii
About the Authors ............................................................. xxxix

### 1. Introduction to OpenGL ES 3.0 ..................................... 1

  OpenGL ES 3.0 ..................................................................... 3
  Vertex Shader ..................................................................... 4
  Primitive Assembly ............................................................ 7
  Rasterization ..................................................................... 7
  Fragment Shader ............................................................... 8
  Per-Fragment Operations ................................................... 9
  What's New in OpenGL ES 3.0 .......................................... 11
  Texturing .......................................................................... 11
  Shaders ............................................................................. 13
Geometry ........................................................................................................ 15
Buffer Objects ............................................................................................. 16
Framebuffer .................................................................................................. 17
OpenGL ES 3.0 and Backward Compatibility .............................................. 17
EGL .............................................................................................................. 19
Programming with OpenGL ES 3.0 ............................................................ 20
Libraries and Include Files ........................................................................ 20
EGL Command Syntax ................................................................................ 20
OpenGL ES Command Syntax .................................................................... 21
Error Handling ............................................................................................ 22
Basic State Management ............................................................................ 23
Further Reading .......................................................................................... 25

2. Hello Triangle: An OpenGL ES 3.0 Example ......................................... 27
   Code Framework .................................................................................... 28
   Where to Download the Examples ......................................................... 28
   Hello Triangle Example .......................................................................... 29
   Using the OpenGL ES 3.0 Framework ................................................. 34
   Creating a Simple Vertex and Fragment Shader ................................... 35
   Compiling and Loading the Shaders ...................................................... 36
   Creating a Program Object and Linking the Shaders ......................... 38
   Setting the Viewport and Clearing the Color Buffer ......................... 39
   Loading the Geometry and Drawing a Primitive ............................... 40
   Displaying the Back Buffer ................................................................... 41
   Summary ................................................................................................. 42

3. An Introduction to EGL ........................................................................ 43
   Communicating with the Windowing System ....................................... 44
   Checking for Errors ................................................................................ 45
   Initializing EGL ...................................................................................... 46
   Determining the Available Surface Configurations ............................. 46
   Querying EGLConfig Attributes ......................................................... 48
   Letting EGL Choose the Configuration .............................................. 51
   Creating an On-Screen Rendering Area: The EGL Window ................ 53
   Creating an Off-Screen Rendering Area: EGL Pbuffers ................. 56
   Creating a Rendering Context ............................................................... 60
## Contents

Making an EGLContext Current .......................................................... 62  
Putting All Our EGL Knowledge Together ............................................ 63  
Synchronizing Rendering ..................................................................... 66  
Summary ............................................................................................... 67  

4. **Shaders and Programs** ................................................................. 69  
Shaders and Programs........................................................................... 69  
  Creating and Compiling a Shader.................................................. 70  
  Creating and Linking a Program.................................................... 74  
Uniforms and Attributes.................................................................... 80  
  Getting and Setting Uniforms....................................................... 81  
  Uniform Buffer Objects............................................................... 87  
  Getting and Setting Attributes ................................................... 92  
Shader Compiler................................................................................... 93  
Program Binaries................................................................................ 94  
Summary............................................................................................... 95  

5. **OpenGL ES Shading Language** ....................................................... 97  
OpenGL ES Shading Language Basics................................................ 98  
Shader Version Specification ............................................................ 98  
Variables and Variable Types ............................................................. 99  
Variable Constructors ......................................................................... 100  
Vector and Matrix Components.......................................................... 101  
Constants............................................................................................ 102  
Structures ............................................................................................ 103  
Arrays .................................................................................................. 104  
Operators ............................................................................................ 104  
Functions ............................................................................................ 106  
Built-In Functions.............................................................................. 107  
Control Flow Statements.................................................................... 107  
Uniforms............................................................................................. 108  
Uniform Blocks................................................................................... 109  
Vertex and Fragment Shader Inputs/Outputs .................................... 111  
Interpolation Qualifiers ....................................................................... 114  
Preprocessor and Directives............................................................... 115  
Uniform and Interpolator Packing....................................................... 117
### 8. Vertex Shaders

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Shader Overview</td>
<td>188</td>
</tr>
<tr>
<td>Vertex Shader Built-In Variables</td>
<td>189</td>
</tr>
<tr>
<td>Precision Qualifiers</td>
<td>192</td>
</tr>
<tr>
<td>Number of Uniforms Limitations in a Vertex Shader</td>
<td>193</td>
</tr>
<tr>
<td>Vertex Shader Examples</td>
<td>196</td>
</tr>
<tr>
<td>Matrix Transformations</td>
<td>196</td>
</tr>
<tr>
<td>Lighting in a Vertex Shader</td>
<td>199</td>
</tr>
<tr>
<td>Generating Texture Coordinates</td>
<td>205</td>
</tr>
<tr>
<td>Vertex Skinning</td>
<td>207</td>
</tr>
<tr>
<td>Transform Feedback</td>
<td>211</td>
</tr>
<tr>
<td>Vertex Textures</td>
<td>214</td>
</tr>
<tr>
<td>OpenGL ES 1.1 Vertex Pipeline as an ES 3.0 Vertex Shader</td>
<td>215</td>
</tr>
<tr>
<td>Summary</td>
<td>223</td>
</tr>
</tbody>
</table>

### 9. Texturing

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Texturing Basics</td>
<td>226</td>
</tr>
<tr>
<td>2D Textures</td>
<td>226</td>
</tr>
<tr>
<td>Cubemap Textures</td>
<td>228</td>
</tr>
<tr>
<td>3D Textures</td>
<td>229</td>
</tr>
<tr>
<td>2D Texture Arrays</td>
<td>230</td>
</tr>
<tr>
<td>Texture Objects and Loading Textures</td>
<td>230</td>
</tr>
<tr>
<td>Texture Filtering and Mipmapping</td>
<td>237</td>
</tr>
<tr>
<td>Automatic Mipmap Generation</td>
<td>242</td>
</tr>
<tr>
<td>Texture Coordinate Wrapping</td>
<td>243</td>
</tr>
<tr>
<td>Texture Swizzles</td>
<td>244</td>
</tr>
<tr>
<td>Texture Level of Detail</td>
<td>245</td>
</tr>
<tr>
<td>Depth Texture Compare (Percentage Closest Filtering)</td>
<td>245</td>
</tr>
<tr>
<td>Texture Formats</td>
<td>246</td>
</tr>
<tr>
<td>Using Textures in a Shader</td>
<td>255</td>
</tr>
<tr>
<td>Example of Using a Cubemap Texture</td>
<td>258</td>
</tr>
<tr>
<td>Loading 3D Textures and 2D Texture Arrays</td>
<td>260</td>
</tr>
<tr>
<td>Compressed Textures</td>
<td>262</td>
</tr>
<tr>
<td>Texture Subimage Specification</td>
<td>266</td>
</tr>
<tr>
<td>Copying Texture Data from the Color Buffer</td>
<td>269</td>
</tr>
</tbody>
</table>
### Sampler Objects
- Immutable Textures
- Pixel Unpack Buffer Objects
- Summary

### 10. Fragment Shaders
- Fixed-Function Fragment Shaders
- Fragment Shader Overview
  - Built-In Special Variables
  - Built-In Constants
  - Precision Qualifiers
- Implementing Fixed-Function Techniques Using Shaders
  - Multitexturing
  - Fog
  - Alpha Test (Using Discard)
  - User Clip Planes
- Summary

### 11. Fragment Operations
- Buffers
  - Requesting Additional Buffers
  - Clearing Buffers
  - Using Masks to Control Writing to Framebuffers
- Fragment Tests and Operations
  - Using the Scissor Test
  - Stencil Buffer Testing
- Blending
- Dithering
- Multisampled Anti-Aliasing
  - Centroid Sampling
- Reading and Writing Pixels to the Framebuffer
  - Pixel Pack Buffer Objects
- Multiple Render Targets
- Summary
12. Framebuffer Objects .................................................................325
   Why Framebuffer Objects? .........................................................325
   Framebuffer and Renderbuffer Objects .....................................327
      Choosing a Renderbuffer Versus a Texture as a Framebuffer Attachment ..........................................................328
   Framebuffer Objects Versus EGL Surfaces .................................329
   Creating Framebuffer and Renderbuffer Objects .........................329
   Using Renderbuffer Objects ......................................................330
      Multisample Renderbuffers ....................................................333
   Renderbuffer Formats ..............................................................333
   Using Framebuffer Objects .......................................................335
      Attaching a Renderbuffer as a Framebuffer Attachment ..........337
      Attaching a 2D Texture as a Framebuffer Attachment .............338
      Attaching an Image of a 3D Texture as a Framebuffer Attachment ........................................................................339
   Checking for Framebuffer Completeness ..................................341
   Framebuffer Blits ......................................................................342
   Framebuffer Invalidation ..........................................................344
   Deleting Framebuffer and Renderbuffer Objects .......................346
      Deleting Renderbuffer Objects That Are Used as Framebuffer Attachments .........................................................347
      Reading Pixels and Framebuffer Objects .................................347
   Examples ..................................................................................348
   Performance Tips and Tricks ....................................................354
   Summary ..................................................................................355

13. Sync Objects and Fences ..........................................................357
   Flush and Finish ......................................................................357
   Why Use a Sync Object? ..........................................................358
   Creating and Deleting a Sync Object .........................................358
   Waiting for and Signaling a Sync Object ....................................359
   Example ..................................................................................360
   Summary ..................................................................................361
14. Advanced Programming with OpenGL ES 3.0 ........................................... 363

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Per-Fragment Lighting</td>
<td>363</td>
</tr>
<tr>
<td>Lighting with a Normal Map</td>
<td>364</td>
</tr>
<tr>
<td>Lighting Shaders</td>
<td>366</td>
</tr>
<tr>
<td>Lighting Equations</td>
<td>369</td>
</tr>
<tr>
<td>Environment Mapping</td>
<td>370</td>
</tr>
<tr>
<td>Particle System with Point Sprites</td>
<td>374</td>
</tr>
<tr>
<td>Particle System Setup</td>
<td>374</td>
</tr>
<tr>
<td>Particle System Vertex Shader</td>
<td>375</td>
</tr>
<tr>
<td>Particle System Fragment Shader</td>
<td>377</td>
</tr>
<tr>
<td>Particle System Using Transform Feedback</td>
<td>380</td>
</tr>
<tr>
<td>Particle System Rendering Algorithm</td>
<td>381</td>
</tr>
<tr>
<td>Particle Emission with Transform Feedback</td>
<td>381</td>
</tr>
<tr>
<td>Rendering the Particles</td>
<td>385</td>
</tr>
<tr>
<td>Image Postprocessing</td>
<td>387</td>
</tr>
<tr>
<td>Render-to-Texture Setup</td>
<td>387</td>
</tr>
<tr>
<td>Blur Fragment Shader</td>
<td>388</td>
</tr>
<tr>
<td>Projective Texturing</td>
<td>390</td>
</tr>
<tr>
<td>Projective Texturing Basics</td>
<td>391</td>
</tr>
<tr>
<td>Matrices for Projective Texturing</td>
<td>392</td>
</tr>
<tr>
<td>Projective Spotlight Shaders</td>
<td>394</td>
</tr>
<tr>
<td>Noise Using a 3D Texture</td>
<td>397</td>
</tr>
<tr>
<td>Generating Noise</td>
<td>397</td>
</tr>
<tr>
<td>Using Noise</td>
<td>402</td>
</tr>
<tr>
<td>Procedural Texturing</td>
<td>404</td>
</tr>
<tr>
<td>A Procedural Texture Example</td>
<td>405</td>
</tr>
<tr>
<td>Anti-Aliasing of Procedural Textures</td>
<td>407</td>
</tr>
<tr>
<td>Further Reading on Procedural Textures</td>
<td>410</td>
</tr>
<tr>
<td>Rendering Terrain with Vertex Texture Fetch</td>
<td>410</td>
</tr>
<tr>
<td>Generating a Square Terrain Grid</td>
<td>411</td>
</tr>
<tr>
<td>Computing Vertex Normal and Fetching Height Value</td>
<td>412</td>
</tr>
<tr>
<td>in Vertex Shader</td>
<td></td>
</tr>
<tr>
<td>Further Reading on Large Terrain Rendering</td>
<td>413</td>
</tr>
<tr>
<td>Shadows Using a Depth Texture</td>
<td>414</td>
</tr>
<tr>
<td>Rendering from the Light Position Into a Depth Texture</td>
<td>415</td>
</tr>
<tr>
<td>Rendering from the Eye Position with the Depth Texture</td>
<td>418</td>
</tr>
<tr>
<td>Summary</td>
<td>420</td>
</tr>
<tr>
<td>15. State Queries</td>
<td>421</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----</td>
</tr>
<tr>
<td>OpenGL ES 3.0 Implementation String Queries</td>
<td>421</td>
</tr>
<tr>
<td>Querying Implementation-Dependent Limits</td>
<td>423</td>
</tr>
<tr>
<td>Querying OpenGL ES State</td>
<td>429</td>
</tr>
<tr>
<td>Hints</td>
<td>435</td>
</tr>
<tr>
<td>Entity Name Queries</td>
<td>436</td>
</tr>
<tr>
<td>Nonprogrammable Operations Control and Queries</td>
<td>436</td>
</tr>
<tr>
<td>Shader and Program State Queries</td>
<td>438</td>
</tr>
<tr>
<td>Vertex Attribute Queries</td>
<td>440</td>
</tr>
<tr>
<td>Texture State Queries</td>
<td>441</td>
</tr>
<tr>
<td>Sampler Queries</td>
<td>442</td>
</tr>
<tr>
<td>Asynchronous Object Queries</td>
<td>442</td>
</tr>
<tr>
<td>Sync Object Queries</td>
<td>443</td>
</tr>
<tr>
<td>Vertex Buffer Queries</td>
<td>444</td>
</tr>
<tr>
<td>Renderbuffer and Framebuffer State Queries</td>
<td>445</td>
</tr>
<tr>
<td>Summary</td>
<td>446</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>16. OpenGL ES Platforms</th>
<th>447</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building for Microsoft Windows with Visual Studio</td>
<td>447</td>
</tr>
<tr>
<td>Building for Ubuntu Linux</td>
<td>449</td>
</tr>
<tr>
<td>Building for Android 4.3+ NDK (C++)</td>
<td>450</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>451</td>
</tr>
<tr>
<td>Building the Example Code with Android NDK</td>
<td>452</td>
</tr>
<tr>
<td>Building for Android 4.3+ SDK (Java)</td>
<td>452</td>
</tr>
<tr>
<td>Building for iOS 7</td>
<td>453</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>453</td>
</tr>
<tr>
<td>Building the Example Code with Xcode 5</td>
<td>453</td>
</tr>
<tr>
<td>Summary</td>
<td>455</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A. GL_HALF_FLOAT</th>
<th>457</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-Bit Floating-Point Number</td>
<td>458</td>
</tr>
<tr>
<td>Converting a Float to a Half-Float</td>
<td>459</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B. Built-In Functions</th>
<th>463</th>
</tr>
</thead>
<tbody>
<tr>
<td>Angle and Trigonometry Functions</td>
<td>465</td>
</tr>
<tr>
<td>Exponential Functions</td>
<td>466</td>
</tr>
<tr>
<td>Common Functions</td>
<td>467</td>
</tr>
</tbody>
</table>
Floating-Point Pack and Unpack Functions................................. 471
Geometric Functions ................................................................ 472
Matrix Functions ..................................................................... 474
Vector Relational Functions .................................................... 475
Texture Lookup Functions ....................................................... 476
Fragment Processing Functions ................................................ 483

C. ES Framework API.................................................................. 485

Framework Core Functions .................................................... 485
Transformation Functions ....................................................... 490

