## Contents

1 Introduction
---
1.1 Formatting of the OpenGL Specification ........................................ 1
  1.1.1 Formatting of Changes .................................................. 1
1.2 What is the OpenGL Graphics System? ........................................ 1
1.3 Programmer’s View of OpenGL ................................................ 2
1.4 Implementor’s View of OpenGL .............................................. 2
1.5 Our View ............................................................................. 2
1.6 The Deprecation Model ......................................................... 3
1.7 Companion Documents ........................................................... 3
  1.7.1 OpenGL Shading Language ............................................... 3
  1.7.2 Window System Bindings ................................................. 3

2 OpenGL Operation
---
2.1 OpenGL Fundamentals ............................................................ 5
  2.1.1 Numeric Computation ..................................................... 7
  2.1.2 Fixed-Point Data Conversions .......................................... 11
2.2 GL State .............................................................................. 12
  2.2.1 Shared Object State ....................................................... 13
2.3 GL Command Syntax .............................................................. 13
  2.3.1 Data Conversion For State-Setting Commands ....................... 15
2.4 Basic GL Operation ............................................................... 17
2.5 GL Errors ............................................................................. 19
2.6 Primitives and Vertices .......................................................... 20
  2.6.1 Primitive Types ............................................................. 22
2.7 Vertex Specification ............................................................... 28
2.8 Vertex Arrays ...................................................................... 30
  2.8.1 Transferring Array Elements ........................................... 34
  2.8.2 Packed Vertex Data Formats .......................................... 34
  2.8.3 Drawing Commands ...................................................... 35
CONTENTS

2.9 Buffer Objects ........................................... 43
  2.9.1 Creating and Binding Buffer Objects ................. 44
  2.9.2 Creating Buffer Object Data Stores .................. 46
  2.9.3 Mapping and Unmapping Buffer Data .................. 48
  2.9.4 Effects of Accessing Outside Buffer Bounds .......... 53
  2.9.5 Copying Between Buffers ................................ 53
  2.9.6 Vertex Arrays in Buffer Objects .................... 53
  2.9.7 Array Indices in Buffer Objects .................... 54
  2.9.8 Indirect Commands in Buffer Objects ................. 55
  2.9.9 Buffer Object State .................................. 55

2.10 Vertex Array Objects ................................... 55

2.11 Vertex Shaders ........................................... 56
  2.11.1 Shader Objects ..................................... 58
  2.11.2 Loading Shader Binaries ............................ 60
  2.11.3 Program Objects ................................... 60
  2.11.4 Program Pipeline Objects ............................ 65
  2.11.5 Program Binaries ................................... 71
  2.11.6 Vertex Attributes .................................. 73
  2.11.7 Uniform Variables .................................. 77
  2.11.8 Subroutine Uniform Variables ....................... 100
  2.11.9 Samplers .......................................... 103
  2.11.10 Images ........................................... 104
  2.11.11 Output Variables .................................. 104
  2.11.12 Shader Execution .................................. 109
  2.11.13 Shader Memory Access .............................. 117
  2.11.14 Required State .................................... 122

2.12 Tessellation ............................................. 124
  2.12.1 Tessellation Control Shaders ......................... 125
  2.12.2 Tessellation Primitive Generation ................... 131
  2.12.3 Tessellation Evaluation Shaders ..................... 139

2.13 Geometry Shaders ....................................... 144
  2.13.1 Geometry Shader Input Primitives .................... 145
  2.13.2 Geometry Shader Output Primitives .................. 147
  2.13.3 Geometry Shader Variables ......................... 147
  2.13.4 Geometry Shader Execution Environment ............. 148

2.14 Coordinate Transformations .............................. 154
  2.14.1 Controlling the Viewport ............................ 154