Index....................................................................................... 495
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1-1</td>
<td>OpenGL ES 3.0 Graphics Pipeline</td>
<td>4</td>
</tr>
<tr>
<td>Figure 1-2</td>
<td>OpenGL ES 3.0 Vertex Shader</td>
<td>5</td>
</tr>
<tr>
<td>Figure 1-3</td>
<td>OpenGL ES 3.0 Rasterization Stage</td>
<td>7</td>
</tr>
<tr>
<td>Figure 1-4</td>
<td>OpenGL ES 3.0 Fragment Shader</td>
<td>8</td>
</tr>
<tr>
<td>Figure 1-5</td>
<td>OpenGL ES 3.0 Per-Fragment Operations</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2-1</td>
<td>Hello Triangle Example</td>
<td>33</td>
</tr>
<tr>
<td>Figure 5-1</td>
<td>Z Fighting Artifacts Due to Not Using Invariance</td>
<td>121</td>
</tr>
<tr>
<td>Figure 5-2</td>
<td>Z Fighting Avoided Using Invariance</td>
<td>122</td>
</tr>
<tr>
<td>Figure 6-1</td>
<td>Triangle with a Constant Color Vertex and Per-Vertex Position Attributes</td>
<td>125</td>
</tr>
<tr>
<td>Figure 6-2</td>
<td>Position, Normal, and Two Texture Coordinates Stored as an Array</td>
<td>128</td>
</tr>
<tr>
<td>Figure 6-3</td>
<td>Selecting Constant or Vertex Array Vertex Attribute</td>
<td>133</td>
</tr>
<tr>
<td>Figure 6-4</td>
<td>Specifying and Binding Vertex Attributes for Drawing One or More Primitives</td>
<td>138</td>
</tr>
<tr>
<td>Figure 7-1</td>
<td>Triangle Primitive Types</td>
<td>162</td>
</tr>
<tr>
<td>Figure 7-2</td>
<td>Line Primitive Types</td>
<td>163</td>
</tr>
<tr>
<td>Figure 7-3</td>
<td>gl_PointCoord Values</td>
<td>165</td>
</tr>
<tr>
<td>Figure 7-4</td>
<td>Cube</td>
<td>167</td>
</tr>
<tr>
<td>Figure 7-5</td>
<td>Connecting Triangle Strips</td>
<td>173</td>
</tr>
<tr>
<td>Figure 7-6</td>
<td>OpenGL ES Primitive Assembly Stage</td>
<td>175</td>
</tr>
<tr>
<td>Figure 7-7</td>
<td>Coordinate Systems</td>
<td>175</td>
</tr>
<tr>
<td>Figure 7-8</td>
<td>Viewing Volume</td>
<td>176</td>
</tr>
<tr>
<td>Figure 7-9</td>
<td>OpenGL ES Rasterization Stage</td>
<td>179</td>
</tr>
<tr>
<td>Figure 7-10</td>
<td>Clockwise and Counterclockwise Triangles</td>
<td>180</td>
</tr>
<tr>
<td>Figure 7-11</td>
<td>Polygon Offset</td>
<td>182</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>8-1</td>
<td>OpenGL ES 3.0 Programmable Pipeline</td>
<td>188</td>
</tr>
<tr>
<td>8-2</td>
<td>OpenGL ES 3.0 Vertex Shader</td>
<td>189</td>
</tr>
<tr>
<td>8-3</td>
<td>Geometric Factors in Computing Lighting Equation for a Directional Light</td>
<td>199</td>
</tr>
<tr>
<td>8-4</td>
<td>Geometric Factors in Computing Lighting Equation for a Spotlight</td>
<td>202</td>
</tr>
<tr>
<td>9-1</td>
<td>2D Texture Coordinates</td>
<td>227</td>
</tr>
<tr>
<td>9-2</td>
<td>3D Texture Coordinate for Cubemap</td>
<td>228</td>
</tr>
<tr>
<td>9-3</td>
<td>3D Texture</td>
<td>229</td>
</tr>
<tr>
<td>9-4</td>
<td>MipMap2D: Nearest Versus Trilinear Filtering</td>
<td>241</td>
</tr>
<tr>
<td>9-5</td>
<td>GL_REPEAT, GL_CLAMP_TO_EDGE, and GL_MIRRORED_REPEAT Modes</td>
<td>243</td>
</tr>
<tr>
<td>10-1</td>
<td>OpenGL ES 3.0 Programmable Pipeline</td>
<td>280</td>
</tr>
<tr>
<td>10-2</td>
<td>OpenGL ES 3.0 Fragment Shader</td>
<td>283</td>
</tr>
<tr>
<td>10-3</td>
<td>Multitextured Quad</td>
<td>287</td>
</tr>
<tr>
<td>10-4</td>
<td>Linear Fog on Torus in PVRShaman</td>
<td>289</td>
</tr>
<tr>
<td>10-5</td>
<td>Alpha Test Using Discard</td>
<td>292</td>
</tr>
<tr>
<td>10-6</td>
<td>User Clip Plane Example</td>
<td>294</td>
</tr>
<tr>
<td>11-1</td>
<td>The Post-Shader Fragment Pipeline</td>
<td>297</td>
</tr>
<tr>
<td>12-1</td>
<td>Framebuffer Objects, Renderbuffer Objects, and Textures</td>
<td>328</td>
</tr>
<tr>
<td>12-2</td>
<td>Render to Color Texture</td>
<td>350</td>
</tr>
<tr>
<td>12-3</td>
<td>Render to Depth Texture</td>
<td>353</td>
</tr>
<tr>
<td>14-1</td>
<td>Per-Fragment Lighting Example</td>
<td>364</td>
</tr>
<tr>
<td>14-2</td>
<td>Environment Mapping Example</td>
<td>370</td>
</tr>
<tr>
<td>14-3</td>
<td>Particle System Sample</td>
<td>374</td>
</tr>
<tr>
<td>14-4</td>
<td>Particle System with Transform Feedback</td>
<td>380</td>
</tr>
<tr>
<td>14-5</td>
<td>Image Postprocessing Example</td>
<td>387</td>
</tr>
<tr>
<td>14-6</td>
<td>Light Bloom Effect</td>
<td>389</td>
</tr>
<tr>
<td>14-7</td>
<td>Light Bloom Stages</td>
<td>390</td>
</tr>
<tr>
<td>14-8</td>
<td>Projective Spotlight Example</td>
<td>391</td>
</tr>
<tr>
<td>14-9</td>
<td>2D Texture Projected onto Object</td>
<td>392</td>
</tr>
<tr>
<td>14-10</td>
<td>Fog Distorted by 3D Noise Texture</td>
<td>397</td>
</tr>
<tr>
<td>14-11</td>
<td>2D Slice of Gradient Noise</td>
<td>402</td>
</tr>
<tr>
<td>14-12</td>
<td>Checkerboard Procedural Texture</td>
<td>407</td>
</tr>
</tbody>
</table>
Figure 14-13  Anti-Aliased Checkerboard Procedural Texture ...............409
Figure 14-14  Terrain Rendered with Vertex Texture Fetch ...............411
Figure 14-15  Shadow Rendering with a Depth Texture and 6 × 6 PCF........................................................................414
Figure 16-1   Building Samples with CMake GUI on Windows ............448
Figure 16-2   VertexArrayObjects Sample in Xcode Running on iOS 7 Simulator.................................................................454
Figure A-1    A 16-Bit Floating-Point Number ..................................458
List of Examples

<p>| Example 1-1 | A Vertex Shader Example | Example 1-2 | A Fragment Shader Example | Example 2-1 | Hello_Triangle.c Example | Example 3-1 | Initializing EGL | Example 3-2 | Specifying EGL Attributes | Example 3-3 | Querying EGL Surface Configurations | Example 3-4 | Creating an EGL Window Surface | Example 3-5 | Creating an EGL Pixel Buffer | Example 3-6 | Creating an EGL Context | Example 3-7 | A Complete Routine for Creating an EGL Window | Example 3-8 | Creating a Window Using the esUtil Library | Example 4-1 | Loading a Shader | Example 4-2 | Create, Attach Shaders to, and Link a Program | Example 4-3 | Querying for Active Uniforms | Example 5-1 | Sample Vertex Shader | Example 5-2 | Vertex and Fragment Shaders with Matching Output/Input Declarations | Example 6-1 | Array of Structures | Example 6-2 | Structure of Arrays | Example 6-3 | Using Constant and Vertex Array Attributes | Example 6-4 | Creating and Binding Vertex Buffer Objects | Example 6-5 | Drawing with and without Vertex Buffer Objects | Example 6-6 | Drawing with a Buffer Object per Attribute | Example 6-7 | Drawing with a Vertex Array Object | Example 6-8 | Mapping a Buffer Object for Writing | Example 8-1 | Vertex Shader with Matrix Transform for the Position |</p>
<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-2</td>
<td>Directional Light</td>
<td>200</td>
</tr>
<tr>
<td>8-3</td>
<td>Spotlight</td>
<td>203</td>
</tr>
<tr>
<td>8-4</td>
<td>Sphere Map Texture Coordinate Generation</td>
<td>206</td>
</tr>
<tr>
<td>8-5</td>
<td>Cubemap Texture Coordinate Generation</td>
<td>206</td>
</tr>
<tr>
<td>8-6</td>
<td>Vertex Skinning Shader with No Check of Whether Matrix Weight = 0</td>
<td>208</td>
</tr>
<tr>
<td>8-7</td>
<td>Vertex Skinning Shader with Checks of Whether Matrix Weight = 0</td>
<td>210</td>
</tr>
<tr>
<td>8-8</td>
<td>Displacement Mapping Vertex Shader</td>
<td>214</td>
</tr>
<tr>
<td>8-9</td>
<td>OpenGL ES 1.1 Fixed-Function Vertex Pipeline</td>
<td>216</td>
</tr>
<tr>
<td>9-1</td>
<td>Generating a Texture Object, Binding It, and Loading Image Data</td>
<td>234</td>
</tr>
<tr>
<td>9-2</td>
<td>Loading a 2D Mipmap Chain</td>
<td>238</td>
</tr>
<tr>
<td>9-3</td>
<td>Vertex and Fragment Shaders for Performing 2D Texturing</td>
<td>255</td>
</tr>
<tr>
<td>9-4</td>
<td>Loading a Cubemap Texture</td>
<td>258</td>
</tr>
<tr>
<td>9-5</td>
<td>Vertex and Fragment Shader Pair for Cubemap Texturing</td>
<td>259</td>
</tr>
<tr>
<td>10-1</td>
<td>Multitexture Fragment Shader</td>
<td>287</td>
</tr>
<tr>
<td>10-2</td>
<td>Vertex Shader for Computing Distance to Eye</td>
<td>289</td>
</tr>
<tr>
<td>10-3</td>
<td>Fragment Shader for Rendering Linear Fog</td>
<td>290</td>
</tr>
<tr>
<td>10-4</td>
<td>Fragment Shader for Alpha Test Using Discard</td>
<td>292</td>
</tr>
<tr>
<td>10-5</td>
<td>User Clip Plane Vertex Shader</td>
<td>294</td>
</tr>
<tr>
<td>10-6</td>
<td>User Clip Plane Fragment Shader</td>
<td>295</td>
</tr>
<tr>
<td>11-1</td>
<td>Setting up Multiple Render Targets</td>
<td>322</td>
</tr>
<tr>
<td>11-2</td>
<td>Fragment Shader with Multiple Render Targets</td>
<td>324</td>
</tr>
<tr>
<td>12-1</td>
<td>Copying Pixels Using Framebuffer Blits</td>
<td>343</td>
</tr>
<tr>
<td>12-2</td>
<td>Render to Texture</td>
<td>348</td>
</tr>
<tr>
<td>12-3</td>
<td>Render to Depth Texture</td>
<td>351</td>
</tr>
<tr>
<td>13-1</td>
<td>Inserting a Fence Command and Waiting for Its Result in Transform Feedback Example</td>
<td>361</td>
</tr>
<tr>
<td>14-1</td>
<td>Per-Fragment Lighting Vertex Shader</td>
<td>366</td>
</tr>
<tr>
<td>14-2</td>
<td>Per-Fragment Lighting Fragment Shader</td>
<td>367</td>
</tr>
<tr>
<td>14-3</td>
<td>Environment Mapping Vertex Shader</td>
<td>371</td>
</tr>
<tr>
<td>14-4</td>
<td>Environment Mapping Fragment Shader</td>
<td>372</td>
</tr>
<tr>
<td>14-5</td>
<td>Particle System Vertex Shader</td>
<td>375</td>
</tr>
<tr>
<td>Example 14-6</td>
<td>Update Function for Particle System Sample</td>
<td>376</td>
</tr>
<tr>
<td>Example 14-7</td>
<td>Particle System Fragment Shader</td>
<td>377</td>
</tr>
<tr>
<td>Example 14-8</td>
<td>Draw Function for Particle System Sample</td>
<td>378</td>
</tr>
<tr>
<td>Example 14-9</td>
<td>Particle Emission Vertex Shader</td>
<td>382</td>
</tr>
<tr>
<td>Example 14-10</td>
<td>Emit Particles with Transform Feedback</td>
<td>384</td>
</tr>
<tr>
<td>Example 14-11</td>
<td>Particle Rendering Vertex Shader</td>
<td>386</td>
</tr>
<tr>
<td>Example 14-12</td>
<td>Blur Fragment Shader</td>
<td>388</td>
</tr>
<tr>
<td>Example 14-13</td>
<td>Projective Texturing Vertex Shader</td>
<td>394</td>
</tr>
<tr>
<td>Example 14-14</td>
<td>Projective Texturing Fragment Shader</td>
<td>396</td>
</tr>
<tr>
<td>Example 14-15</td>
<td>Generating Gradient Vectors</td>
<td>398</td>
</tr>
<tr>
<td>Example 14-16</td>
<td>3D Noise</td>
<td>400</td>
</tr>
<tr>
<td>Example 14-17</td>
<td>Noise-Distorted Fog Fragment Shader</td>
<td>402</td>
</tr>
<tr>
<td>Example 14-18</td>
<td>Checker Vertex Shader</td>
<td>405</td>
</tr>
<tr>
<td>Example 14-19</td>
<td>Checker Fragment Shader with Conditional Checks</td>
<td>406</td>
</tr>
<tr>
<td>Example 14-20</td>
<td>Checker Fragment Shader without Conditional Checks</td>
<td>406</td>
</tr>
<tr>
<td>Example 14-21</td>
<td>Anti-Aliased Checker Fragment Shader</td>
<td>407</td>
</tr>
<tr>
<td>Example 14-22</td>
<td>Terrain Rendering Flat Grid Generation</td>
<td>411</td>
</tr>
<tr>
<td>Example 14-23</td>
<td>Terrain Rendering Vertex Shader</td>
<td>412</td>
</tr>
<tr>
<td>Example 14-24</td>
<td>Set up a MVP Matrix from the Light Position</td>
<td>415</td>
</tr>
<tr>
<td>Example 14-25</td>
<td>Create a Depth Texture and Attach It to a Framebuffer Object</td>
<td>416</td>
</tr>
<tr>
<td>Example 14-26</td>
<td>Rendering to Depth Texture Shaders</td>
<td>417</td>
</tr>
<tr>
<td>Example 14-27</td>
<td>Rendering from the Eye Position Shaders</td>
<td>418</td>
</tr>
</tbody>
</table>
This page intentionally left blank
Table 1-1  EGL Data Types .................................................. 21
Table 1-2  OpenGL ES Command Suffixes and Argument Data Types ...................... 22
Table 1-3  OpenGL ES Basic Error Codes .................................. 23
Table 3-1  EGLConfig Attributes ........................................ 49
Table 3-2  Attributes for Window Creation Using eglCreateWindowSurface ................. 54
Table 3-3  Possible Errors When eglCreateWindowSurface Fails ....................... 55
Table 3-4  EGL Pixel Buffer Attributes .................................. 57
Table 3-5  Possible Errors When eglCreatePbufferSurface Fails .......................... 58
Table 3-6  Attributes for Context Creation Using eglCreateContext ...................... 61
Table 5-1  Data Types in the OpenGL ES Shading Language ............................... 99
Table 5-2  OpenGL ES Shading Language Operators ................................... 104
Table 5-3  OpenGL ES Shading Language Qualifiers .................................. 106
Table 5-4  Uniform Block Layout Qualifiers .................................... 111
Table 5-5  Extension Behaviors ............................................. 116
Table 5-6  Uniform Storage without Packing .................................. 118
Table 5-7  Uniform Storage with Packing .................................... 119
Table 6-1  Data Conversions .................................................. 132
Table 6-2  Buffer Usage ....................................................... 143
Table 7-1  Provoking Vertex Selection for the \(i\)th Primitive Instance Where Vertices Are Numbered from 1 to \(n\), and \(n\) Is the Number of Vertices Drawn ........................................ 169
Table 8-1  Transform Feedback Primitive Mode and Allowed Draw Mode .................. 213
Table 9-1  Texture Base Formats ............................................. 227
Table 9-2  Pixel Storage Options ..............................................................236
Table 9-3  Texture Wrap Modes..............................................................243
Table 9-4  Valid Unsized Internal Format Combinations
for glTexImage2D..............................................................247
Table 9-5  Normalized Sized Internal Format Combinations
for glTexImage2D..............................................................248
Table 9-6  Valid Sized Floating-Point Internal Format
Combinations for glTexImage2D ..............................................249
Table 9-7  Valid Sized Internal Integer Texture Format
Combinations for glTexImage2D ..............................................251
Table 9-8  Valid Shared Exponent Sized Internal Format
Combinations for glTexImage2D ..............................................253
Table 9-9  Valid sRGB Sized Internal Format Combinations
for glTexImage2D..............................................................254
Table 9-10 Valid Depth Sized Internal Format Combinations
for glTexImage2D..............................................................255
Table 9-11 Mapping of Texture Formats to Colors ......................................257
Table 9-12 Standard Texture Compression Formats ....................................264
Table 9-13 Valid Format Conversions for glCopyTex*Image* ...............273
Table 10-1 OpenGL ES 1.1 RGB Combine Functions ..................................281
Table 11-1 Fragment Test Enable Tokens................................................304
Table 11-2 Stencil Operations..................................................................306
Table 11-3 Blending Functions..................................................................312
Table 12-1 Renderbuffer Formats for Color-Renderable Buffer .............333
Table 12-2 Renderbuffer Formats for Depth-Renderable
and Stencil-Renderable Buffer.................................................335
Table 15-1 Implementation-Dependent State Queries...............................423
Table 15-2 Application-Modifiable OpenGL ES State Queries .............429
Table 15-3 OpenGL ES 3.0 Capabilities Controlled by
glEnable and glDisable..........................................................437
Table B-1 Angle and Trigonometry Functions........................................465
Table B-2 Exponential Functions..........................................................466
Table B-3 Common Functions.............................................................467
Table B-4 Floating-Point Pack and Unpack Functions .........................471
<table>
<thead>
<tr>
<th>Table B-5</th>
<th>Geometric Functions .................................................................473</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table B-6</td>
<td>Matrix Functions ......................................................................474</td>
</tr>
<tr>
<td>Table B-7</td>
<td>Vector Relational Functions .....................................................475</td>
</tr>
<tr>
<td>Table B-8</td>
<td>Supported Combinations of Sampler and Internal Texture Formats ..........476</td>
</tr>
<tr>
<td>Table B-9</td>
<td>Texture Lookup Functions ..........................................................478</td>
</tr>
<tr>
<td>Table B-10</td>
<td>Fragment Processing Functions ......................................................484</td>
</tr>
</tbody>
</table>
Foreword

Five years have passed since the OpenGL ES 2.0 version of this reference book helped alert developers everywhere that programmable 3D graphics on mobile and embedded systems had not just arrived, but was here to stay.

Five years later, more than 1 billion people around the world use OpenGL ES every day to interact with their computing devices, for both information and entertainment. Nearly every pixel on nearly every smartphone screen has been generated, manipulated, or composited by this ubiquitous graphics API.