2.15 Asynchronous Queries ................................... 157

2.16 Conditional Rendering ................................... 160

2.17 Transform Feedback ..................................... 161
2.17.1 Transform Feedback Objects .................................. 161
2.17.2 Transform Feedback Primitive Capture ...................... 163
2.17.3 Transform Feedback Draw Operations ......................... 167
2.18 Primitive Queries .................................................. 168
2.19 Flatshading .......................................................... 169
2.20 Primitive Clipping ................................................... 170
2.20.1 Clipping Shader Outputs ......................................... 172
3 Rasterization .............................................................. 174
3.1 Discarding Primitives Before Rasterization ...................... 175
3.2 Invariance ................................................................ 176
3.3 Antialiasing .............................................................. 176
3.3.1 Multisampling ........................................................ 177
3.4 Points .................................................................... 179
3.4.1 Basic Point Rasterization ........................................ 181
3.4.2 Point Rasterization State ........................................ 182
3.4.3 Point Multisample Rasterization ............................... 182
3.5 Line Segments .......................................................... 182
3.5.1 Basic Line Segment Rasterization .............................. 182
3.5.2 Other Line Segment Features .................................... 185
3.5.3 Line Rasterization State ......................................... 187
3.5.4 Line Multisample Rasterization ............................... 187
3.6 Polygons .................................................................. 188
3.6.1 Basic Polygon Rasterization ..................................... 188
3.6.2 Antialiasing .......................................................... 190
3.6.3 Options Controlling Polygon Rasterization ............... 190
3.6.4 Depth Offset .......................................................... 191
3.6.5 Polygon Multisample Rasterization ......................... 192
3.6.6 Polygon Rasterization State .................................... 193
3.7 Pixel Rectangles ........................................................ 193
3.7.1 Pixel Storage Modes and Pixel Buffer Objects ............ 194
3.7.2 Transfer of Pixel Rectangles .................................... 195
3.8 Early Per-Fragment Tests .............................................. 207
3.9 Texturing ................................................................. 208
3.9.1 Texture Objects ...................................................... 209
3.9.2 Sampler Objects ..................................................... 211
3.9.3 Texture Image Specification ..................................... 213
3.9.4 Alternate Texture Image Specification Commands ...... 227
3.9.5 Compressed Texture Images .................................... 232
3.9.6 Multisample Textures ............................................. 239
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.9.7</td>
<td>Buffer Textures</td>
<td>240</td>
</tr>
<tr>
<td>3.9.8</td>
<td>Texture Parameters</td>
<td>243</td>
</tr>
<tr>
<td>3.9.9</td>
<td>Depth Component Textures</td>
<td>245</td>
</tr>
<tr>
<td>3.9.10</td>
<td>Cube Map Texture Selection</td>
<td>245</td>
</tr>
<tr>
<td>3.9.11</td>
<td>Texture Minification</td>
<td>247</td>
</tr>
<tr>
<td>3.9.12</td>
<td>Texture Magnification</td>
<td>257</td>
</tr>
<tr>
<td>3.9.13</td>
<td>Combined Depth/Stencil Textures</td>
<td>257</td>
</tr>
<tr>
<td>3.9.14</td>
<td>Texture Completeness</td>
<td>257</td>
</tr>
<tr>
<td>3.9.15</td>
<td>Texture State and Proxy State</td>
<td>259</td>
</tr>
<tr>
<td>3.9.16</td>
<td>Immutable-Format Texture Images</td>
<td>261</td>
</tr>
<tr>
<td>3.9.17</td>
<td>Texture Comparison Modes</td>
<td>265</td>
</tr>
<tr>
<td>3.9.18</td>
<td>sRGB Texture Color Conversion</td>
<td>266</td>
</tr>
<tr>
<td>3.9.19</td>
<td>Shared Exponent Texture Color Conversion</td>
<td>267</td>
</tr>
<tr>
<td>3.9.20</td>
<td>Texture Image Loads and Stores</td>
<td>267</td>
</tr>
<tr>
<td>3.10</td>
<td>Fragment Shaders</td>
<td>275</td>
</tr>
<tr>
<td>3.10.1</td>
<td>Shader Variables</td>
<td>275</td>
</tr>
<tr>
<td>3.10.2</td>
<td>Shader Execution</td>
<td>277</td>
</tr>
<tr>
<td>3.11</td>
<td>Antialiasing Application</td>
<td>284</td>
</tr>
<tr>
<td>3.12</td>
<td>Multisample Point Fade</td>
<td>285</td>
</tr>
<tr>
<td>4</td>
<td>Per-Fragment Operations and the Framebuffer</td>
<td>286</td>
</tr>
<tr>
<td>4.1</td>
<td>Per-Fragment Operations</td>
<td>287</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Pixel Ownership Test</td>
<td>288</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Scissor Test</td>
<td>288</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Multisample Fragment Operations</td>
<td>290</td>
</tr>
<tr>
<td>4.1.4</td>
<td>Stencil Test</td>
<td>292</td>
</tr>
<tr>
<td>4.1.5</td>
<td>Depth Buffer Test</td>
<td>294</td>
</tr>
<tr>
<td>4.1.6</td>
<td>Occlusion Queries</td>
<td>294</td>
</tr>
<tr>
<td>4.1.7</td>
<td>Blending</td>
<td>295</td>
</tr>
<tr>
<td>4.1.8</td>
<td>sRGB Conversion</td>
<td>302</td>
</tr>
<tr>
<td>4.1.9</td>
<td>Dithering</td>
<td>302</td>
</tr>
<tr>
<td>4.1.10</td>
<td>Logical Operation</td>
<td>303</td>
</tr>
<tr>
<td>4.1.11</td>
<td>Additional Multisample Fragment Operations</td>
<td>304</td>
</tr>
<tr>
<td>4.2</td>
<td>Whole Framebuffer Operations</td>
<td>305</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Selecting Buffers for Writing</td>
<td>305</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Fine Control of Buffer Updates</td>
<td>309</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Clearing the Buffers</td>
<td>311</td>
</tr>
<tr>
<td>4.3</td>
<td>Reading and Copying Pixels</td>
<td>313</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Reading Pixels</td>
<td>314</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Copying Pixels</td>
<td>319</td>
</tr>
</tbody>
</table>
## CONTENTS