Now, OpenGL ES 3.0 has been developed by Khronos Group and is shipping on the latest mobile devices, continuing the steady flow of advanced graphics features into the hands of consumers everywhere—features that were first developed and proven on high-end systems shipping with desktop OpenGL.

In fact, OpenGL is now easily the most widely deployed family of 3D APIs, with desktop OpenGL and OpenGL ES being joined by WebGL to bring the power of OpenGL ES to web content everywhere. OpenGL ES 3.0 will be instrumental in powering the evolution of WebGL, enabling HTML5 developers to tap directly into the power of the latest GPUs from the first truly portable 3D applications.

OpenGL ES 3.0 not only places more graphics capabilities into the hands of developers across a huge range of devices and platforms, but also enables faster, more power-efficient 3D applications that are easier to write, port, and maintain—and this book will show you how.
There has never been a more fascinating and rewarding time to be a 3D developer. My thanks and congratulations go to the authors for continuing to be a vital part of the evolving story of OpenGL ES, and for working hard to produce this book that helps ensure developers everywhere can better understand and leverage the full power of OpenGL ES 3.0.

—Neil Trevett
President, Khronos Group
Vice President Mobile Ecosystem, NVIDIA
OpenGL ES 3.0 is a software interface for rendering sophisticated 3D graphics on handheld and embedded devices. OpenGL ES is the primary graphics library for handheld and embedded devices with programmable 3D hardware including cell phones, personal digital assistants (PDAs), consoles, appliances, vehicles, and avionics. This book details the entire OpenGL ES 3.0 application programming interface (API) and pipeline, including detailed examples, to provide a guide for developing a wide range of high-performance 3D applications for handheld devices.

Intended Audience

This book is intended for programmers who are interested in learning OpenGL ES 3.0. We expect the reader to have a solid grounding in computer graphics. In the text we explain many of the relevant graphics concepts as they relate to various parts of OpenGL ES 3.0, but we expect the reader to understand basic 3D concepts. The code examples in the book are all written in C. We assume that the reader is familiar with C or C++ and cover language topics only where they are relevant to OpenGL ES 3.0.

The reader will learn about setting up and programming every aspect of the graphics pipeline. The book details how to write vertex and fragment shaders and how to implement advanced rendering techniques such as per-pixel lighting and particle systems. In addition, it provides performance tips and tricks for efficient use of the API and hardware. After finishing the book, the reader will be ready to write OpenGL ES 3.0 applications that fully harness the programmable power of embedded graphics hardware.
Organization of This Book

This book is organized to cover the API in a sequential fashion, building up your knowledge of OpenGL ES 3.0 as we go.

Chapter 1—Introduction to OpenGL ES 3.0

Chapter 1 introduces OpenGL ES and provides an overview of the OpenGL ES 3.0 graphics pipeline. We discuss the philosophies and constraints that went into the design of OpenGL ES 3.0. Finally, the chapter covers some general conventions and types used in OpenGL ES 3.0.

Chapter 2—Hello Triangle: An OpenGL ES 3.0 Example

Chapter 2 walks through a simple OpenGL ES 3.0 example program that draws a triangle. Our purpose here is to show what an OpenGL ES 3.0 program looks like, introduce the reader to some API concepts, and describe how to build and run an example OpenGL ES 3.0 program.

Chapter 3—An Introduction to EGL

Chapter 3 presents EGL, the API for creating surfaces and rendering contexts for OpenGL ES 3.0. We describe how to communicate with the native windowing system, choose a configuration, and create EGL rendering contexts and surfaces. We teach you enough EGL so that you can do everything you will need to do to get up and rendering with OpenGL ES 3.0.

Chapter 4—Shaders and Programs

Shader objects and program objects form the most fundamental objects in OpenGL ES 3.0. In Chapter 4, we describe how to create a shader object, compile a shader, and check for compile errors. The chapter also explains how to create a program object, attach shader objects to it, and link a final program object. We discuss how to query the program object for information and how to load uniforms. In addition, you will learn about the difference between source shaders and program binaries and how to use each.
Chapter 5—OpenGL ES Shading Language

Chapter 5 covers the shading language basics needed for writing shaders. These shading language basics include variables and types, constructors, structures, arrays, uniforms, uniform blocks, and input/output variables. This chapter also describes some more nuanced parts of the shading language, such as precision qualifiers and invariance.

Chapter 6—Vertex Attributes, Vertex Arrays, and Buffer Objects

Starting with Chapter 6 (and ending with Chapter 11), we begin our walk through the pipeline to teach you how to set up and program each part of the graphics pipeline. This journey begins with a description of how geometry is input into the graphics pipeline, and includes discussion of vertex attributes, vertex arrays, and buffer objects.

Chapter 7—Primitive Assembly and Rasterization

After discussing how geometry is input into the pipeline in the previous chapter, in Chapter 7 we consider how that geometry is assembled into primitives. All of the primitive types available in OpenGL ES 3.0, including point sprites, lines, triangles, triangle strips, and triangle fans, are covered. In addition, we describe how coordinate transformations are performed on vertices and introduce the rasterization stage of the OpenGL ES 3.0 pipeline.

Chapter 8—Vertex Shaders

The next portion of the pipeline that is covered is the vertex shader. Chapter 8 provides an overview of how vertex shaders fit into the pipeline and the special variables available to vertex shaders in the OpenGL ES Shading Language. Several examples of vertex shaders, including computation of per-vertex lighting and skinning, are covered. We also give examples of how the OpenGL ES 1.0 (and 1.1) fixed-function pipeline can be implemented using vertex shaders.

Chapter 9—Texturing

Chapter 9 begins the introduction to fragment shaders by describing all of the texturing functionality available in OpenGL ES 3.0. This chapter provides details on how to create textures, how to load them with data,
and how to render with them. It describes texture wrap modes, texture filtering, texture formats, compressed textures, sampler objects, immutable textures, pixel unpack buffer objects, and mipmapping. This chapter covers all of the texture types supported in OpenGL ES 3.0: 2D textures, cubemaps, 2D texture arrays, and 3D textures.

**Chapter 10—Fragment Shaders**

Chapter 9 focused on how to use textures in a fragment shader; Chapter 10 covers the rest of what you need to know to write fragment shaders. We give an overview of fragment shaders and all of the special built-in variables available to them. We also demonstrate how to implement all of the fixed-function techniques that were available in OpenGL ES 1.1 using fragment shaders. Examples of multitexturing, fog, alpha test, and user clip planes are all implemented in fragment shaders.

**Chapter 11—Fragment Operations**

Chapter 11 discusses the operations that can be applied either to the entire framebuffer, or to individual fragments after the execution of the fragment shader in the OpenGL ES 3.0 fragment pipeline. These operations include the scissor test, stencil test, depth test, multisampling, blending, and dithering. This chapter covers the final phase in the OpenGL ES 3.0 graphics pipeline.

**Chapter 12—Framebuffer Objects**

Chapter 12 discusses the use of framebuffer objects for rendering to off-screen surfaces. Framebuffer objects have several uses, the most common of which is for rendering to a texture. This chapter provides a complete overview of the framebuffer object portion of the API. Understanding framebuffer objects is critical for implementing many advanced effects such as reflections, shadow maps, and postprocessing.

**Chapter 13—Sync Objects and Fences**

Chapter 13 provides an overview of sync objects and fences, which are efficient primitives for synchronizing within the host application and GPU execution in OpenGL ES 3.0. We discuss how to use sync objects and fences and conclude with an example.
Chapter 14—Advanced Programming with OpenGL ES 3.0

Chapter 14 is the capstone chapter, tying together many of the topics presented throughout the book. We have selected a sampling of advanced rendering techniques and show examples that demonstrate how to implement these features. This chapter includes rendering techniques such as per-pixel lighting using normal maps, environment mapping, particle systems, image postprocessing, procedural textures, shadow mapping, terrain rendering and projective texturing.

Chapter 15—State Queries

A large number of state queries are available in OpenGL ES 3.0. For just about everything you set, there is a corresponding way to get the current value. Chapter 15 is provided as a reference for the various state queries available in OpenGL ES 3.0.

Chapter 16—OpenGL ES Platforms

In the final chapter, we move away from the details of the API to talk about how to build the OpenGL ES sample code in this book for iOS7, Android 4.3 NDK, Android 4.3 SDK, Windows, and Linux. This chapter is intended to serve as a reference to get you up and running with the book sample code on the OpenGL ES 3.0 platform of your choosing.

Appendix A—GL_HALF_FLOAT_OES

Appendix A details the half-float format and provides a reference for how to convert from IEEE floating-point values into half-floats (and back).

Appendix B—Built-In Functions

Appendix B provides a reference for all of the built-in functions available in the OpenGL ES Shading Language.

Appendix C—ES Framework API

Appendix C provides a reference for the utility framework we developed for the book and describes what each function does.
OpenGL ES 3.0 Reference Card

Included as a color insert in the middle of the book is the OpenGL ES 3.0 Reference Card, copyrighted by Khronos and reprinted with permission. This reference contains a complete list of all of the functions in OpenGL ES 3.0, along with all of the types, operators, qualifiers, built-ins, and functions in the OpenGL ES Shading Language.

Example Code and Shaders

This book is filled with example programs and shaders. You can download the examples from the book’s website at opengles-book.com, which provides a link to the github.com site hosting the book code. As of this writing, the example programs have been built and tested on iOS7, Android 4.3 NDK, Android 4.3 SDK, Windows (OpenGL ES 3.0 Emulation), and Ubuntu Linux. Several of the advanced shader examples in the book are implemented in PVRShaman, a shader development tool from PowerVR available for Windows, Mac OS X, and Linux. The book’s website (opengles-book.com) provides links through which to download any of the required tools.

Errata

If you find something in the book that you believe is in error, please send us a note at errors@opengles-book.com. The list of errata for the book can be found on the book’s website: opengles-book.com.
I want to thank Affie Munshi and Dave Shreiner for their enormous contributions to the first edition of this book. I am extremely grateful to have Budi Purnomo join me to update the book for OpenGL ES 3.0. I would also like to thank the many colleagues with whom I have worked over the years, who have helped in my education on computer graphics, OpenGL, and OpenGL ES. There are too many people to list all of them, but special thanks go to Shawn Leaf, Bill Licea-Kane, Maurice Ribble, Benj Lipchak, Roger Descheniaux, David Gosselin, Thorsten Scheuermann, John Isidoro, Chris Oat, Jason Mitchell, Dan Gessel, and Evan Hart.

I would like to extend a special thanks to my wife, Sofia, for her support while I worked on this book. I would also like to thank my son, Ethan, who was born during the writing of this book. Your smile and laugh bring me joy every single day.

— Dan Ginsburg

I would like to express my deepest gratitude to Dan Ginsburg for providing me with an opportunity to contribute to this book. Thank you to my manager, Callan McInally, and colleagues at AMD for supporting this endeavor. I would also like to thank my past professors, Jonathan Cohen, Subodh Kumar, Ching-Kuang Shene, and John Lowther, for introducing me to the world of computer graphics and OpenGL.

I would like to thank my parents and sister for their unconditional love. Special thanks to my wonderful wife, Liana Hadi, whose love and support allowed me to complete this project. Thank you to my daughters, Michelle Lo and Scarlett Lo. They are the sunshine in my life.

— Budi Purnomo
We all want to thank Neil Trevett for writing the Foreword and getting approval from the Khronos Board of Promoters to allow us to use text from the OpenGL ES Shading Language specification in Appendix B, as well as the OpenGL ES 3.0 Reference Card. A special thank you and debt of gratitude go to the reviewers for their enormously valuable feedback—Maurice Ribble, Peter Lohrmann, and Emmanuel Agu. We also wish to acknowledge the technical reviewers from the first edition of the book—Brian Collins, Chris Grimm, Jeremy Sandmel, Tom Olson, and Adam Smith.

We owe a huge amount of gratitude to our editor, Laura Lewin, at Addison-Wesley, who was enormously helpful in every aspect of creating this book. There were many others at Addison-Wesley who were invaluable in putting together this book and whom we would like to thank, including Debra Williams Cauley, Olivia Basegio, Sheri Cain, and Curt Johnson.

We want to thank our readers from the first edition who have helped us immensely by reporting errata and improving the sample code. We would especially like to thank our reader Javed Rabbani Shah, who ported the OpenGL ES 3.0 sample code to the Android 4.3 SDK in Java. He also helped us with the Android NDK port and resolving many device-specific issues. We thank Jarkko Vatjus-Anttila for providing the Linux X11 port, and Eduardo Pelegri-Llopart and Darryl Gough for porting the first-edition code to the BlackBerry Native SDK.

A big thank you to the OpenGL ARB, the OpenGL ES working group, and everyone who contributed to the development of OpenGL ES.
About the Authors

Dan Ginsburg
Dan is the founder of Upsample Software, LLC, a software company offering consulting services in 3D graphics and GPU computing. Dan has coauthored several other books, including the OpenCL Programming Guide and OpenGL Shading Language, Third Edition. In previous roles Dan has worked on developing OpenGL drivers, desktop and handheld 3D demos, GPU developer tools, 3D medical visualization, and games. He holds a B.S. in computer science from Worcester Polytechnic Institute and an M.B.A. from Bentley University.

Budirijanto Purnomo
Budi is a senior software architect at Advanced Micro Devices, Inc., where he leads the software enablement efforts of GPU debugging and profiling technology across multiple AMD software stacks. He collaborates with many software and hardware architects within AMD to define future hardware architectures for debugging and profiling GPU applications. He has published many computer graphics technical articles at international conferences. He received his B.S. and M.S. in computer science from Michigan Technological University, and his M.S.E. and Ph.D. in computer science from Johns Hopkins University.

Aaftab Munshi
Affie has been architecting GPUs for more than a decade. At ATI (now AMD), he was a senior architect in the Handheld Group. He is the spec editor for the OpenGL ES 1.1, OpenGL ES 2.0, and OpenCL specifications. He currently works at Apple.
Dave Shreiner

Dave has been working with OpenGL for almost two decades, and more recently with OpenGL ES. He authored the first commercial training course on OpenGL while working at Silicon Graphics Computer Systems (SGI), and has worked as an author on the OpenGL Programming Guide. He has presented introductory and advanced courses on OpenGL programming worldwide at numerous conferences, including SIGGRAPH.

Dave is now a media systems architect at ARM, Inc. He holds a B.S. in mathematics from the University of Delaware.
Introduction to OpenGL ES 3.0

OpenGL for Embedded Systems (OpenGL ES) is an application programming interface (API) for advanced 3D graphics targeted at handheld and embedded devices. OpenGL ES is the dominant graphics API in today's smartphones and has even extended its reach onto the desktop. The list of platforms supporting OpenGL ES includes iOS, Android, BlackBerry, bada, Linux, and Windows. OpenGL ES also underpins WebGL, a web standard for browser-based 3D graphics.

Since the release of the iPhone 3GS in June 2009 and Android 2.0 in March 2010, OpenGL ES 2.0 has been supported on iOS and Android devices. The first edition of this book covered OpenGL ES 2.0 in detail. The current edition focuses on OpenGL ES 3.0, the next revision of OpenGL ES. It is almost inevitable that every handheld platform that continues to evolve will support OpenGL ES 3.0. Indeed, OpenGL ES 3.0 is already supported on devices using Android 4.3+ and on the iPhone 5s with iOS7. OpenGL ES 3.0 is backward compatible with OpenGL ES 2.0, meaning that applications written for OpenGL ES 2.0 will continue to work with OpenGL ES 3.0.

OpenGL ES is one of a set of APIs created by the Khronos Group. The Khronos Group, founded in January 2000, is a member-funded industry consortium that is focused on the creation of open standard and royalty-free APIs. The Khronos Group also manages OpenGL, a cross-platform standard 3D API for desktop systems running Linux, various flavors of UNIX, Mac OS X, and Microsoft Windows. It is a widely accepted standard 3D API that has seen significant real-world usage.

Due to the widespread adoption of OpenGL as a 3D API, it made sense to start with the desktop OpenGL API in developing an open standard 3D
Chapter 1: Introduction to OpenGL ES 3.0

API for handheld and embedded devices and then modify it to meet the needs and constraints of the handheld and embedded device space. In the earlier versions of OpenGL ES (1.0, 1.1, and 2.0), the device constraints that were considered in the design included limited processing capabilities and memory availability, low memory bandwidth, and sensitivity to power consumption. The working group used the following criteria in the definition of the OpenGL ES specification(s):

- The OpenGL API is very large and complex, and the goal of the OpenGL ES working group was to create an API suitable for constrained devices. To achieve this goal, the working group removed any redundancy from the OpenGL API. In any case where the same operation could be performed in more than one way, the most useful method was taken and the redundant techniques were removed. A good example of this is seen with specifying geometry, where in OpenGL an application can use immediate mode, display lists, or vertex arrays. In OpenGL ES, only vertex arrays exist; immediate mode and display lists were removed.

- Removing redundancy was an important goal, but maintaining compatibility with OpenGL was also important. As much as possible, OpenGL ES was designed so that applications written to the embedded subset of functionality in OpenGL would also run on OpenGL ES. This was an important goal because it allows developers to leverage both APIs and to develop applications and tools that use the common subset of functionality.

- New features were introduced to address specific constraints of handheld and embedded devices. For example, to reduce the power consumption and increase the performance of shaders, precision qualifiers were introduced to the shading language.

- The designers of OpenGL ES aimed to ensure a minimum set of features for image quality. In early handheld devices, the screen sizes were limited, making it essential that the quality of the pixels drawn on the screen was as good as possible.

- The OpenGL ES working group wanted to ensure that any OpenGL ES implementation would meet certain acceptable and agreed-on standards for image quality, correctness, and robustness. This was achieved by developing appropriate conformance tests that an OpenGL ES implementation must pass to be considered compliant.