<table>
<thead>
<tr>
<th>Version</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F.1</td>
<td>479</td>
</tr>
<tr>
<td></td>
<td>F.2</td>
<td>480</td>
</tr>
<tr>
<td></td>
<td>F.3</td>
<td>481</td>
</tr>
<tr>
<td></td>
<td>F.4</td>
<td>481</td>
</tr>
<tr>
<td></td>
<td>F.5</td>
<td>483</td>
</tr>
<tr>
<td>G</td>
<td>G.1</td>
<td>486</td>
</tr>
<tr>
<td></td>
<td>G.2</td>
<td>487</td>
</tr>
<tr>
<td></td>
<td>G.3</td>
<td>487</td>
</tr>
<tr>
<td></td>
<td>G.4</td>
<td>488</td>
</tr>
<tr>
<td>H</td>
<td>H.1</td>
<td>491</td>
</tr>
<tr>
<td></td>
<td>H.2</td>
<td>492</td>
</tr>
<tr>
<td></td>
<td>H.3</td>
<td>492</td>
</tr>
<tr>
<td></td>
<td>H.4</td>
<td>493</td>
</tr>
<tr>
<td></td>
<td>H.5</td>
<td>495</td>
</tr>
<tr>
<td>I</td>
<td>I.1</td>
<td>497</td>
</tr>
<tr>
<td></td>
<td>I.2</td>
<td>498</td>
</tr>
<tr>
<td></td>
<td>I.3</td>
<td>499</td>
</tr>
<tr>
<td></td>
<td>I.4</td>
<td>499</td>
</tr>
<tr>
<td>J</td>
<td>J.1</td>
<td>501</td>
</tr>
<tr>
<td></td>
<td>J.2</td>
<td>503</td>
</tr>
<tr>
<td></td>
<td>J.3</td>
<td>503</td>
</tr>
<tr>
<td></td>
<td>J.4</td>
<td>503</td>
</tr>
<tr>
<td>K</td>
<td>K.1</td>
<td>506</td>
</tr>
<tr>
<td></td>
<td>K.2</td>
<td>507</td>
</tr>
<tr>
<td></td>
<td>K.3</td>
<td>507</td>
</tr>
<tr>
<td></td>
<td>K.4</td>
<td>507</td>
</tr>
<tr>
<td></td>
<td>K.5</td>
<td>507</td>
</tr>
</tbody>
</table>

OpenGL 4.2 (Core Profile) - April 27, 2012
CONTENTS

L  Version 4.2 ........................................... 510
   L.1 New Features ................................. 510
   L.2 Deprecation Model ......................... 511
   L.3 Changed Tokens ............................ 511
   L.4 Change Log ................................ 512
   L.5 Credits and Acknowledgements .......... 520

M  Extension Registry, Header Files, and ARB Extensions .......... 522
   M.1 Extension Registry .......................... 522
   M.2 Header Files ................................ 522
   M.3 ARB Extensions ............................. 523
      M.3.1 Naming Conventions .................... 523
      M.3.2 Promoting Extensions to Core Features 524
      M.3.3 Multitexture ........................... 524
      M.3.4 Transpose Matrix ....................... 524
      M.3.5 Multisample ............................ 524
      M.3.6 Texture Add Environment Mode .......... 525
      M.3.7 Cube Map Textures ....................... 525
      M.3.8 Compressed Textures .................... 525
      M.3.9 Texture Border Clamp .................... 525
      M.3.10 Point Parameters ....................... 525
      M.3.11 Vertex Blend ............................ 525
      M.3.12 Matrix Palette .......................... 525
      M.3.13 Texture Combine Environment Mode .... 526
      M.3.14 Texture Crossbar Environment Mode .... 526
      M.3.15 Texture Dot3 Environment Mode ......... 526
      M.3.16 Texture Mirrored Repeat ............... 526
      M.3.17 Depth Texture .......................... 526
      M.3.18 Shadow .................................. 526
      M.3.19 Shadow Ambient ......................... 526
      M.3.20 Window Raster Position ................. 526
      M.3.21 Low-Level Vertex Programming .......... 527
      M.3.22 Low-Level Fragment Programming ....... 527
      M.3.23 Buffer Objects .......................... 527
      M.3.24 Occlusion Queries ....................... 527
      M.3.25 Shader Objects .......................... 527
      M.3.26 High-Level Vertex Programming .......... 527
      M.3.27 High-Level Fragment Programming ....... 527
      M.3.28 OpenGL Shading Language ............... 528
      M.3.29 Non-Power-Of-Two Textures .............. 528
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>M.3.30</td>
<td>Point Sprites</td>
<td>528</td>
</tr>
<tr>
<td>M.3.31</td>
<td>Fragment Program Shadow</td>
<td>528</td>
</tr>
<tr>
<td>M.3.32</td>
<td>Multiple Render Targets</td>
<td>528</td>
</tr>
<tr>
<td>M.3.33</td>
<td>Rectangular Textures</td>
<td>528</td>
</tr>
<tr>
<td>M.3.34</td>
<td>Floating-Point Color Buffers</td>
<td>529</td>
</tr>
<tr>
<td>M.3.35</td>
<td>Half-Precision Floating Point</td>
<td>529</td>
</tr>
<tr>
<td>M.3.36</td>
<td>Floating-Point Textures</td>
<td>529</td>
</tr>
<tr>
<td>M.3.37</td>
<td>Pixel Buffer Objects</td>
<td>529</td>
</tr>
<tr>
<td>M.3.38</td>
<td>Floating-Point Depth Buffers</td>
<td>530</td>
</tr>
<tr>
<td>M.3.39</td>
<td>Instanced Rendering</td>
<td>530</td>
</tr>
<tr>
<td>M.3.40</td>
<td>Framebuffer Objects</td>
<td>530</td>
</tr>
<tr>
<td>M.3.41</td>
<td>sRGB Framebuffers</td>
<td>530</td>
</tr>
<tr>
<td>M.3.42</td>
<td>Geometry Shaders</td>
<td>530</td>
</tr>
<tr>
<td>M.3.43</td>
<td>Half-Precision Vertex Data</td>
<td>531</td>
</tr>
<tr>
<td>M.3.44</td>
<td>Instanced Rendering</td>
<td>531</td>
</tr>
<tr>
<td>M.3.45</td>
<td>Flexible Buffer Mapping</td>
<td>531</td>
</tr>
<tr>
<td>M.3.46</td>
<td>Texture Buffer Objects</td>
<td>531</td>
</tr>
<tr>
<td>M.3.47</td>
<td>RGTC Texture Compression Formats</td>
<td>531</td>
</tr>
<tr>
<td>M.3.48</td>
<td>One- and Two-Component Texture Formats</td>
<td>531</td>
</tr>
<tr>
<td>M.3.49</td>
<td>Vertex Array Objects</td>
<td>532</td>
</tr>
<tr>
<td>M.3.50</td>
<td>Versioned Context Creation</td>
<td>532</td>
</tr>
<tr>
<td>M.3.51</td>
<td>Uniform Buffer Objects</td>
<td>532</td>
</tr>
<tr>
<td>M.3.52</td>
<td>Restoration of features removed from OpenGL 3.0</td>
<td>532</td>
</tr>
<tr>
<td>M.3.53</td>
<td>Fast Buffer-to-Buffer Copies</td>
<td>533</td>
</tr>
<tr>
<td>M.3.54</td>
<td>Shader Texture Level of Detail Control</td>
<td>533</td>
</tr>
<tr>
<td>M.3.55</td>
<td>Depth Clamp Control</td>
<td>533</td>
</tr>
<tr>
<td>M.3.56</td>
<td>Base Vertex Offset Drawing Commands</td>
<td>533</td>
</tr>
<tr>
<td>M.3.57</td>
<td>Fragment Coordinate Convention Control</td>
<td>533</td>
</tr>
<tr>
<td>M.3.58</td>
<td>Provoking Vertex Control</td>
<td>533</td>
</tr>
<tr>
<td>M.3.59</td>
<td>Seamless Cube Maps</td>
<td>534</td>
</tr>
<tr>
<td>M.3.60</td>
<td>Fence Sync Objects</td>
<td>534</td>
</tr>
<tr>
<td>M.3.61</td>
<td>Multisample Textures</td>
<td>534</td>
</tr>
<tr>
<td>M.3.62</td>
<td>BGRA Attribute Component Ordering</td>
<td>534</td>
</tr>
<tr>
<td>M.3.63</td>
<td>Per-Buffer Blend Control</td>
<td>534</td>
</tr>
<tr>
<td>M.3.64</td>
<td>Sample Shading Control</td>
<td>534</td>
</tr>
<tr>
<td>M.3.65</td>
<td>Cube Map Array Textures</td>
<td>535</td>
</tr>
<tr>
<td>M.3.66</td>
<td>Texture Gather</td>
<td>535</td>
</tr>
<tr>
<td>M.3.67</td>
<td>Texture Level-Of-Detail Queries</td>
<td>535</td>
</tr>
<tr>
<td>M.3.68</td>
<td>Profiled Context Creation</td>
<td>535</td>
</tr>
<tr>
<td>M.3.69</td>
<td>Shading Language Include</td>
<td>535</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Block diagram of the GL.</td>
<td>17</td>
</tr>
<tr>
<td>2.2</td>
<td>Vertex processing and primitive assembly.</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>Triangle strips, fans, and independent triangles.</td>
<td>23</td>
</tr>
<tr>
<td>2.4</td>
<td>Lines with adjacency.</td>
<td>24</td>
</tr>
<tr>
<td>2.5</td>
<td>Triangles with adjacency.</td>
<td>25</td>
</tr>
<tr>
<td>2.6</td>
<td>Triangle strips with adjacency.</td>
<td>26</td>
</tr>
<tr>
<td>2.7</td>
<td>Domain parameterization for tessellation.</td>
<td>131</td>
</tr>
<tr>
<td>2.8</td>
<td>Inner triangle tessellation.</td>
<td>135</td>
</tr>
<tr>
<td>2.9</td>
<td>Inner quad tessellation.</td>
<td>137</td>
</tr>
<tr>
<td>2.10</td>
<td>Isoline tessellation.</td>
<td>139</td>
</tr>
<tr>
<td>3.1</td>
<td>Rasterization.</td>
<td>174</td>
</tr>
<tr>
<td>3.2</td>
<td>Visualization of Bresenham’s algorithm.</td>
<td>183</td>
</tr>
<tr>
<td>3.3</td>
<td>Rasterization of non-antialiased wide lines.</td>
<td>185</td>
</tr>
<tr>
<td>3.4</td>
<td>The region used in rasterizing an antialiased line segment.</td>
<td>186</td>
</tr>
<tr>
<td>3.5</td>
<td>Transfer of pixel rectangles.</td>
<td>195</td>
</tr>
<tr>
<td>3.6</td>
<td>Selecting a subimage from an image.</td>
<td>199</td>
</tr>
<tr>
<td>3.7</td>
<td>A texture image and the coordinates used to access it.</td>
<td>226</td>
</tr>
<tr>
<td>3.8</td>
<td>Example of the components returned for <code>textureGather</code>.</td>
<td>252</td>
</tr>
<tr>
<td>4.1</td>
<td>Per-fragment operations.</td>
<td>288</td>
</tr>
<tr>
<td>4.2</td>
<td>Operation of <strong>ReadPixels</strong>.</td>
<td>314</td>
</tr>
</tbody>
</table>
List of Tables