Khronos has released four OpenGL ES specifications so far: OpenGL ES 1.0 and ES 1.1 (referred to jointly as OpenGL ES 1.x in this book), OpenGL ES 2.0, and OpenGL ES 3.0. The OpenGL ES 1.0 and 1.1 specifications
implement a fixed function pipeline and are derived from the OpenGL 1.3 and 1.5 specifications, respectively.

The OpenGL ES 2.0 specification implements a programmable graphics pipeline and is derived from the OpenGL 2.0 specification. Being derived from a revision of the OpenGL specification means that the corresponding OpenGL specification was used as the baseline for determining the feature set included in the particular revision of OpenGL ES.

OpenGL ES 3.0 is the next step in the evolution of handheld graphics and is derived from the OpenGL 3.3 specification. While OpenGL ES 2.0 was successful in bringing capabilities similar to DirectX9 and the Microsoft Xbox 360 to handheld devices, graphics capabilities have continued to evolve on desktop GPUs. Significant features that enable techniques such as shadow mapping, volume rendering, GPU-based particle animation, geometry instancing, texture compression, and gamma correction were missing from OpenGL ES 2.0. OpenGL ES 3.0 brings these features to handheld devices, while continuing the philosophy of adapting to the constraints of embedded systems.

Of course, some of the constraints that were taken into consideration while designing previous versions of OpenGL ES are no longer relevant today. For example, handheld devices now feature large screen sizes (some offer a higher resolution than most desktop PC monitors). Additionally, many handheld devices now feature high-performance multicore CPUs and large amounts of memory. The focus for the Khronos Group in developing OpenGL ES 3.0 shifted toward appropriate market timing of features relevant to handheld applications rather than addressing the limited capabilities of devices.

The following sections introduce the OpenGL ES 3.0 pipeline.

**OpenGL ES 3.0**

As noted earlier, OpenGL ES 3.0 is the API covered in this book. Our goal is to cover the OpenGL ES 3.0 specification in thorough detail, give specific examples of how to use the features in OpenGL ES 3.0, and discuss various performance optimization techniques. After reading this book, you should have an excellent grasp of the OpenGL ES 3.0 API, be able to easily write compelling OpenGL ES 3.0 applications, and not have to worry about reading multiple specifications to understand how a feature works.

OpenGL ES 3.0 implements a graphics pipeline with programmable shading and consists of two specifications: the **OpenGL ES 3.0**
API specification and the OpenGL ES Shading Language 3.0 Specification (OpenGL ES SL). Figure 1-1 shows the OpenGL ES 3.0 graphics pipeline. The shaded boxes in this figure indicate the programmable stages of the pipeline in OpenGL ES 3.0. An overview of each stage in the OpenGL ES 3.0 graphics pipeline is presented next.

**Figure 1-1** OpenGL ES 3.0 Graphics Pipeline

### Vertex Shader

This section gives a high-level overview of vertex shaders. Vertex and fragment shaders are covered in depth in later chapters. The vertex shader implements a general-purpose programmable method for operating on vertices.

The inputs to the vertex shader consist of the following:

- **Shader program**—Vertex shader program source code or executable that describes the operations that will be performed on the vertex.
- **Vertex shader inputs (or attributes)**—Per-vertex data supplied using vertex arrays.
- **Uniforms**—Constant data used by the vertex (or fragment) shader.
- **Samplers**—Specific types of uniforms that represent textures used by the vertex shader.
The outputs of the vertex shader were called varying variables in OpenGL ES 2.0, but were renamed vertex shader output variables in OpenGL ES 3.0. In the primitive rasterization stage, the vertex shader output values are calculated for each generated fragment and are passed in as inputs to the fragment shader. The mechanism used to generate a value for each fragment from the vertex shader outputs that is assigned to each vertex of the primitive is called interpolation. Additionally, OpenGL ES 3.0 adds a new feature called transform feedback, which allows the vertex shader outputs to be selectively written to an output buffer (in addition to, or instead of, being passed to the fragment shader). For example, as covered in the transform feedback example in Chapter 14, a particle system can be implemented in the vertex shader in which particles are output to a buffer object using transform feedback. The inputs and outputs of the vertex shader are shown in Figure 1-2.

![OpenGL ES 3.0 Vertex Shader Diagram](image)

**Figure 1-2** OpenGL ES 3.0 Vertex Shader

Vertex shaders can be used for traditional vertex-based operations such as transforming the position by a matrix, computing the lighting equation to generate a per-vertex color, and generating or transforming texture...
coordinates. Alternatively, because the vertex shader is specified by the application, vertex shaders can be used to perform custom math that enables new transforms, lighting, or vertex-based effects not allowed in more traditional fixed-function pipelines.

Example 1-1 shows a vertex shader written using the OpenGL ES shading language. We explain vertex shaders in significant detail later in the book. We present this shader here just to give you an idea of what a vertex shader looks like. The vertex shader in Example 1-1 takes a position and its associated color data as input attributes, transforms the position using a $4 \times 4$ matrix, and outputs the transformed position and color.

**Example 1-1**  A Vertex Shader Example

```cpp
1.   #version 300 es
2.   uniform mat4 u_mvpMatrix; // matrix to convert a_position
3.       // from model space to normalized
4.       // device space
5.
6.   // attributes input to the vertex shader
7.   in vec4 a_position;       // position value
8.   in vec4 a_color;          // input vertex color
9.
10.  // output of the vertex shader - input to fragment
11.  // shader
12.  out vec4 v_color;         // output vertex color
13.  void main()
14.  {
15.     v_color = a_color;
16.     gl_Position = u_mvpMatrix * a_position;
17.  }
```

Line 1 provides the version of the Shading Language—information that must appear on the first line of the shader (#version 300 es indicates the OpenGL ES Shading Language v3.00). Line 2 describes a uniform variable u_mvpMatrix that stores the combined model view and projection matrix. Lines 7 and 8 describe the inputs to the vertex shader and are referred to as vertex attributes. a_position is the input vertex position attribute and a_color is the input vertex color attribute. On line 12, we declare the output v_color to store the output of the vertex shader that describes the per-vertex color. The built-in variable called gl_Position is declared automatically, and the shader must write the transformed position to this variable. A vertex or fragment shader has a single entry point called the main function. Lines 13–17 describe the

---

*Chapter 1: Introduction to OpenGL ES 3.0*
vertex shader main function. In line 15, we read the vertex attribute input a_color and write it as the vertex output color v_color. In line 16, the transformed vertex position is output by writing it to gl_Position.

**Primitive Assembly**

After the vertex shader, the next stage in the OpenGL ES 3.0 graphics pipeline is primitive assembly. A primitive is a geometric object such as a triangle, line, or point sprite. Each vertex of a primitive is sent to a different copy of the vertex shader. During primitive assembly, these vertices are grouped back into the primitive.

For each primitive, it must be determined whether the primitive lies within the view frustum (the region of 3D space that is visible on the screen). If the primitive is not completely inside the view frustum, it might need to be clipped to the view frustum. If the primitive is completely outside this region, it is discarded. After clipping, the vertex position is converted to screen coordinates. A culling operation can also be performed that discards primitives based on whether they face forward or backward. After clipping and culling, the primitive is ready to be passed to the next stage of the pipeline—the rasterization stage.

**Rasterization**

The next stage, shown in Figure 1-3, is the rasterization phase, where the appropriate primitive (point sprite, line, or triangle) is drawn. Rasterization is the process that converts primitives into a set of two-dimensional fragments, which are then processed by the fragment shader. These two-dimensional fragments represent pixels that can be drawn on the screen.

![Diagram](https://via.placeholder.com/150)

*Figure 1-3 OpenGL ES 3.0 Rasterization Stage*
Fragment Shader

The fragment shader implements a general-purpose programmable method for operating on fragments. As shown in Figure 1-4, this shader is executed for each generated fragment by the rasterization stage and takes the following inputs:

- **Shader program**—Fragment shader program source code or executable that describes the operations that will be performed on the fragment.
- **Input variables**—Outputs of the vertex shader that are generated by the rasterization unit for each fragment using interpolation.
- **Uniforms**—Constant data used by the fragment (or vertex) shader.
- **Samplers**—Specific types of uniforms that represent textures used by the fragment shader.

The fragment shader can either discard the fragment or generate one or more color values referred to as outputs. Typically, the fragment shader outputs just

![Fragment Shader Diagram](image)

**Figure 1-4** OpenGL ES 3.0 Fragment Shader
a single color value, except when rendering to multiple render targets (see the section Multiple Render Targets in Chapter 11); in the latter case, a color value is output for each render target. The color, depth, stencil, and screen coordinate location \((x_w, y_w)\) generated by the rasterization stage become inputs to the per-fragment operations stage of the OpenGL ES 3.0 pipeline.

Example 1-2 describes a simple fragment shader that can be coupled with the vertex shader described in Example 1-1 to draw a Gouraud-shaded triangle. Again, we will go into much more detail on fragment shaders later in the book. We present this example just to give you a basic idea of what a fragment shader looks like.

**Example 1-2  A Fragment Shader Example**

1.  #version 300 es
2.  precision mediump float;
3.  
4.  in vec4 v_color;    // input vertex color from vertex shader
5.  
6.  out vec4 fragColor; // output fragment color
7.  void main()
8.  {
9.     fragColor = v_color;
10. }

Just as in the vertex shader, line 1 provides the version of the Shading Language; this information must appear on the first line of the fragment shader (#version 300 es indicates the OpenGL ES Shading Language v3.00). Line 2 sets the default precision qualifier, which is explained in detail in Chapter 4, “Shaders and Programs.” Line 4 describes the input to the fragment shader. The vertex shader must write out the same set of variables that are read in by the fragment shader. Line 6 provides the declaration for the output variable of the fragment shader, which will be the color passed on to the next stage. Lines 7–10 describe the fragment shader main function. The output color is set to the input color \(v\_color\). The inputs to the fragment shader are linearly interpolated across the primitive before being passed into the fragment shader.

**Per-Fragment Operations**

After the fragment shader, the next stage is per-fragment operations. A fragment produced by rasterization with \((x_w, y_w)\) screen coordinates can only modify the pixel at location \((x_w, y_w)\) in the framebuffer. Figure 1-5 describes the OpenGL ES 3.0 per-fragment operations stage.
During the per-fragment operations stage, the following functions (and tests) are performed on each fragment, as shown in Figure 1-5:

- **Pixel ownership test**—This test determines whether the pixel at location \((x, y)\) in the framebuffer is currently owned by OpenGL ES. This test allows the window system to control which pixels in the framebuffer belong to the current OpenGL ES context. For example, if a window displaying the OpenGL ES framebuffer window is obscured by another window, the windowing system may determine that the obscured pixels are not owned by the OpenGL ES context and, therefore, the pixels might not be displayed at all. While the pixel ownership test is part of OpenGL ES, it is not controlled by the developer, but rather takes place internally inside of OpenGL ES.

- **Scissor test**—The scissor test determines whether \((x, y)\) lies within the scissor rectangle defined as part of the OpenGL ES state. If the fragment is outside the scissor region, the fragment is discarded.

- **Stencil and depth tests**—These tests are performed on the stencil and depth value of the incoming fragment to determine whether the fragment should be rejected.

- **Blending**—Blending combines the newly generated fragment color value with the color values stored in the framebuffer at location \((x, y)\).

- **Dithering**—Dithering can be used to minimize the artifacts that occur as a result of using limited precision to store color values in the framebuffer.

At the end of the per-fragment stage, either the fragment is rejected or a fragment color(s), depth, or stencil value is written to the framebuffer at location \((x, y)\). Writing of the fragment color(s), depth, and stencil values depends on whether the appropriate write masks are enabled. Write masks allow finer control over the color, depth, and stencil values written into the associated buffers. For example, the write mask for the...
color buffer could be set such that no red values are written into the color buffer. In addition, OpenGL ES 3.0 provides an interface to read back the pixels from the framebuffer.

Note: Alpha test and LogicOp are no longer part of the per-fragment operations stage. These two stages exist in OpenGL 2.0 and OpenGL ES 1.x. The alpha test stage is no longer needed because the fragment shader can discard fragments; thus the alpha test can be performed in the fragment shader. In addition, LogicOp was removed because it is used only rarely by applications, and the OpenGL ES working group did not receive requests from independent software vendors (ISVs) to support this feature in OpenGL ES 2.0.

What’s New in OpenGL ES 3.0

OpenGL ES 2.0 ushered in the era of programmable shaders for handheld devices and has been wildly successful in powering games, applications, and user interfaces across a wide range of devices. OpenGL ES 3.0 extends OpenGL ES 2.0 to support many new rendering techniques, optimizations, and visual quality enhancements. The following sections provide a categorized overview of the major new features that have been added to OpenGL ES 3.0. Each of these features will be described in detail later in the book.

Texturing

OpenGL ES 3.0 introduces many new features related to texturing:

• sRGB textures and framebuffers—Allow the application to perform gamma-correct rendering. Textures can be stored in gamma-corrected sRGB space, uncorrected to linear space upon being fetched in the shader, and then converted back to sRGB gamma-corrected space on output to the framebuffer. This enables potentially higher visual fidelity by properly computing lighting and other calculations in linear space.

• 2D texture arrays—A texture target that stores an array of 2D textures. Such arrays might, for example, be used to perform texture animation. Prior to 2D texture arrays, such animation was typically done by tiling the frames of an animation in a single 2D texture and modifying the texture coordinates to change animation frames. With 2D texture arrays, each frame of the animation can be specified in a 2D slice of the array.
• 3D textures—While some OpenGL ES 2.0 implementations supported 3D textures through an extension, OpenGL ES 3.0 has made this a mandatory feature. 3D textures are essential in many medical imaging applications, such as those that perform direct volume rendering of 3D voxel data (e.g., CT, MRI, or PET data).

• Depth textures and shadow comparison—Enable the depth buffer to be stored in a texture. The most common use for depth textures is in rendering shadows, where a depth buffer is rendered from the viewpoint of the light source and then used for comparison when rendering the scene to determine whether a fragment is in shadow. In addition to depth textures, OpenGL ES 3.0 allows the comparison against the depth texture to be done at the time of fetch, thereby allowing bilinear filtering to be done on depth textures (also known as percentage closest filtering [PCF]).

• Seamless cubemaps—in OpenGL ES 2.0, rendering with cubemaps could produce artifacts at the boundaries between cubemap faces. In OpenGL ES 3.0, cubemaps can be sampled such that filtering uses data from adjacent faces and removes the seaming artifact.

• Floating-point textures—OpenGL ES 3.0 greatly expands on the texture formats supported. Floating-point half-float (16-bit) textures are supported and can be filtered, whereas full-float (32-bit) textures are supported but not filterable. The ability to access floating-point texture data has many applications, including high dynamic range texturing to general-purpose computation.

• ETC2/EAC texture compression—While several OpenGL ES 2.0 implementations provided support for vendor-specific compressed texture formats (e.g., ATC by Qualcomm, PVRTC by Imagination Technologies, and Ericsson Texture Compression by Sony Ericsson), there was no standard compression format that developers could rely on. In OpenGL ES 3.0, support for ETC2/EAC is mandatory. The ETC2/EAC formats provide compression for RGB888, RGBA8888, and one- and two-channel signed/unsigned texture data. Texture compression offers several advantages, including better performance (due to better utilization of the texture cache) as well as a reduction in GPU memory utilization.

• Integer textures—OpenGL ES 3.0 introduces the capability to render to and fetch from textures stored as unnormalized signed or unsigned 8-bit, 16-bit, and 32-bit integer textures.

• Additional texture formats—in addition to those formats already mentioned, OpenGL ES 3.0 includes support for 11-11-10 RGB
floating-point textures, shared exponent RGB 9-9-9-5 textures, 10-10-10-2 integer textures, and 8-bit-per-component signed normalized textures.

- Non-power-of-2 textures (NPOT)—Textures can now be specified with non-power-of-2 dimensions. This is useful in many situations, such as when texturing from a video or camera feed that is captured/recorded at a non-power-of-2 dimension.

- Texture level of detail (LOD) features—The texture LOD parameter used to determine which mipmap to fetch from can now be clamped. Additionally, the base and maximum mipmap level can be clamped. These two features, in combination, make it possible to stream mipmaps. As larger mipmap levels become available, the base level can be increased and the LOD value can be smoothly increased to provide smooth-looking streaming textures. This is very useful, for example, when downloading texture mipmap data over a network connection.

- Texture swizzles—A new texture object state was introduced to allow independent control of where each channel (R, G, B, and A) of texture data is mapped to in the shader.

- Immutable textures—Provide a mechanism for the application to specify the format and size of a texture before loading it with data. In doing so, the texture format becomes immutable and the OpenGL ES driver can perform all consistency and memory checks up-front. This can improve performance by allowing the driver to skip consistency checks at draw time.

- Increased minimum sizes—All OpenGL ES 3.0 implementations are required to support much larger texture resources than OpenGL ES 2.0. For example, the minimum supported 2D texture dimension in OpenGL ES 2.0 was 64 but was increased to 2048 in OpenGL ES 3.0.

### Shaders

OpenGL ES 3.0 includes a major update to the OpenGL ES Shading Language (ESSL; to v3.00) and new API features to support new shader features:

- Program binaries—In OpenGL ES 2.0, it was possible to store shaders in a binary format, but it was still required to link them into program at runtime. In OpenGL ES 3.0, the entire linked program binary (containing the vertex and fragment shader) can be stored in an
offline binary format with no link step required at runtime. This can potentially help reduce the load time of applications. Additionally, OpenGL ES 3.0 provides an interface to retrieve the program binary from the driver so no offline tools are required to use program binaries.

- Mandatory online compiler—OpenGL ES 2.0 made it optional whether the driver would support online compilation of shaders. The intent was to reduce the memory requirements of the driver, but this achievement came at a major cost to developers in terms of having to rely on vendor-specific tools to generate shaders. In OpenGL ES 3.0, all implementations will have an online shader compiler.