2.1 GL command suffixes .................................................. 14
2.2 GL data types ............................................................. 16
2.3 Summary of GL errors .................................................... 20
2.4 Triangles generated by triangle strips with adjacency. ............... 27
2.5 Vertex array sizes (values per vertex) and data types ................. 32
2.6 Packed component layout for non-BGRA formats. ..................... 35
2.7 Packed component layout for BGRA format. ........................ 35
2.8 Buffer object binding targets. ....................................... 44
2.9 Buffer object parameters and their values. ........................ 45
2.10 Buffer object initial state. .......................................... 48
2.11 Buffer object state set by MapBufferRange. ....................... 50
2.12 Scalar and vector vertex attribute types .......................... 74
2.13 OpenGL Shading Language type tokens ............................ 89
2.14 Transform feedback modes ......................................... 164
2.15 Provoking vertex selection. ......................................... 170

3.1 PixelStore parameters. ................................................. 194
3.2 Pixel data types. .......................................................... 197
3.3 Pixel data formats. ...................................................... 198
3.4 Swap Bytes bit ordering. .............................................. 198
3.5 Packed pixel formats. ................................................... 201
3.6 UNSIGNED_BYTE formats. Bit numbers are indicated for each component ........................................... 202
3.7 UNSIGNED_SHORT formats .............................................. 203
3.8 UNSIGNED_INT formats .................................................. 204
3.9 FLOAT_UNSIGNED_INT formats ......................................... 205
3.10 Packed pixel field assignments. ..................................... 206
3.11 Conversion from RGBA, depth, and stencil pixel components to internal texture components. .......................... 215
LIST OF TABLES

3.12 Sized internal color formats. ................................. 220
3.13 Sized internal depth and stencil formats. .................. 221
3.14 Generic and specific compressed internal formats. ....... 222
3.15 Internal formats for buffer textures ......................... 242
3.16 Texture parameters and their values. ....................... 244
3.17 Selection of cube map images. ............................... 246
3.18 Texel location wrap mode application. ..................... 251
3.19 Depth texture comparison functions. ......................... 266
3.20 Mapping of image load, store, and atomic texel coordinate components to texel numbers. 269
3.21 Supported image unit formats, with equivalent format layout qualifiers. 272
3.22 Texel sizes, compatibility classes, and pixel format/type combinations for each image format. 274
3.23 Correspondence of filtered texture components to texture base components. 278

4.1 RGB and Alpha blend equations. .............................. 298
4.2 Blending functions. ............................................. 300
4.3 Arguments to LogicOp and their corresponding operations. 304
4.4 Buffer selection for the default framebuffer ................ 307
4.5 Buffer selection for a framebuffer object ................... 307
4.6 DrawBuffers buffer selection for the default framebuffer .... 307
4.7 PixelStore parameters. ......................................... 315
4.8 ReadPixels GL data types and reversed component conversion formulas. 320
4.9 ReadPixels index masks. ....................................... 321
4.10 Correspondence of renderbuffer sized to base internal formats. 330
4.11 Framebuffer attachment points. .............................. 332
4.12 Layer numbers for cube map texture faces. ................ 346

5.1 Initial properties of a sync object created with FenceSync. 350
5.2 Hint targets and descriptions .................................. 353

6.1 Texture, table, and filter return values. ...................... 360
6.2 Context profile bits ............................................ 363
6.3 State Variable Types ........................................... 382
6.4 Current Values and Associated Data ......................... 383
6.5 Vertex Array Object State (cont.) ........................... 384
6.6 Vertex Array Object State (cont.) ........................... 385
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.7</td>
<td>Vertex Array Data (not in Vertex Array objects)</td>
<td>386</td>
</tr>
<tr>
<td>6.8</td>
<td>Buffer Object State</td>
<td>387</td>
</tr>
<tr>
<td>6.9</td>
<td>Transformation state</td>
<td>388</td>
</tr>
<tr>
<td>6.10</td>
<td>Coloring</td>
<td>389</td>
</tr>
<tr>
<td>6.11</td>
<td>Rasterization</td>
<td>390</td>
</tr>
<tr>
<td>6.12</td>
<td>Rasterization (cont.)</td>
<td>391</td>
</tr>
<tr>
<td>6.13</td>
<td>Multisampling</td>
<td>392</td>
</tr>
<tr>
<td>6.14</td>
<td>Textures (state per texture unit and binding point)</td>
<td>393</td>
</tr>
<tr>
<td>6.15</td>
<td>Textures (state per texture unit and binding point)(cont.)</td>
<td>394</td>
</tr>
<tr>
<td>6.16</td>
<td>Textures (state per texture object)</td>
<td>395</td>
</tr>
<tr>
<td>6.17</td>
<td>Textures (state per texture image)</td>
<td>396</td>
</tr>
<tr>
<td>6.18</td>
<td>Textures (state per sampler object)</td>
<td>397</td>
</tr>
<tr>
<td>6.19</td>
<td>Texture Environment and Generation</td>
<td>398</td>
</tr>
<tr>
<td>6.20</td>
<td>Pixel Operations</td>
<td>399</td>
</tr>
<tr>
<td>6.21</td>
<td>Pixel Operations (cont.)</td>
<td>400</td>
</tr>
<tr>
<td>6.22</td>
<td>Framebuffer Control</td>
<td>401</td>
</tr>
<tr>
<td>6.23</td>
<td>Framebuffer (state per target binding point)</td>
<td>402</td>
</tr>
<tr>
<td>6.24</td>
<td>Framebuffer (state per framebuffer object)†</td>
<td>403</td>
</tr>
<tr>
<td>6.25</td>
<td>Framebuffer (state per attachment point)</td>
<td>404</td>
</tr>
<tr>
<td>6.26</td>
<td>Renderbuffer (state per target and binding point)</td>
<td>405</td>
</tr>
<tr>
<td>6.27</td>
<td>Renderbuffer (state per renderbuffer object)</td>
<td>406</td>
</tr>
<tr>
<td>6.28</td>
<td>Pixels</td>
<td>407</td>
</tr>
<tr>
<td>6.29</td>
<td>Pixels (cont.)</td>
<td>408</td>
</tr>
<tr>
<td>6.30</td>
<td>Shader Object State</td>
<td>409</td>
</tr>
<tr>
<td>6.31</td>
<td>Program Pipeline Object State</td>
<td>410</td>
</tr>
<tr>
<td>6.32</td>
<td>Program Object State</td>
<td>411</td>
</tr>
<tr>
<td>6.33</td>
<td>Program Object State (cont.)</td>
<td>412</td>
</tr>
<tr>
<td>6.34</td>
<td>Program Object State (cont.)</td>
<td>413</td>
</tr>
<tr>
<td>6.35</td>
<td>Program Object State (cont.)</td>
<td>414</td>
</tr>
<tr>
<td>6.36</td>
<td>Program Object State (cont.)</td>
<td>415</td>
</tr>
<tr>
<td>6.37</td>
<td>Program Object State (cont.)</td>
<td>416</td>
</tr>
<tr>
<td>6.38</td>
<td>Program Object State (cont.)</td>
<td>417</td>
</tr>
<tr>
<td>6.39</td>
<td>Program Object State (cont.)</td>
<td>418</td>
</tr>
<tr>
<td>6.40</td>
<td>Vertex and Geometry Shader State</td>
<td>419</td>
</tr>
<tr>
<td>6.41</td>
<td>Query Object State</td>
<td>420</td>
</tr>
<tr>
<td>6.42</td>
<td>Image State (state per image unit)</td>
<td>421</td>
</tr>
<tr>
<td>6.43</td>
<td>Transform Feedback State</td>
<td>422</td>
</tr>
<tr>
<td>6.44</td>
<td>Atomic Counter State</td>
<td>423</td>
</tr>
<tr>
<td>6.45</td>
<td>Sync (state per sync object)</td>
<td>424</td>
</tr>
</tbody>
</table>
LIST OF TABLES