- Non-square matrices—New matrix types other than square matrices are supported, and associated uniform calls were added to the API to support loading them. Non-square matrices can reduce the instruction count required for performing transformations. For example, if performing an affine transformation, a 4 × 3 matrix can be used in place of a 4 × 4 where the last row is (0, 0, 0, 1), thus reducing the instructions required to perform the transformation.

- Full integer support—Integer (and unsigned integer) scalar and vector types, along with full integer operations, are supported in ESSL 3.00. There are various built-in functions such as conversion from int to float, and from float to int, as well as the ability to read integer values from textures and output integer values to integer color buffers.

- Centroid sampling—To avoid rendering artifacts when multisampling, the output variables from the vertex shader (and inputs to the fragment shader) can be declared with centroid sampling.

- Flat/smooth interpolators—in OpenGL ES 2.0, all interpolators were implicitly linearly interpolated across the primitive. In ESSL 3.00, interpolators (vertex shader outputs/fragment shader inputs) can be explicitly declared to have either smooth or flat shading.

- Uniform blocks—Uniform values can be grouped together into uniform blocks. Uniform blocks can be loaded more efficiently and also shared across multiple shader programs.

- Layout qualifiers—Vertex shader inputs can be declared with layout qualifiers to explicitly bind the location in the shader source without requiring making API calls. Layout qualifiers can also be used for fragment shader outputs to bind the outputs to each target when rendering to multiple render targets. Further, layout qualifiers can be used to control the memory layout for uniform blocks.
• Instance and vertex ID—The vertex index is now accessible in the vertex shader as well as the instance ID if using instance rendering.

• Fragment depth—The fragment shader can explicitly control the depth value for the current fragment rather than relying on the interpolation of its depth value.

• New built-in functions—ESSL 3.00 introduces many new built-in functions to support new texture features, fragment derivatives, half-float data conversion, and matrix and math operations.

• Relaxed limitations—ESSL 3.0 greatly relaxes the restrictions on shaders. Shaders are no longer limited in terms of instruction length, fully support looping and branching on variables, and support indexing on arrays.

Geometry

OpenGL ES 3.0 introduces several new features related to geometry specification and control of primitive rendering:

• Transform feedback—Allows the output of the vertex shader to be captured in a buffer object. This is useful for a wide range of techniques that perform animation on the GPU without any CPU intervention—for example, particle animation or physics simulation using render-to-vertex-buffer.

• Boolean occlusion queries—Enable the application to query whether any pixels of a draw call (or a set of draw calls) passes the depth test. This feature can be used within a variety of techniques, such as visibility determination for a lens flare effect as well as optimization to avoid performing geometry processing on objects whose bounding volume is obscured.

• Instanced rendering—Efficiently renders objects that contain similar geometry but differ by attributes (such as transformation matrix, color, or size). This feature is useful in rendering large quantities of similar objects, such as for crowd rendering.

• Primitive restart—When using triangle strips in OpenGL ES 2.0 for a new primitive, the application would have to insert indices into the index buffer to represent a degenerate triangle. In OpenGL ES 3.0, a special index value can be used that indicates the beginning of a new primitive. This obviates the need for generating degenerate triangles when using triangle strips.
• New vertex formats—New vertex formats, including 10-10-10-2 signed and unsigned normalized vertex attributes; 8-bit, 16-bit, and 32-bit integer attributes; and 16-bit half-float, are supported in OpenGL ES 3.0.

Buffer Objects

OpenGL ES 3.0 introduces many new buffer objects to increase the efficiency and flexibility of specifying data to various parts of the graphics pipeline:

• Uniform buffer objects—Provide an efficient method for storing/binding large blocks of uniforms. Uniform buffer objects can be used to reduce the performance cost of binding uniform values to shaders, which is a common bottleneck in OpenGL ES 2.0 applications.

• Vertex array objects—Provide an efficient method for binding and switching between vertex array states. Vertex array objects are essentially container objects for vertex array states. Using them allows an application to switch the vertex array state in a single API call rather than making several calls.

• Sampler objects—Separate the sampler state (texture wrap mode and filtering) from the texture object. This provides a more efficient method of sharing the sampler state across textures.

• Sync objects—Provide a mechanism for the application to check on whether a set of OpenGL ES operations has finished executing on the GPU. A related new feature is a fence, which provides a way for the application to inform the GPU that it should wait until a set of OpenGL ES operations has finished executing before queuing up more operations for execution.

• Pixel buffer objects—Enable the application to perform asynchronous transfer of data to pixel operations and texture transfer operations. This optimization is primarily intended to provide faster transfer of data between the CPU and the GPU, where the application can continue doing work during the transfer operation.

• Buffer subrange mapping—Allows the application to map a subregion of a buffer for access by the CPU. This can provide better performance than traditional buffer mapping, in which the whole buffer needs to be available to the client.

• Buffer object to buffer object copies—Provide a mechanism to efficiently transfer data from one buffer object to another without intervention on the CPU.
Framebuffer

OpenGL ES 3.0 adds many new features related to off-screen rendering to framebuffer objects:

- **Multiple render targets (MRTs)**—Allow the application to render simultaneously to several color buffers at one time. With MRTs, the fragment shader outputs several colors, one for each attached color buffer. MRTs are used in many advanced rendering algorithms, such as deferred shading.

- **Multisample renderbuffers**—Enable the application to render to off-screen framebuffers with multisample anti-aliasing. The multisample renderbuffers cannot be directly bound to textures, but they can be resolved to single-sample textures using the newly introduced framebuffer blit.

- **Framebuffer invalidation hints**—Many implementations of OpenGL ES 3.0 are based on GPUs that use tile-based rendering (TBR; explained in the Framebuffer Invalidation section in Chapter 12). It is often the case that TBR incurs a significant performance cost when having to unnecessarily restore the contents of the tiles for further rendering to a framebuffer. Framebuffer invalidation gives the application a mechanism to inform the driver that the contents of the framebuffer are no longer needed. This allows the driver to take optimization steps to skip unnecessary restore operations on the tiles. Such functionality is very important to achieve peak performance in many applications, especially those that do significant amounts of off-screen rendering.

- **New blend equations**—The min/max functions are supported in OpenGL ES 3.0 as a blend equation.

OpenGL ES 3.0 and Backward Compatibility

OpenGL ES 3.0 is backward compatible with OpenGL ES 2.0. This means that just about any application written to use OpenGL ES 2.0 will run on implementations of OpenGL ES 3.0. There are some very minor changes to the later version that will affect a small number of applications in terms of backward compatibility. Namely, framebuffer objects are no longer shared between contexts, cubemaps are always filtered using seamless filtering, and there are minor changes in the way signed fixed-point numbers are converted to floating-point numbers.

*OpenGL ES 3.0 and Backward Compatibility*
The fact that OpenGL ES 3.0 is backward compatible with OpenGL ES 2.0 differs from what was done for OpenGL ES 2.0 with respect to its backward compatibility with previous versions of OpenGL ES. OpenGL ES 2.0 is not backward compatible with OpenGL ES 1.x. OpenGL ES 2.0/3.0 do not support the fixed-function pipeline that OpenGL ES 1.x supports. The OpenGL ES 2.0/3.0 programmable vertex shader replaces the fixed-function vertex units implemented in OpenGL ES 1.x. The fixed-function vertex units implement a specific vertex transformation and lighting equation that can be used to transform the vertex position, transform or generate texture coordinates, and calculate the vertex color. Similarly, the programmable fragment shader replaces the fixed-function texture combine units implemented in OpenGL ES 1.x. The fixed-function texture combine units implement a texture combine stage for each texture unit. The texture color is combined with the diffuse color and the output of the previous texture combine stage with a fixed set of operations such as add, modulate, subtract, and dot.

The OpenGL ES working group decided against backward compatibility between OpenGL ES 2.0/3.0 and OpenGL ES 1.x for the following reasons:

- Supporting the fixed-function pipeline in OpenGL ES 2.0/3.0 implies that the API would support more than one way of implementing a feature, in violation of one of the criteria used by the working group in determining which features should be supported. The programmable pipeline allows applications to implement the fixed-function pipeline using shaders, so there is really no compelling reason to be backward compatible with OpenGL ES 1.x.

- Feedback from ISVs indicated that most games do not mix programmable and fixed-function pipelines. That is, games are written either for a fixed-function pipeline or for a programmable pipeline. Once you have a programmable pipeline, there is no reason to use a fixed-function pipeline, as you have much more flexibility in the effects that can be rendered.

- The OpenGL ES 2.0/3.0 driver’s memory footprint would be much larger if it had to support both the fixed-function and programmable pipelines. For the devices targeted by OpenGL ES, minimizing memory footprint is an important design criterion. Separating the fixed-function support into the OpenGL ES 1.x API and placing the programmable shader support into the OpenGL ES 2.0/3.0 APIs meant that vendors that do not require OpenGL ES 1.x support no longer need to include this driver.
OpenGL ES commands require a rendering context and a drawing surface. The rendering context stores the appropriate OpenGL ES state. The drawing surface is the surface to which primitives will be drawn. The drawing surface specifies the types of buffers that are required for rendering, such as a color buffer, depth buffer, and stencil buffer. The drawing surface also specifies the bit depths of each of the required buffers.

The OpenGL ES API does not mention how a rendering context is created or how the rendering context gets attached to the native windowing system. EGL is one interface between the Khronos rendering APIs such as OpenGL ES and the native window system; there is no hard-and-fast requirement to provide EGL when implementing OpenGL ES. Developers should refer to the platform vendor’s documentation to determine which interface is supported. As of this writing, the only known platform supporting OpenGL ES that does not support EGL is iOS.

Any OpenGL ES application will need to perform the following tasks using EGL before any rendering can begin:

- Query the displays that are available on the device and initialize them. For example, a flip phone might have two LCD panels, and it is possible that we might use OpenGL ES to render to surfaces that can be displayed on either or both panels.

- Create a rendering surface. Surfaces created in EGL can be categorized as on-screen surfaces or off-screen surfaces. On-screen surfaces are attached to the native window system, whereas off-screen surfaces are pixel buffers that do not get displayed but can be used as rendering surfaces. These surfaces can be used to render into a texture and can be shared across multiple Khronos APIs.

- Create a rendering context. EGL is needed to create an OpenGL ES rendering context. This context needs to be attached to an appropriate surface before rendering can actually begin.

The EGL API implements the features just described as well as additional functionality such as power management, support for multiple rendering contexts in a process, sharing objects (such as textures or vertex buffers) across rendering contexts in a process, and a mechanism to get function pointers to EGL or OpenGL ES extension functions supported by a given implementation.

The latest version of the EGL specification is EGL version 1.4.
Chapter 1: Introduction to OpenGL ES 3.0

To write any OpenGL ES 3.0 application, you need to know which header files must be included and with which libraries your application needs to link. It is also useful to understand the syntax used by the EGL and GL command names and command parameters.

Libraries and Include Files

OpenGL ES 3.0 applications need to link with the following libraries: the OpenGL ES 3.0 library named \texttt{libGLESv2.lib} and the EGL library named \texttt{libEGL.lib}.

OpenGL ES 3.0 applications also need to include the appropriate ES 3.0 and EGL header files. The following include files must be included by any OpenGL ES 3.0 application:

\begin{verbatim}
#include <EGL/egl.h>
#include <GLES3/gl3.h>
\end{verbatim}

egl.h is the EGL header file and gl3.h is the OpenGL ES 3.0 header file. Applications can optionally include gl2ext.h, which is the header file that describes the list of Khronos-approved extensions for OpenGL ES 2.0/3.0.

The header file and library names are platform dependent. The OpenGL ES working group has tried to define the library and header names and indicate how they should be organized, but this arrangement might not be found on all OpenGL ES platforms. Developers should, however, refer to the platform vendor's documentation for information on how the libraries and include files are named and organized. The official OpenGL ES header files are maintained by Khronos and available from \url{http://khronos.org/registry/gles/}. The sample code for the book also includes a copy of the header files (working with the sample code is described in the next chapter).

EGL Command Syntax

All EGL commands begin with the prefix \texttt{egl} and use an initial capital letter for each word making up the command name (e.g., \texttt{eglCreateWindowSurface}). Similarly, EGL data types also begin with the prefix \texttt{Egl} and use an initial capital letter for each word making up the type name, except for \texttt{EGLint} and \texttt{EGLenum}.

Table 1-1 briefly describes the EGL data types used.
OpenGL ES Command Syntax

All OpenGL ES commands begin with the prefix gl and use an initial capital letter for each word making up the command name (e.g., glBlendEquation). Similarly, OpenGL ES data types also begin with the prefix GL.

In addition, some commands might take arguments in different flavors. The flavors or types vary in terms of the number of arguments taken (one to four arguments), the data type of the arguments used (byte [b], unsigned byte [ub], short [s], unsigned short [us], int [i], and float [f]), and whether the arguments are passed as a vector (v). A few examples of command flavors allowed in OpenGL ES follow:

The following two commands are equivalent except that one specifies the uniform value as floats and the other as integers:

```c
glUniform2f(location, 1.0f, 0.0f);
glUniform2i(location, 1, 0)
```

The following lines describe commands that are also equivalent, except that one passes command arguments as a vector and the other does not:

```c
GLfloat coord[4] = { 1.0f, 0.75f, 0.25f, 0.0f };
glUniform4fv(location, coord);
glUniform4f(location, coord[0], coord[1], coord[2], coord[3]);
```

Table 1-2 describes the command suffixes and argument data types used in OpenGL ES.

Finally, OpenGL ES defines the type GLvoid. This type is used for OpenGL ES commands that accept pointers.

In the rest of this book, OpenGL ES commands are referred to by their base names only, and an asterisk is used to indicate that this base name refers
Chapter 1: Introduction to OpenGL ES 3.0

to multiple flavors of the command name. For example, `glUniform*()` stands for all variations of the command you use to specify uniforms and `glUniform*v()` refers to all the vector versions of the command you use to specify uniforms. If a particular version of a command needs to be discussed, we use the full command name with the appropriate suffixes.

**Error Handling**

OpenGL ES commands incorrectly used by applications generate an error code. This error code is recorded and can be queried using `glGetError`. No other errors will be recorded until the application has queried the first error code using `glGetError`. Once the error code has been queried, the current error code is reset to `GL_NO_ERROR`. The command that generated the error is ignored and does not affect the OpenGL ES state except for the `GL_OUT_OF_MEMORY` error described later in this section.

The `glGetError` command is described next.
Table 1-3 lists the basic error codes and their description. Other error codes besides the basic ones listed in this table are described in the chapters that cover OpenGL ES commands that generate these specific errors.

Table 1-3 OpenGL ES Basic Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_NO_ERROR</td>
<td>No error has been generated since the last call to glGetError.</td>
</tr>
<tr>
<td>GL_INVALID_ENUM</td>
<td>A GLenum argument is out of range. The command that generated the error is ignored.</td>
</tr>
<tr>
<td>GL_INVALID_VALUE</td>
<td>A numeric argument is out of range. The command that generated the error is ignored.</td>
</tr>
<tr>
<td>GL_INVALID_OPERATION</td>
<td>The specific command cannot be performed in the current OpenGL ES state. The command that generated the error is ignored.</td>
</tr>
<tr>
<td>GL_OUT_OF_MEMORY</td>
<td>There is insufficient memory to execute this command. The state of the OpenGL ES pipeline is considered to be undefined if this error is encountered except for the current error code.</td>
</tr>
</tbody>
</table>

Basic State Management

Figure 1-1 showed the various pipeline stages in OpenGL ES 3.0. Each pipeline stage has a state that can be enabled or disabled and appropriate state values that are maintained per context. Examples of states are blending enable, blend factors, cull enable, and cull face. The state is initialized with default values when an OpenGL ES context (EGLContext) is initialized. The state enables can be set using the glEnable and glDisable commands.
Chapter 1: Introduction to OpenGL ES 3.0

Later chapters will describe the specific state enables for each pipeline stage shown in Figure 1-1. You can also check whether a state is currently enabled or disabled by using the `glIsEnabled` command.

Specific state values such as blend factor, depth test values, and so on can also be queried using appropriate `glGet***` commands. These commands are described in detail in Chapter 15, “State Queries.”
Further Reading

The OpenGL ES 1.0, 1.1, 2.0, and 3.0 specifications can be found at khronos.org/opengles/. In addition, the Khronos website (khronos.org) has the latest information on all Khronos specifications, developer message boards, tutorials, and examples.