6.46 Hints ................................................. 425
6.47 Implementation Dependent Values .......................... 426
6.48 Implementation Dependent Values (cont.) ................. 427
6.49 Implementation Dependent Values (cont.) .................. 428
6.50 Implementation Dependent Version and Extension Support 429
6.51 Implementation Dependent Vertex Shader Limits .......... 430
6.52 Implementation Dependent Tessellation Shader Limits .. 431
6.53 Implementation Dependent Geometry Shader Limits ..... 432
6.54 Implementation Dependent Fragment Shader Limits ....... 433
6.55 Implementation Dependent Aggregate Shader Limits ... 434
6.56 Implementation Dependent Aggregate Shader Limits (cont.) 435
6.57 Implementation Dependent Aggregate Shader Limits (cont.) 436
† The minimum value for each stage is
\[
\text{MAX}_{\text{stage}}_{\text{UNIFORM_BLOCKS}} \times \text{MAX}_{\text{UNIFORM_BLOCK_SIZE}} / \left(4 + \text{MAX}_{\text{stage}}_{\text{UNIFORM_COMPONENTS}}\right)
\] 436
6.58 Implementation Dependent Values (cont.) ................. 437
6.59 Implementation Dependent Values (cont.) ................. 438
6.60 Internal Format Dependent Values ........................ 439
6.61 Implementation Dependent Transform Feedback Limits .... 440
6.62 Framebuffer Dependent Values ............................. 441
6.63 Miscellaneous .......................................... 442

C.1 Mode-dependent BPTC parameters .......................... 458
C.2 Partition table for 2 subset ................................ 459
C.3 Partition table for 3 subset ................................ 460
C.4 Anchor index values for the second subset of two-subset partitioning 461
C.5 Anchor index values for the second subset of three-subset partitioning 461
C.6 Anchor index values for the third subset of three-subset partitioning 461
C.7 Endpoint and partition parameters for block modes .......... 464
C.8 Block formats for block modes ................................ 465

F.1 New token names ......................................... 481
H.1 New token names ......................................... 493
K.1 New token names ......................................... 507
L.1 New token names ......................................... 512

OpenGL 4.2 (Core Profile) - April 27, 2012
Chapter 1

Introduction

This document describes the OpenGL graphics system: what it is, how it acts, and what is required to implement it. We assume that the reader has at least a rudimentary understanding of computer graphics. This means familiarity with the essentials of computer graphics algorithms as well as familiarity with basic graphics hardware and associated terms.

1.1 Formatting of the OpenGL Specification

1.1.1 Formatting of Changes

This version of the OpenGL 4.2 Specification marks changes relative to the first public release of OpenGL 4.2 Specification by typesetting them in purple, like this paragraph. Details of these changes are also described in appendix L.4.

1.2 What is the OpenGL Graphics System?

OpenGL (for “Open Graphics Library”) is a software interface to graphics hardware. The interface consists of a set of several hundred procedures and functions that allow a programmer to specify the objects and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects.

Most of OpenGL requires that the graphics hardware contain a framebuffer. Many OpenGL calls pertain to drawing objects such as points, lines, and polygons, but the way that some of this drawing occurs (such as when antialiasing is enabled) relies on the existence of a framebuffer. Further, some of OpenGL is specifically concerned with framebuffer manipulation.
1.3 Programmer’s View of OpenGL

To the programmer, OpenGL is a set of commands that allow the specification of geometric objects in two or three dimensions, together with commands that control how these objects are rendered into the framebuffer.

A typical program that uses OpenGL begins with calls to open a window into the framebuffer into which the program will draw. Then, calls are made to allocate an OpenGL context and associate it with the window. Once a context is allocated, the programmer is free to issue OpenGL commands. Some calls are used to draw simple geometric objects (i.e. points, line segments, and polygons), while others affect the rendering of these primitives including how they are lit or colored and how they are mapped from the user’s two- or three-dimensional model space to the two-dimensional screen. There are also calls to effect direct control of the framebuffer, such as reading and writing pixels.

1.4 Implementor’s View of OpenGL

To the implementor, OpenGL is a set of commands that affect the operation of graphics hardware. If the hardware consists only of an addressable framebuffer, then OpenGL must be implemented almost entirely on the host CPU. More typically, the graphics hardware may comprise varying degrees of graphics acceleration, from a raster subsystem capable of rendering two-dimensional lines and polygons to sophisticated floating-point processors capable of transforming and computing on geometric data. The OpenGL implementor’s task is to provide the CPU software interface while dividing the work for each OpenGL command between the CPU and the graphics hardware. This division must be tailored to the available graphics hardware to obtain optimum performance in carrying out OpenGL calls.

OpenGL maintains a considerable amount of state information. This state controls how objects are drawn into the framebuffer. Some of this state is directly available to the user: he or she can make calls to obtain its value. Some of it, however, is visible only by the effect it has on what is drawn. One of the main goals of this Specification is to make OpenGL state information explicit, to elucidate how it changes, and to indicate what its effects are.

1.5 Our View

We view OpenGL as a pipeline having some programmable stages and some state-driven stages that control a set of specific drawing operations. This model should
engender a specification that satisfies the needs of both programmers and implementors. It does not, however, necessarily provide a model for implementation. An implementation must produce results conforming to those produced by the specified methods, but there may be ways to carry out a particular computation that are more efficient than the one specified.

1.6 The Deprecation Model

OpenGL Features marked as deprecated in one version of the Specification are expected to be removed in a future version, allowing applications time to transition away from use of deprecated features. The deprecation model is described in more detail, together with a summary of the commands and state deprecated from this version of the API, in appendix E.