- Khronos OpenGL ES 1.1 website: http://khronos.org/opengles/1_X/
- Khronos OpenGL ES 2.0 website: http://khronos.org/opengles/2_X/
- Khronos OpenGL ES 3.0 website: http://khronos.org/opengles/3_X/
- Khronos EGL website: http://khronos.org/egl/
Index

.(dot), vector access operator, 101–102
#elif directive, 116
#else directive, 116
#error directive, 116
#extension directive, 116
#if directive, 116
#pragma directive
definition, 116
enabling global invariance, 123
[] (square brackets), array subscript operator, 101–102
2D texture arrays
loading, 260–262
new features, 11
overview, 230
2D textures
attached to framebuffer objects, 338–339
base formats, 227
overview, 226–227
in shaders, 255–257
3D textures
attached to framebuffer objects, 339–340
loading, 260–262
new features, 12
overview, 229
3D textures, noise
dust effects, 403–404
water wave simulation, 404
wispy fog, 402–404
A
abs function, 467
acos function, 465
acosh function, 466
Advanced RenderMan: Creating CGI for Motion
Pictures, 407
Aliasing artifacts. See Anti-aliasing;
Mipmapping.
all function, 473
Alpha test, 291–293
Android 4.3 (API 18), 451
Android NDK, 451
Android SDK, 451
Angles, built-in functions, 465–466
Animation, 2D images. See 2D texture
arrays.
Anti-aliasing
multi-sampled, 314–316
procedural textures, 407–410
any function, 473
Apache Ant, 451
ARM Mali OpenGL ES Emulator, 448
Array buffer objects, 140–141
Arrays, 104
Arrays of structures, 128. See also
Structures.
asin function, 465
asinh function, 466
Asynchronous objects, querying,
442–443
atan function, 465
atanh function, 466
Attributes. See also specific attributes.
active, counting, 77
active, querying, 93
getting, 92–93
largest name, getting, 77
setting, 92–93
B
Back buffers, 41, 298
Backward compatibility, 17–18
Bias matrix, 392–393
Binaries. See Program binaries.
Binding
  program objects, example, 39
  renderbuffer objects, 330–331
  texture objects, 231
  vertex array objects, 151
  vertex attributes, 137–140
  vertex buffer objects, 141
Blend equations, new features, 17
Blending
  colors, 311–314
  per-fragment operations, 10
Blur effects, 387–390
Boolean occlusion queries, new features, 15
Buffer object to buffer object copies, new features, 16
Buffer objects
  copying, 159–160
  deleting, 150
  drawing with and without, 145–150
  initializing, 145
  updating, 145
Buffer objects, mapping
  changing screen resolution, 157
  data storage pointer, getting, 155–156
  flushing mapped buffers, 158–159
  overview, 154–155
  unmapping, 155–156
Buffer objects, new features. See also
  Uniform buffer objects; Vertex buffer objects.
  buffer object to buffer object copies, 16
  buffer subrange mapping, 16
  pixel buffer objects, 16
  sampler objects, 16
  sync objects, 16
  uniform buffer objects, 16
  vertex array objects, 16
Buffer subrange mapping, new features, 16
Buffer write masks, 301–303
Buffers, fragments. See also Pbuffers (pixel buffers).
  back, 298
  buffer write masks, 301–303
  clearing, 299–301
  depth of, 298
front, 298
making writable, 301–303
requesting, 299
size of, 298
swapping, 298–299
types of, 298
Built-in functions. See also Functions.
  abs function, 467
  acos, 465
  acosh, 466
  all, 473
  angles, 465–466
  any, 473
  asin, 465
  asinh, 466
  atan, 465
  atanh, 466
  ceil, 468
  clamp, 469
  cos, 465
  cosh, 466
  cross, 473
  degrees, 465
  description, 107
  determinant, 473
dFdx, 484
dFdy, 484
distance, 473
dot, 473
equal, 473
exp, 466
exp2, 466
exponential, 466–467
faceforward, 473
floatBitsToFloat, 470
floatBitsToUint, 470
floating-point pack and unpack, 471–472
floor, 467
fract, 468
fragment processing, 483–484
fwidth, 484
greaterThan, 475
greaterThanOrEqual, 475
intBitsToFloat, 470
inverse, 474
inversesqrt, 467
isnan, 470
isinf, 470
length, 473

Index
Index

lessThan, 475
lessThanEqual, 475
log, 466
log2, 467
max, 468
min, 468
mix, 469
mod, 468
new features, 15
normalize, 473
not, 475
notEqual, 475
outerProduct, 474
pow, 466
radians, 465
reflect, 473
refract, 473
round, 468
roundEven, 468
sin, 465
sinh, 466
smoothstep, 470
sqrt, 467
step, 470
tan, 465
tanh, 466
texture built-in, 478
textureGrad, 481
textureGradOffset, 481
textureLod, 479
textureOffset, 480
textureProj built-in, 479
textureProjGrad, 482
textureProjGradOffset, 482
textureProjLod, 480
textureProjLodOffset, 480
textureSize, 478
transpose, 474
trigonometry, 465–466
trunc, 467
uintBitsToFloat, 470
unpackHalf2x16, 472
unpackSnorm2x16, 471
unpackUnorm2x16, 471–472
vector relational, 475

C
ceil function, 468
centroid keyword, 115
Centroid sampling, 14, 316
Checkerboard example, 405–407
clamp function, 469
Client space, 126
Clip panes, user, 293–295
Clipping
description, 176–177
lines, 177
point sprites, 177
triangles, 177
Color
blending, 311–314
color depth, simulating, 314
depth, simulating, 314
dithering, 314
fragments, 311–314
specifying for multiple render targets, 321–322
Color buffers
clearing, example, 39–40
fragment operations, 298–299, 302.
See also Fragments, buffers.
column_major qualifier, 110
Combining texture maps, 286–287
Command syntax, 20–21
Compiling shaders, example, 36–38
Compressing textures, 262–265
Compression formats, textures, 264–265
Conditional statements. See Control flow statements.
Conditional tests, preprocessor directives, 115–117
cost declarations, examples, 102–103
Constant store, 109
Constants, description, 102–103
Constructors. See Variable constructors.
Control flow statements, 107–108
Coordinate systems
clipping, 176–177
guard band region, 177
overview, 175
Copying
buffer objects, 159–160
pixels in framebuffer objects, 342–344
textures from the color buffer, 269–273
cos function, 465
cosh function, 466
Creating
  EGL windows, 53–56, 64–65
  EGLContexts, 60–62
  fragment shaders, example, 35–36
  pbuffers, 56–60
  program objects, example, 38–39
  renderbuffer objects, 329–330
  rendering context, 60–62
  shaders, example, 35–36
  sync objects, 358–359
  texture objects, 230
  vertex array objects, 144, 151
  vertex buffer objects, 141
  vertex shaders, example, 35–36
  windows, example, 34–35

cross function, 473

Cubemaps
  example, 205–206
  seamless filtering, new features, 12, 241
  texturing, example, 258–260

Culling, 7, 180–181

Cygwin, 451

D

Data types
  EGL, 20–21
  matrix, 99–100
  scalar, 99–100
  type conversion, 100
  vector, 99–100

Deferred shading, multiple render targets, 320–321

Degenerate triangles, 172
degrees function, 465

Deleting
  buffer objects, 150
  framebuffer objects, 346–347
  program objects, 75
  renderbuffer objects, 346–347
  shaders, 70
  sync objects, 358–359
  texture objects, 230–231
  vertex array objects, 154

Deletion status, querying, 77

Depth buffer test, 311

Depth buffers, fragment operations. See also
  Fragments, buffers.
    buffer write masks, 302–303
    description, 298–299

Depth-of-field. See Rendering, to textures.

Depth test, per-fragment
  operations, 10

Depth texture compare, 245–246

Depth textures, 12, 254–255
determinant function, 473
dFdx function, 484
dFdy function, 484

Directional light, example, 199–202

Directives. See Preprocessor directives.
disable behavior, 117

Displacement mapping, vertex shaders, 214–215
distance function, 473

Dithering, 10, 314

Dot (.), vector access operator, 101–102
dot function, 473

Double buffering, example, 41

Drawing fragments, example, 35–36

Drawing primitives
  example, 40–41
  geometry instancing, 169–172
  multiple disconnected primitives, 168–169
  multiple primitives, different attributes, 169–172
  overview, 165–168
  performance tips, 172–174
  primitive restart, 168–169
  provoking vertex, 168–169

Drawing surface, creating, 325–327.
  See also FBOs (framebuffer objects).

Dust effects, 403–404

Dynamic reflections. See Rendering, to textures.

E

EGL
  command syntax, 20–21
  data types, 20–21
  description, 19
  display server, connecting to, 44–45
  include files, 20
  initializing, 44, 46
  libraries, 20
Index 499

programming with OpenGL ES 3.0, 20
rendering context, creating, 19
rendering surfaces, creating, 19
EGL error codes
EGL_BAD_ALLOC, 55, 58
EGL_BAD_ATTRIBUTE, 48, 51, 58
EGL_BAD_CONFIG, 55, 58, 61
EGL_BAD_DISPLAY, 46
EGL_BAD_MATCH, 55, 58
EGL_BAD_NATIVE_WINDOW, 55
EGL_BAD_PARAMETER, 47, 58, 67
EGL windows, creating
description, 53–56
with the esUtil library, 65–66
eglChooseConfig function, 51–53
EGLConfig data type
choosing surface configurations, 51–53
creating pbuffers, 56–60
determining available surface configurations, 46–47
EGLConfig data type, attributes
querying, 48–50
specifying, 51–52
summary of, 49–50
EGL_CONTEXT_CLIENT_VERSION attribute, 61
EGLContext
associating with an EGLSurface, 62–63
creating, 60–62
making current, 62–63
EGL_CORE_NATIVE_ENGINE, 67
eglCreateContext function, 60–62
eglCreatePbufferSurface command, 56–60
eglCreateWindowSurface function, 53–56
EGLDisplay data type, 44–45
eglGetConfigs function, 47
eglGetDisplay function, 44–45
eglGetError function, 45
EGL_HEIGHT attribute, 57
EGL_LARGEST_PBUFFER attribute, 57
eglMakeCurrent function, 62–63
EGL_MIPMAP_TEXTURE attribute, 57
EGL_NO_CONTEXT error, 58
EGL_NOT_INITIALIZED error, 46–47
EGLSurface, 62–63
eglSwapBuffers function, 41
EGL_TEXTURE_FORMAT attribute, 57
EGL_TEXTURE_TARGET attribute, 57
eglWaitClient function, 66–67
EGL_WIDTH attribute, 57
Element buffer objects, 140–141
Emulating OpenGL ES 3.0
ARM Mali OpenGL ES Emulator, 448
iOS 7, 453–455
OpenGL ES 3.0 Emulator, 447–449
PowerVR Insider SDK v 3.2+, 448
PowerVR OpenGL ES 3.0 Emulator, 449–450
Qualcomm Adreno SDK v3.4+, 447
Ubuntu Linux, 449–450
Windows, 447–449
enable behavior, 116
Entity names, querying, 429–435
Environment mapping
definition, 228, 370
element, 370
fragment shader, 372–373
vertex shader, 370–372
equal function, 473
Error checking, querying for error codes, 45
Error codes. See also specific codes.
querying for, 45
summary of, 23
Error handling, 22–23
ES Framework API
core functions, 485–489
esCreateWindow function, 34, 485–486
esFrustum function, 198, 490
esGenCube function, 489
esGenSphere function, 488
esGenSquareGrid function, 489
esLoadProgram function, 487–488
esLoadShader function, 487
esLoadTGA function, 488
esLogMessage function, 489
esMatrixLoadIdentity function, 493
esMatrixMultiply function, 493
esOrtho function, 491
esPerspective function, 198–199, 491
esRegisterDrawFunc function, 486
esRegisterKeyFunc function, 487
esRegisterShutdownFunc function, 487
ES Framework API (cont.)
  esRegisterUpdateFunc function, 486
  esRotate function, 492–493
  esScale function, 492
  esTranslate function, 492
  transformation functions, 490–494
  esCreateWindow function, 34, 485–486
  esFrustum function, 198, 491
  esGenCube function, 489
  esGenSphere function, 488
  esGenSquareGrid function, 489
  esLoadProgram function, 487–488
  esLoadShader function, 487
  esLoadTGA function, 488
  esLogMessage function, 489
  esMain function, 34
  esMatrixLoadIdentity function, 493
  esMatrixMultiply function, 493
  esOrtho function, 198, 491
  esPerspective function, 198–199, 491
  esRegisterDrawFunc function, 486
  esRegisterKeyFunc function, 487
  esRegisterShutdownFunc function, 487
  esRegisterUpdateFunc function, 486
  esRotate function, 492–493
  esScale function, 492
  esTranslate function, 492
  esUtil library, creating EGL windows, 65–66
  ETC/EAC texture compression, 12, 264–265
  Example code. See also specific examples.
  creating. See Hello Triangle.
  downloading, 28–29
  exp function, 466
  exp2 function, 466
  Exponential built-in functions, 466–467
  Extension behaviors, 116–117
  Extensions, 116–117

F
  faceforward function, 473
  FBOs (framebuffer objects). See also
    Renderbuffer objects.
    attachment points, 336–337
    binding, 335–336
    bits, 342–344
    checking for completeness, 341–342
    copying pixels, 342–344
    creating, 329–330
    definition, 327
    deleting, 346–347
    vs. EGL surfaces, 329
    examples, 348–354
    invalidation, 344–346
    new features, 17
    purpose of, 325–327
    querying, 445–446
    vs. renderbuffer objects, 328
    resolving multisample renderbuffers to
      textures, 342–344
    state values, 336
    TBR GPUs, 345
  FBOs (framebuffer objects), attachments
    2D textures, 338–339
    3D textures, 339–340
    depth buffers, 337–338
    renderbuffer objects, 337–338, 347
  Fences, 358–361
  Filtering textures. See Texture filtering.
  Flat/smooth interpolators, 14, 114
  floatBitsToInt function, 470
  floatBitsToUInt function, 470
  Floating-point
    numbers. See GL_HALF_FLOAT data
      type.
    pack and unpack, built-in functions,
      471–472
    texture formats, 249–250
    textures, new features, 12
  floor function, 467
  Fog effects. See also Particle systems.
    linear fog, creating with a fragment
      shader, 288–291
    wispy fog, creating with noise, 402–404
  fract function, 468
  Fragment depth, new features, 15
  Fragment processing, built-in functions,
    483–484
  Fragment shaders
    2D texturing, 255–257
    built-in constants, 284–285
    built-in special variables, 283–284
    creating, example, 35–36
    examples, 9, 111–114
    fragment depth, overriding, 284
    front-facing fragments, identifying, 284
    input variables, 8

Index
inputs, 8
inputs/outputs, 111–114
maximum uniform blocks, querying, 91
MRTs (multiple render targets),
  minimum/maximum number
  of, 285
offsets, minimum/maximum, 285
overview, 8–9, 282–285
precision qualifiers, 285
samplers, 8
shader inputs, minimum/maximum
  number of, 284
shader program, 8
Shading Language version, specifying, 9
texture coordinates for point sprites, 284
texture image units, minimum/
  maximum number of, 285
uniforms, 8
vec4 uniform entries, minimum/
  maximum number of, 285
window coordinates of current fragment,
  283–284
Fragment shaders, fixed-function
  techniques
alpha test, 291–293
combining texture maps, 286–287
fog effects, 288–291
multitexturing, 286–287
pipeline description, 280–282
transparent fragments, 291–293
user clip panes, 293–295
Fragments
blending pixel colors, 311–314
centroid sampling, 316
color depth, simulating, 314
deepth, overriding, 284
dithering, 314
front-facing, identifying, 284
MRTs (multiple render targets), 320–324
multi-sampled anti-aliasing, 314–316
pixel pack buffer objects, 320
pixels, reading and writing, 316–320
rendered images, saving, 316–320
sample coverage masks, 315
transparent, 291–293
window coordinates of, 283–284
Fragments, buffers
back, 298
buffer write masks, 301–303
clearing, 299–301
depth of, 298
double buffering, example, 41
front, 298
making writable, 301–303
requesting, 299
size of, 298
swapping, 298–299
types of, 298. See also specific types.
Fragments, tests
depth buffer test, 311
overview, 303–304
scissor test, 304–305
stencil buffer test, 305–311
test enable tokens, 304
Framebuffer invalidation hints, 17, 344–345
Framebuffer objects (FBOs). See FBOs
  (framebuffer objects).
Front buffers, 298
Frustrum, 7
Full integer support, new
  features, 14
Functions. See also Built-in functions; ES
   Framework API; specific functions.
description, 106
   passing parameters to, 106
   recursion, 106
fwidth function, 484
G
Gamma-correct rendering, new features, 11,
  254. See also sRGB textures.
Geometric built-in functions, 472–473
Geometry, new features, 15. See also
   Primitives.
Geometry instancing, 169–172
GL_ACTIVE_ATTRIBUTE_MAX_ LENGTH, 77
GL_ACTIVE_ATTRIBUTES, 77, 93
glActiveTexture function, 256
GL_ACTIVE_UNIFORM BLOCK_ MAX_LENGTH, 77
GL_ACTIVE_UNIFORM_BLOCKS, 77
GL_ACTIVE_UNIFORM_MAX_LENGTH, 77
GL_ACTIVE_UNIFORMS, 77
GL_ARRAY_BUFFER token, 140–141
GL_ATTACHED_SHADERS, 77
glAttachShader function, 75
glBeginQuery command, 184
glBeginTransformFeedback command, 213
glBindAttribLocation command, 139
<table>
<thead>
<tr>
<th>Function/Command</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>glBindBuffer command</td>
<td>142–143, 212</td>
</tr>
<tr>
<td>glBindBufferBase function</td>
<td>91, 212</td>
</tr>
<tr>
<td>glBindBufferRange function</td>
<td>91, 212</td>
</tr>
<tr>
<td>glBindFramebuffer</td>
<td>335–336</td>
</tr>
<tr>
<td>glBindRenderbuffer function</td>
<td>144</td>
</tr>
<tr>
<td>glBindSampler function</td>
<td>274–275</td>
</tr>
<tr>
<td>glBindTextures function</td>
<td>231</td>
</tr>
<tr>
<td>glBindVertexArray function</td>
<td>151</td>
</tr>
<tr>
<td>GL_BLEND token</td>
<td>304</td>
</tr>
<tr>
<td>glBlendColor function</td>
<td>313</td>
</tr>
<tr>
<td>glBlendEquation function</td>
<td>313–314</td>
</tr>
<tr>
<td>glBlendEquationSeparate function</td>
<td>313–314</td>
</tr>
<tr>
<td>glBlendFunc function</td>
<td>312–313</td>
</tr>
<tr>
<td>glBlendFuncSeparate function</td>
<td>312–313</td>
</tr>
<tr>
<td>glBlitFramebuffer command</td>
<td>343–344</td>
</tr>
<tr>
<td>glBufferData command</td>
<td>144</td>
</tr>
<tr>
<td>GL_BUFFER_SIZE</td>
<td>143</td>
</tr>
<tr>
<td>glBufferSubData command</td>
<td>145</td>
</tr>
<tr>
<td>GL_BUFFER_USAGE</td>
<td>143</td>
</tr>
<tr>
<td>glCheckFramebufferStatus command</td>
<td>342</td>
</tr>
<tr>
<td>GL_CLAMP_TO_EDGE mode</td>
<td>243–244</td>
</tr>
<tr>
<td>glClear function</td>
<td>40</td>
</tr>
<tr>
<td>glClear* functions</td>
<td>299–300</td>
</tr>
<tr>
<td>glClientWaitSync function</td>
<td>359–360</td>
</tr>
<tr>
<td>glColorMask function</td>
<td>302</td>
</tr>
<tr>
<td>glCompileShader function</td>
<td>37, 71–72</td>
</tr>
<tr>
<td>glCompressedTexImage* functions</td>
<td>267–278</td>
</tr>
<tr>
<td>glCompressedTexImage2D function</td>
<td>263–264</td>
</tr>
<tr>
<td>glCompressedTexImage3D function</td>
<td>263–264</td>
</tr>
<tr>
<td>glCompressedTexSubImage2D function</td>
<td>267</td>
</tr>
<tr>
<td>GL_COMPRESSED_TEXTURE_FORMATS</td>
<td>265</td>
</tr>
<tr>
<td>glCopyBufferSubData function</td>
<td>159–160</td>
</tr>
<tr>
<td>glCopyTexImage2D function</td>
<td>270–272</td>
</tr>
<tr>
<td>glCopyTexSubImage2D function</td>
<td>270–272</td>
</tr>
<tr>
<td>glCopyTexSubImage3D function</td>
<td>270–272</td>
</tr>
<tr>
<td>glCreateProgram function</td>
<td>74–75</td>
</tr>
<tr>
<td>glCreateShader function</td>
<td>36–37, 70–71</td>
</tr>
<tr>
<td>glCullFace command</td>
<td>181</td>
</tr>
<tr>
<td>GL_CULL_FACE state</td>
<td>181</td>
</tr>
<tr>
<td>GL_DECR operation</td>
<td>306</td>
</tr>
<tr>
<td>GL_DECR_WRAP operation</td>
<td>306–307</td>
</tr>
<tr>
<td>glDeleteBuffers command</td>
<td>150</td>
</tr>
<tr>
<td>glDeleteFramebuffers command</td>
<td>346–347</td>
</tr>
<tr>
<td>glDeleteProgram function</td>
<td>75</td>
</tr>
<tr>
<td>glDeleteQueries command</td>
<td>184</td>
</tr>
<tr>
<td>glDeleteRenderbuffers command</td>
<td>346–347</td>
</tr>
<tr>
<td>glDeleteSamplers function</td>
<td>273–274</td>
</tr>
<tr>
<td>glDeleteShader function</td>
<td>70–71</td>
</tr>
<tr>
<td>GL_DELETE_STATUS</td>
<td>77</td>
</tr>
<tr>
<td>glDeleteSync function</td>
<td>359</td>
</tr>
<tr>
<td>glDeleteTextures function</td>
<td>230–231</td>
</tr>
<tr>
<td>glDeleteVertexArrays command</td>
<td>154</td>
</tr>
<tr>
<td>glDepthFunc function</td>
<td>311</td>
</tr>
<tr>
<td>glDepthMask function</td>
<td>302</td>
</tr>
<tr>
<td>gl_DepthRange uniform type</td>
<td>190</td>
</tr>
<tr>
<td>gl_DepthRangeUniform type</td>
<td>179</td>
</tr>
<tr>
<td>gl_DepthRangeParameters uniform type</td>
<td>132–135</td>
</tr>
<tr>
<td>GL_DEPTH_TEST token</td>
<td>304, 311</td>
</tr>
<tr>
<td>glDetachShader function</td>
<td>75</td>
</tr>
<tr>
<td>glDisable command</td>
<td>23–24</td>
</tr>
<tr>
<td>glDisable function</td>
<td>437–438</td>
</tr>
<tr>
<td>glDisableVertexAttribArray command</td>
<td>132–135</td>
</tr>
<tr>
<td>GL_DITHER token</td>
<td>304</td>
</tr>
<tr>
<td>glDrawArrays function</td>
<td>40–41, 165–168, 341</td>
</tr>
<tr>
<td>glDrawArraysInstanced command</td>
<td>165–168, 170–172</td>
</tr>
<tr>
<td>glDrawBuffers function</td>
<td>321–322</td>
</tr>
<tr>
<td>glDrawElements function</td>
<td>165–168, 172–174, 341</td>
</tr>
<tr>
<td>GL_DYNAMIC_COPY</td>
<td>143</td>
</tr>
<tr>
<td>GL_DYNAMIC_DRAW</td>
<td>143</td>
</tr>
<tr>
<td>GL_ELEMENT_ARRAY_BUFFER token</td>
<td>140–141</td>
</tr>
<tr>
<td>glEnable function</td>
<td>23–24, 437–438</td>
</tr>
<tr>
<td>glEnableVertexAttribArray command</td>
<td>132–135</td>
</tr>
<tr>
<td>glEndQuery command</td>
<td>184</td>
</tr>
</tbody>
</table>
Index
GL_INVALID_OPERATION code, 23
GL_INVALID_VALUE code, 23
GL_INVERT operation, 307
gIs* functions, 436
gIsEnabled function, 24, 437
GL KEEP operation, 307
GL_LINE_LOOP, 163
GL_LINES, 163
GL_LINES mode, 213
GL_LINE_STRIP, 163
gLineWidth API call, 164
gLinkProgram command, 212
gLinkProgram function, 75–76
GL_LINK_STATUS, 95
gMapBufferRange command, 155–157
gl_MaxCombinedTextureImageUnits constant, 190
GL_MAX_COMBINED_UNIFORM_BLOCKS, 91
gl_MaxDrawBuffers constant, 285
gl_MaxFragmentInputVectors constant, 284
GL_MAX_FRAGMENT_UNIFORM_BLOCKS, 91
GL_MAX_FRAGMENT_UNIFORM VECTORS, 109
gl_MaxFragmentUniformVectors constant, 285
gl_MaxFragmentUniformVectors variable, 109
gl_MaxProgramTexelOffset constant, 285
gl_MaxTextureImageUnits constant, 285
GL_MAX_UNIFORM_BLOCK_SIZE, 92
gl_MaxVertexAttribs constant, 190
gl_MaxVertexAttribs variable, 112
GL_MAX_VERTEX_ATTRIBS variable, 112
GL_MAX_VERTEX_OUTPUT_COMPONENTS variable, 113
gl_MaxVertexAttribs constant, 190
gl_MaxVertexAttribs variable, 113
gl_MaxVertexTextureImageUnits constant, 190
GL_MAX_VERTEX_TEXTURE_UNITS, 213
GL_MAX_VERTEX_UNIFORM_BLOCKS, 91
GL_MAX_VERTEX_UNIFORM_VECTORS, 109
gl_MaxVertexUniformVectors constant, 190, 193–196
gl_MaxVertexUniformVectors variable, 109
gl_MinProgramTexelOffset constant, 285
GL_MIRRORED_REPEAT mode, 243–244
GL_NO_ERROR code, 23
GL_OUT_OF_MEMORY code, 23
gPixelStorei function, 235
GL_PIXEL_PACK_BUFFER, 277–278
gl_PointCoord variable, 164–165, 284
GL_POINTS mode, 164–165, 213
gl_PointSize variable, 164, 190
gPolygonOffset command, 182–183
gl_Position variable, 190
gProgramBinary function, 94
GL_PROGRAM_BINARY_RETRIEVABLE_HINT, 77
GL_RASTERIZER_DISCARD, 214
gReadBuffer function, 269–270
gReadPixels function, 316–320, 346–347
GL_RESCALE_NORMAL mode, 206
gReleaseShaderCompiler function, 93
gRenderbufferStorage function, 331–332
gRenderbufferStorage Multisample function, 331–332
GL_REPEAT mode, 243–244
GL_REPLACE operation, 306
GL_SAMPLE_ALPHA_TO_COVERAGE token, 304
gSampleCoverage function, 315–316
GL_SAMPLE_COVERAGE token, 304
gScissor test, 304–305
GL_SEPARATE_ATTRIBS, 77
gShaderSource function, 37, 71
GL_SPHERE_MAP mode, 206
GL_STATIC_COPY, 143
GL_STATIC_DRAW, 143
GL_STATIC_READ, 143
gStencilFunc function, 305–311
gStencilFuncSeparate function, 305–306
gStencilMask function, 302–303

Index
glStencilMaskSeparate function, 303
glStencilOp function, 306–311
GL_STENCIL_TEST token, 304
GL_STREAM_COPY, 144
GL_STREAM_DRAW, 144
GL_STREAM_READ, 144
gTexImage* functions, 277–278
gTexImage2D function, 231–234
gTexImage3D function, 260–262
gTexParameter* commands
  API overhead, 273
  setting minification/magnification
  filtering modes, 236, 239–240
  texture coordinate wrapping, 243
  texture detail level, setting, 245
gTexStorage2D function, 276–277
gTexStorage3D function, 276–277
gTexSubImage* functions, 277–278
gTexSubImage2D function, 266–267
gTexSubImage3D function, 267–269
GL_TEXTURE_BASE_LEVEL parameter, 245
GL_TEXTURE_COMPARE_FUNC parameter, 245–246
GL_TEXTURE_COMPARE_MODE parameter, 245–246
GL_TEXTURE_MAX_LOD parameter, 245
GL_TEXTURE_MIN_LOD parameter, 245
GL_TEXTURE_SWIZZLE_A parameter, 244–245
GL_TEXTURE_SWIZZLE_B parameter, 244–245
GL_TEXTURE_SWIZZLE_G parameter, 244–245
GL_TEXTURE_SWIZZLE_R parameter, 244–245
GL_TRANSFORM_FEEDBACK_VARYINGS, 77
GL_UNIFORM_BLOCK_ACTIVE_NUMBER_INDICES, 90
GL_TRIANGLES mode, 213
GL_TRIANGLES, 162
GL_UNIFORM_BLOCK_ACTIVE_UNIFORMS, 90
GL_UNIFORM_BLOCK_BINDING, 90
gUniformBlockBinding function, 90–91
GL_UNIFORM_BLOCK_DATA_SIZE, 90
GL_UNIFORM_BLOCK_NAME_LENGTH, 90
GL_UNIFORM_BLOCK_REFERENCED_BY_VERTEX_SHADER, 90
GL_UNIFORM Block REFERENCED BY_FRAGMENT_SHADER, 90
gUnmapBuffer command, 156–157
gluPerspective function, see esPerspective function
gUseProgram function, 39, 78
GL_VALIDATE_STATUS, 77
GL_VALIDATE_STATUS, 77
glTransformFeedbackVaryings command, 170–172
GL_VERTEX_ATTRIB_POINTER function, 40, 131–132
GL_VERTEX ID variable, 189
glViewport command, 39, 178–179
GLvoid data type, 21
gWaitSync function, 360
glZero operation, 306
greaterThan function, 475
greaterThanEqual function, 475
Guard band region, 177

H
Hello Triangle
  back buffer, displaying, 41
  code framework, 28
  color buffer, clearing, 39–40
double buffering, 41
drawing fragments, 35–36
  geometry, loading, 40–41
OpenGL ES 3.0 framework, 34–35
  primitives, drawing, 40–41
  programs objects, 38–39
source code, 29–33
transforming vertices, 35–36
viewport, setting, 39–40
windows, creating, 34–35

Index
Hello Triangle, shaders
  compiling and loading, 36–38
  creating, 35–36
  fragment, creating, 35–36
  linking, 38–39
  vertex, creating, 35–36
highp keyword, 120, 192–193
Hints, 77, 435–436

I
if-then-else tests. See Control flow statements.

Images
  dimensions, specifying, 331
  format, specifying, 331
Images, postprocessing
  blur effect, 387–390
  light bloom, 389–390
  render-to-texture setup, 387
Immutable textures, 13, 276–277
in qualifier, 106
Include files, EGL, 20
info logs, 77–78
Initializing
  arrays, 104
  buffer objects, 145
  EGL, 44, 46
  scalar data types, 100
  structures, 103
  vector data types, 100
  vertex array objects, 144
inout qualifier, 106
Input variables, fragment shader, 8
Instance ID, new features, 15
Instanced rendering, new features, 15
intBitsToFloat function, 470
Integer texture formats, 250–252
Integer textures, new features, 12
Interpolation, 114–115
Interpolation qualifiers
  centroid sampling, 115
  default behavior, 114
  flat shading, 114
  smooth shading, 114
Interpolators
  definition, 113
  packing, 117–119
Invariance, 121–123
invariant keyword, 121–123
inverse function, 474
inverseSqrt function, 467
iOS 7, 453–455
isinf function, 470
isnan function, 470

J
JDK (Java SE Development Kit) 7, 451

K
Keywords
  centroid, 115
  highp, 120, 192–193
  invariant, 121–123
  lowp, 120, 192–193
  mediump, 120, 192–193

L
Latitude-longitude maps, example, 205–206
Layout qualifiers, 14, 109–110
length function, 473
Lens flare effects, 183–185
lessThan function, 475
lessThanEqual function, 475
Libraries, EGL, 20
Light bloom, 389–390
Light projection matrix, 392–393
Light view matrix, 392–393
Lighting
  equations, 369–370
  example, 199–205
Lighting, per fragment
  lighting equations, 369–370
  lighting shaders, 366–369
  with a normal map, 364–365
  overview, 363–364
Lighting shaders, 366–369
Lines
  clipping, 177
  description, 163–164
  width, specifying, 164
Linking shaders, example, 38–39
Loading
  2D texture arrays, 260–262
  3D textures, 260–262
  geometry, example, 40–41
  shaders, 36–38, 73–74
  shaders, example, 36–38
  texture objects, 231–234
  textures, 230–236
  uniforms, 83–85
LoadShader function, 36–37
log function, 466
log2 function, 467
Loops. See Control flow statements.
lowp keyword, 120, 192–193

M
Macros, defining, 115–117
Magnification filtering mode, 236, 238–241
main function, vertex shader, 6–7
Mandatory online compiler, new features, 14
Mapping, texture formats to colors, 257
Mapping buffer objects
  changing screen resolution, 157
  data storage pointer, getting, 155–156
  flushing mapped buffers, 158–159
  overview, 154–155
  unmapping, 155–156
Matrices
  non-square, new features, 14
  projective texturing, 392–393
Matrix built-in functions, 474
Matrix components, 101–102
Matrix construction, 101
Matrix data types, 99–100
Matrix transformations, example, 196–199
matrixCompMult function, 474
max function, 468
mediump keyword, 120, 192–193
Meshes, connecting, 172
mix function, 469
mod function, 468
Model matrix, example, 197–198
Motion blur effects. See Rendering, to
  textures.
MRTs (multiple render targets)
  deferred shading, 320–321
  in fragment shaders, 285
  in fragments, 320–324
  new features, 17
  overview, 320
  setting up, 322–324
  specifying color attachments, 321–322

N
Naming conventions, 102
Nearest sampling, 237
New features, buffer objects
  buffer object to buffer object
    copies, 16
  buffer subrange mapping, 16
  pixel buffer objects, 16
  sampler objects, 16
  sync objects, 16
  uniform buffer objects, 16
  vertex array objects, 16
New features, framebuffer
  blend equations, 17
  framebuffer invalidation hints, 17
  min/max functions, 17
  MRTs (multiple render targets), 17
  multisample renderbuffers, 17
  off-screen rendering, 17
New features, geometry
  Boolean occlusion queries, 15
  instanced rendering, 15
  new vertex formats, 15
  primitive restart, 15
  transform feedback, 15
New features, shaders
  built-in functions, 15
  centroid sampling, 14
  flat/smooth interpolators, 14
  fragment depth, 15
  full integer support, 14
  instance ID, 15
  layout qualifiers, 14
  mandatory online compiler, 14
  non-square matrices, 14
  program binaries, 13–14
  relaxed restrictions, 15
  uniform blocks, 14
  vertex ID, 15
New features, texturing
  2D texture arrays, 11
  3D textures, 12
  depth textures, 12
  ETC/EAC texture compression, 12
  floating-point textures, 12
  gamma-correct rendering, 11
  immutable textures, 13
New features, texturing (cont.)
increased minimum sizes, 13
integer textures, 12
NPOT (non-power-of-2) textures, 13
seamless cubemap filtering, 12, 241
shadow comparisons, 12
sRGB textures and framebuffers, 11
texture LOD (level of detail)
features, 13
texture swizzles, 13
texturing, 11–13
vendor-specific compressed texture formats, 12
New features, vendor-specific compressed texture formats, 12
New vertex formats, new features, 15
Noise, 3D texture
dust effects, 403–404
example, 397
generating, 397–402
water wave simulation, 404
wispy fog, 402–404
noise3D function, 401
Non-square matrices, new features, 14
normalize function, 473
Normalized flag, 131–132
Normalized texture formats, 247–248
not function, 475
notEqual function, 475
NPOT (non-power-of-2) textures, new features, 13
new features. See New features.
platforms. See Emulating OpenGL ES 3.0.
specifications, 2–3
OpenGL ES 3.0, graphics pipeline. See also specific components.
diagram, 4
fragment shader, 8–9
per-fragment operations, 9–11
primitive assembly, 7
rasterization, 7
vertex shader, 4–7
OpenGL ES 3.0 Emulator, 447–449.
See also Emulating OpenGL ES 3.0.
OpenGL ES Shading Language 3.0, specifications, 4
Operators, 104–105
out qualifier, 106
outerProduct function, 474
Overlapping polygons, 181–183

P
packed qualifier, 110
packHalf2x16 function, 472
Packing
interpolators, 117–119
uniforms, 117–119
packSnorm2x16 function, 471
Particle emissions, 381–385
Particle systems
fragment shader, 377–379
particle emissions, 381–385
point sprites, 374
rendering algorithm, 381
rendering particles, 385–386
setup, 374–375
transform feedback, 380, 381–385
vertex shader, 375–377
Pbuffers (pixel buffers)
attributes, 57
creating, 56–60
description, 56
errors, 58
PCF (percentage closest filtering), 245–246, 414
Per fragment lighting
lighting equations, 369–370
lighting shaders, 366–369
with a normal map, 364–365
overview, 363–364
Per-fragment operations
- blending, 10
- depth test, 10
- dithering, 10
- overview, 9–11
- scissor test, 10
- stencil test, 10

Performance
- drawing primitives, 172–174
- FBOs (framebuffer objects), 354
- hints, 435–436
- primitives, drawing, 172–174
- vertex attributes, storing, 131–135

Perspective division, 178

Pixel buffer objects
- new features, 16
- pixel pack buffer objects, 320
- pixel unpack buffer objects, 277–278

Pixel buffers (pbuffers)
- attributes, 57
- creating, 56–60
- description, 56
- errors, 59

Pixel pack buffer objects, 320

Pixel unpack buffer objects, 277–278

Pixels
- copying in framebuffer objects, 342–344
- in fragments, reading and writing, 316–320
- reading in framebuffer objects, 347
- storage options, 236
- texels (texture pixels), 226–227

Point light, example, 202

Point sampling, 237

Point sprites
- clipping, 177
- description, 164–165
- position, 164
- radius, 164
- texture coordinates for, 284

Point sprites in particle systems, 374

Polygons
- joins, smoothing (example), 207–211
- offsetting, 181–183
- overlapping, 181–183

Position, point sprites, 164

Postprocessing effects. See Rendering, to textures.

pow function, 466

PowerVR Insider SDK v 3.2+, 448

PowerVR OpenGL ES 3.0 Emulator, 449–450

Precision qualifiers
- default precision, 120
- variables, 119–120
- vertex shaders, 119–120, 192–193

Preprocessor directives. See also specific directives.
- conditional tests, 115–117
- description, 115–117

Primitive assembly
- culling primitives, 7
- description, 174–175
- perspective division, 178
- view frustrum, 7
- viewport transformation, 178–179

Primitive assembly, coordinate systems
- clipping, 176–177
- guard band region, 177
- overview, 175

Primitive restart, 15, 168–169

Primitives. See also Geometry, new features.
- definition, 7, 161
- drawing, 7.
- example, 40–41
- geometry instancing, 169–172
- multiple disconnected primitives, 168–169
- multiple primitives, different attributes, 169–172
- overview, 165–168
- performance tips, 172–174
- primitive restart, 168–169
- provoking vertex, 168–169

Procedural textures
- anti-aliasing, 407–410
- checkerboard example, 405–407
- example, 405–407
- pros and cons, 404

Program binaries
- compatibility check, 95
- definition, 94
- format, 95
- getting, 94
- new features, 13–14
- saving, 94
Program objects. See also Shader objects; Shaders.
- attached shaders, counting, 77
- attaching shaders, 75, 79
- creating, 74–79
- definition, 69–70
- deleting, 75
- deletion status, querying, 77
- detaching shaders, 75
- linking, 74–79
- making active, 78
- validating, 78

Projection matrix, example, 198–199

Projective texturing
- basic techniques, 391–392
- bias matrix, 392–393
- definition, 391
- light projection matrix, 392–393
- light view matrix, 392–393
- matrices, 392–393
- overview, 390–391
- spotlight shaders, 394–397

Provoking vertex, 168–169

Q
Qualcomm Adreno SDK v3.4+, 447

Qualifiers
- column_major, 110
- in, 106
- inout, 106
- out, 106
- packed, 110
- row_major, 110
- shared, 110
- std140, 110

Queries. See State queries.

R
radians function, 465
Radius, point sprites, 164
Rasterization
- culling, 180–181
- enabling/disabling, 214
- pipeline, 179
- polygon offset, 181–183

Recursion, in functions, 106
reflect function, 473

Reflective surfaces, 205–206. See also
- Environment mapping; Projective texturing; Rendering, to textures.
- refract function, 473

Renderbuffer objects. See also FBOs (framebuffer objects).
- attached to framebuffer objects, 337–338, 347
- binding, 330–331
- creating, 329–330
- default values, 331
- definition, 327
- deleting, 346–347
- vs. FBOs (framebuffer objects), 328
- formats, 333–335
- image dimensions, specifying, 331
- image format, specifying, 331
- multisample, 333
- state values, 331
- vs. textures, 328

Renderbuffers
- multisample, new features, 17
- querying, 445–446

Rendering
- from eye position with depth texture, 418–420
- gamma-correct, new features, 11
- instanced, new features, 15
- from light position into depth texture, 415–418
- off-screen area. See Pbuffers (pixel buffers).
- on-screen area. See Windows.
- particles, 381, 385–386
- rendered images, saving, 316–320
- shadows with depth texture, 414–420
- synchronizing, 66–67
- terrain with vertex texture fetch, 410–414

Rendering, to off-screen surfaces. See also FBOs (framebuffer objects); Renderbuffer objects.
- basic techniques, 326–327
- new features, 17

Rendering, to textures. See also FBOs (framebuffer objects).
- basic techniques, 326–327
- examples, 348–354
- uses for, 326
- while using the texture object in a fragment shader, 341

Rendering context, creating, 19, 60–62, 325–327. See also EGL; FBOs (framebuffer objects).
Index

Rendering surfaces, creating with EGL, 19. See also EGL.
require behavior, 116
round function, 468
roundEven function, 468
row_major qualifier, 110

S
Sample coverage masks, 315
Sampler objects, 16, 273–275
Samplers
definition, 256
fragment shader, 8
querying, 442
vertex shader, 4
Scalar data types
description, 99–100
initializing, 100
type conversion, 100
Scissor test, 10, 304–305
Screen resolution, effect on mapped buffer objects, 157
Seamless cubemap filtering, new features, 12, 241
Shader compiler, 93
Shader objects, 69–70. See also Program objects; Shaders.
Shaders. See also Fragment shaders; Vertex shaders.
2D textures, 255–257
attached to programs, querying, 438–440
compiling, 70–74
creating, 70–74
deleting, 70
info log, retrieving, 72–73
linking, 70
loading, 73–74
source, providing, 71
texturing, 255–257
version specification, declaring, 98
Shaders, new features
built-in functions, 15
centroid sampling, 14
flat/smooth interpolators, 14
fragment depth, 15
full integer support, 14
instance ID, 15
layout qualifiers, 14
mandatory online compiler, 14
non-square matrices, 14
program binaries, 13–14
relaxed restrictions, 15
uniform blocks, 14
vertex ID, 15
Shading Language version, specifying in fragment shaders, 9
Shadow comparisons, new features, 12
Shadow mapping, 245–246. See also Projective texturing; Rendering, to textures.
Shadows, rendering, 414–420
Shared exponent texture formats, 252–253
shared qualifier, 110
Shimmering. See Z fighting.
Shiny surfaces, example, 205–206
sign function, 467
Signaling sync objects, 359–360
sin function, 465
sinh function, 466
Smoke effects. See Particle systems.
Smooth shading, 114
smoothstep function, 470
Specifications, OpenGL ES
1.0, 2
1.1, 2
2.0, 2–3
3.0, 2–3
3.0 API, 3–4
Shading Language 3.0, 4
Sphere maps, example, 205–206
Spotlight, example, 202–205
Spotlight shaders, 394–397
sqrt function, 467
Square brackets [ ], array subscript operator, 101–102
sRGB textures, 11, 254. See also Gamma-correct rendering.
Stairstep effects. See Anti-aliasing; Mipmapping.
State management
checking current state, 24
enabling/disabling state, 23–24
overview, 23–24
querying state values, 24
State queries
application-modifiable queries, 429–435
asynchronous objects, 442–443
entity names, 429–435
framebuffer, 445–446
implementation-dependent limits, 423–428
State queries (cont.)
  nonprogrammable operations control, 436–438
OpenGL ES 3.0 implementation string queries, 421–422
renderbuffer, 445–446
samplers, 442
shaders attached to programs, 438–440
sync objects, 443
texture state, 441–442
vertex attributes, 440–441
vertex buffers, 444
std140 qualifier, 88–89, 110
Stencil buffer test, 305–311
Stencil buffers
  buffer write masks, 303
  fragment operations, 298–299, 303.
  See also Fragments, buffers.
  sharing, 329
Stencil test, per-fragment operations, 10
step function, 470
Structures, 103. See also Arrays of structures.
Structures of arrays, 128. See also Arrays.
Surface configurations
  available, determining, 46–47
  choosing with EGL, 51–53
Swapping, buffers, 298–299
Swizzles. See Texture swizzles.
Sync objects
  creating, 358–359
  deleting, 358–359
  example, 360–361
  fences, 358–361
  new features, 16
  overview, 357–358
  querying, 443
  signaling, 359–360
  waiting for, 359–360
Synchronizing rendering, 66–67

T
tan function, 465
tanh function, 466
Terrain surfaces, 214–215, 410–414
Test enable tokens, 304
Tests, fragments
  depth buffer test, 311
  overview, 303–304
  scissor test, 304–305
  stencil buffer test, 305–311
  test enable tokens, 304

Texels (texture pixels), 226–227
texture built-in function, 257, 260, 478
Texture coordinates
  generating, example, 205–206
  wrapping, 243–244
Texture filtering
  magnification, 236, 238–241
  minification, 236, 238–241
  nearest sampling, 237
  overview, 237–241
  point sampling, 237
  seamless cubemap filtering, 241
Texture filtering, mipmapping
  automatic generation, 242
  detail levels, specifying, 245
  mipmap chains, 237–238
  overview, 237–241
Texture formats
  depth textures, 254–255
  floating-point, 249–250
  integer, 250–252
  mapping to colors, 257
  normalized, 247–248
  overview, 246–247
  shared exponent, 252–253
  sRGB, 254
  unsized, 247
Texture image units, in fragment shaders, 285
Texture LOD (level of detail) features, new features, 13
Texture lookup built-in functions, 476–482
Texture maps, combining, 286–287
Texture objects
  overview, 230–236
  pixel storage options, 236
Texture pixels (texels), 226–227
Texture state, querying, 441–442
Texture swizzles
  accessing vector components, 101
  new features, 13
  overview, 244–245
Texture units, specifying min/max number, 190
textureGrad function, 481
textureGradOffset function, 481
textureLod function, 479
textureOffset function, 480
textureProj built-in function, 391, 479
textureProjGrad function, 482
textureProjGradOffset function, 482
textureProjLod function, 480
textureProjLodOffset function, 480
Textures
color components, mapping. See Texture swizzles.
combining texture maps, 286–287
compressing, 262–265
compression formats, 264–265
copying from the color buffer, 269–273
immutable, 276–277
multitexturing, 286–287
vs. renderbuffer objects, 328
subimage selection, 266–269
textureSize function, 478
Texturing
depth texture compare, 245–246
fetching from a texture map, 256
loading textures, 230–236
PCF (percentage closest filtering), 245–246
pixel unpack buffer objects, 277–278
sampler objects, 273–275
samplers, 256
in shaders, 255–257
texels (texture pixels), 226–227
volume textures. See 3D textures.
Texturing, 2D texture arrays
loading, 260–262
overview, 230
Texturing, 2D textures
attached to framebuffer objects, 338–339
base formats, 227
overview, 226–227
in shaders, 255–257
Texturing, 3D textures
attached to framebuffer objects, 339–340
loading, 260–262
overview, 229
Texturing, cubemap textures
environment mapping, 228–229
overview, 228–229
Texturing, new features
2D texture arrays, 11
3D textures, 12
depth textures, 12
ETC/EAC texture compression, 12
floating-point textures, 12
gamma-correct rendering, 11
immutable textures, 13
increased minimum sizes, 13
integer textures, 12
NPOT (non-power-of-2) textures, 13
seamless cubemap filtering, 12, 241
shadow comparisons, 12
sRGB textures and framebuffers, 11
texture LOD (level of detail) features, 13
texture swizzles, 13
texturing, 11–13
vendor-specific compressed texture formats, 12
Texturing, texture objects
binding, 231
creating, 230
deleting, 230–231
loading, 231–234
minification/magnification filtering modes, setting, 236
overview, 230–236
pixel storage options, 236
3D textures
attached to framebuffer objects, 339–340
loading, 260–262
new features, 12
overview, 229
3D textures, noise
dust effects, 403–404
water wave simulation, 404
wispy fog, 402–404
Transform feedback
example, 211–214
new features, 15
in particle systems, 380–385
vertex shader, 5
Transformation functions, 490–494
Transforming vertices, example, 35–36
Transparent fragments, 291–293
transpose function, 474
Triangle fans, drawing, 162–163
Triangle strips
connecting, 172–174
drawing, 162–163
generating degenerate triangles, 15
primitive restart, 15
winding order, 174
Triangles
clipping, 177
culling, 180–181
degenerate, 15, 172
description, 162–163
drawing, 162–163
Trigonometry built-in functions, 465–466
\texttt{trunc} function, 467
2D texture arrays
loading, 260–262
new features, 11
overview, 230
2D textures
attached to framebuffer objects, 338–339
base formats, 227
noise, 402
overview, 226–227
in shaders, 255–257
Type conversion, 100

\textbf{U}
Ubuntu Linux, emulating OpenGL ES 3.0, 449–450
\texttt{uintBitsToFloat} function, 470
Uniform block indexes, associating with binding points, 90
Uniform blocks
active uniforms, counting, 90
description, 109–111
examples, 109–110
last buffer binding point, getting, 90
layout qualifiers, 109–110
maximum for all shaders, querying, 91
maximum per shader, querying, 91
minimum supported number, 91
minimum total buffer object size, 90
name length, getting, 90
new features, 14
references to, querying, 90
Uniform buffer objects. See also Buffer objects.
available storage, querying, 92
binding, 91
new features, 16
overview, 87–92
programming limitations, 91
Uniform names
largest, counting characters, 81
largest, getting, 77, 81
maximum length, querying, 77
Uniform variables. See Uniforms.
Uniforms
active, counting, 77
active, querying, 77, 86–87
constant store, 109
description, 80, 108–109
first category, 80
fragment shader, 8
getting, 81–87
indexes, getting, 89
loading, 83–85
maximum number in vertex shaders, 193–196
maximum number of, determining, 109
named uniform blocks, 80, 88
packing, 117–119
properties, getting, 81–87
setting, 81–87
sharing, 87–92
\texttt{std140} block layout, 88
vertex shader, 4
Unmapping mapped buffer objects, 155–156
\texttt{unpackHalf2x16} function, 472
\texttt{unpackSnorm2x16} function, 471
\texttt{unpackUnorm2x16} function, 471–472
Unsized texture formats, 247
Updating, buffer objects, 145
User clip panes, 293–295

\textbf{V}
Validating programs, description, 78
Validation status, querying, 77
Variable constructors, 100–101
Variables, 119–120. See also specific variables.
Varying variables. See Vertex shaders, output variables.
\texttt{vec4} uniform entries, in fragment shaders, 285
Vector components
accessing, 101–102
naming conventions, 102
Vector data types
description, 99–100
initializing, 100
type conversion, 100
Vector relational built-in functions, 475
Vertex array objects
binding, 151
creating, 144, 151
deleting, 154
drawing with, 152–154
initializing, 144
new features, 16
overview, 150–151
Vertex attribute variables, declaring in vertex shaders, 135–137
Vertex attributes
  active, listing, 136–137
  enabling/disabling, 132–135
  minimum number required, 126
  querying, 126, 440–441
Vertex attributes, binding
to attribute variables, 137–140
to locations, 139–140
  querying results of, 140
Vertex attributes, specifying
  client vertex arrays, 126–135
  constant vertex attributes, 126
  description, 126
Vertex attributes, storing. See also Arrays of structures; Structures of arrays.
  constant vertex attributes, 132–135
data conversions, 132
  normalized flag, 131–132
  performance tips, 131–135
  selecting a data format, 131
  vertex arrays, 132–135
Vertex buffer objects. See also Buffer objects.
  array buffer objects, 140–141
  binding, 141
  creating, 141
  element buffer objects, 140–141
  making current, 141
  overview, 140–141
  state, 143–144
  types of, 140–141
Vertex buffers, querying, 444
Vertex ID, new features, 15
Vertex shaders
  built-ins
    constants, 190–192
    special variables, 189–190
    uniform state, 190
Vertex shaders, examples
  creating vertex shaders, 35–36
cube maps, 205–206
directional light, 199–202
  generating texture coordinates, 205–206
  height value, fetching, 412–413
  latitude-longitude maps, 205–206
  lighting, 199–205
  matrix transformations, 196–199
  model matrix, 197–198
OpenGL ES 1.1 fixed-function vertex pipeline, 215–223
  point light, 202
  polygon joins, smoothing, 207–211
  projection matrix, 198–199
  reflective surfaces, 205–206
  shiny surfaces, 205–206
  sphere maps, 205–206
  spotlight, 202–205
  terrain surfaces, 214–215
  transform feedback, 211–214
vertex skinning, 207–211
view matrix, 197–198
Vertex shaders, overview
  entry point, 6–7
  example, 6
  inputs/outputs, 4–5
  main function, 6–7
  samplers, 4
  shader program, 4
  transform feedback, 5
  uniforms, 4
  vertex shader output variables, 5
Vertex skinning, example, 207–211
Vertex textures, 214–215
Vertices, transforming (example), 35–36
View frustrum, 7
View matrix, example, 197–198
Viewport, setting (example), 39–40
Viewport transformation, 178–179
Visual artifacts. See Mipmapping; Z-fighting artifacts.

W
Waiting for sync objects, 359–360
warn behavior, 117
Water surfaces, vertex shaders, 214–215
Index

Water wave simulation, 404
Winding order, triangle strips, 174
Windowing systems, communicating with, 44–45
Windows, 34–35. See also EGL windows.
Windows, emulating OpenGL ES 3.0, 447–449
Wrapping, texture coordinates, 243–244

Z
Z-fighting artifacts
avoiding, 121–123
polygon offset, 181–183