1.7 Companion Documents

1.7.1 OpenGL Shading Language

This Specification should be read together with a companion document titled The OpenGL Shading Language. The latter document (referred to as the OpenGL Shading Language Specification hereafter) defines the syntax and semantics of the programming language used to write vertex and fragment shaders (see sections 2.11 and 3.10). These sections may include references to concepts and terms (such as shading language variable types) defined in the companion document.

OpenGL 4.2 implementations are guaranteed to support version 4.20 of the OpenGL Shading Language. All references to sections of that specification refer to version 4.20. The latest supported version of the shading language may be queried as described in section 6.1.5.

The core profile of OpenGL 4.2 is also guaranteed to support all previous versions of the OpenGL Shading Language back to version 1.40. In some implementations the core profile may also support earlier versions of the Shading Language, and may support compatibility profile versions of the Shading Language, although errors will be generated when using language features not supported by the core profile API.

1.7.2 Window System Bindings

OpenGL requires a companion API to create and manage graphics contexts, windows to render into, and other resources beyond the scope of this Specification. There are several such APIs supporting different operating and window systems.
1.7. COMPANION DOCUMENTS

*OpenGL Graphics with the X Window System*, also called the “GLX Specification”, describes the GLX API for use of OpenGL in the X Window System. It is primarily directed at Linux and Unix systems, but GLX implementations also exist for Microsoft Windows, MacOS X, and some other platforms where X is available. The GLX Specification is available in the OpenGL Extension Registry (see appendix M).

The WGL API supports use of OpenGL with Microsoft Windows. WGL is documented in Microsoft’s MSDN system, although no full specification exists.

Several APIs exist supporting use of OpenGL with Quartz, the MacOS X window system, including CGL, AGL, and NSOpenGLView. These APIs are documented on Apple’s developer website.

The *Khronos Native Platform Graphics Interface* or “EGL Specification” describes the EGL API for use of OpenGL ES on mobile and embedded devices. EGL implementations may be available supporting OpenGL as well. The EGL Specification is available in the Khronos Extension Registry at URL

http://www.khronos.org/registry/egl
Chapter 2

OpenGL Operation

2.1 OpenGL Fundamentals

OpenGL (henceforth, the “GL”) is concerned only with rendering into a framebuffer (and reading values stored in that framebuffer). There is no support for other peripherals sometimes associated with graphics hardware, such as mice and keyboards. Programmers must rely on other mechanisms to obtain user input.

The GL draws primitives subject to a number of selectable modes and shader programs. Each primitive is a point, line segment, or polygon. Each mode may be changed independently; the setting of one does not affect the settings of others (although many modes may interact to determine what eventually ends up in the framebuffer). Modes are set, primitives specified, and other GL operations described by sending commands in the form of function or procedure calls.

Primitives are defined by a group of one or more vertices. A vertex defines a point, an endpoint of an edge, or a corner of a polygon where two edges meet. Data such as positional coordinates, colors, normals, texture coordinates, etc. are associated with a vertex and each vertex is processed independently, in order, and in the same way. The only exception to this rule is if the group of vertices must be clipped so that the indicated primitive fits within a specified region; in this case vertex data may be modified and new vertices created. The type of clipping depends on which primitive the group of vertices represents.

Commands are always processed in the order in which they are received, although there may be an indeterminate delay before the effects of a command are realized. This means, for example, that one primitive must be drawn completely before any subsequent one can affect the framebuffer. It also means that queries and pixel read operations return state consistent with complete execution of all previously invoked GL commands, except where explicitly specified otherwise. In
general, the effects of a GL command on either GL modes or the framebuffer must be complete before any subsequent command can have any such effects.

In the GL, data binding occurs on call. This means that data passed to a command are interpreted when that command is received. Even if the command requires a pointer to data, those data are interpreted when the call is made, and any subsequent changes to the data have no effect on the GL (unless the same pointer is used in a subsequent command).

The GL provides direct control over the fundamental operations of 3D and 2D graphics. This includes specification of parameters of application-defined shader programs performing transformation, lighting, texturing, and shading operations, as well as built-in functionality such as antialiasing and texture filtering. It does not provide a means for describing or modeling complex geometric objects. Another way to describe this situation is to say that the GL provides mechanisms to describe how complex geometric objects are to be rendered rather than mechanisms to describe the complex objects themselves.

The model for interpretation of GL commands is client-server. That is, a program (the client) issues commands, and these commands are interpreted and processed by the GL (the server). The server may or may not operate on the same computer as the client. In this sense, the GL is “network-transparent.” A server may maintain a number of GL contexts, each of which is an encapsulation of current GL state. A client may choose to connect to any one of these contexts. Issuing GL commands when the program is not connected to a context results in undefined behavior.

The GL interacts with two classes of framebuffers: window system-provided and application-created. There is at most one window system-provided framebuffer at any time, referred to as the default framebuffer. Application-created framebuffers, referred to as framebuffer objects, may be created as desired. These two types of framebuffer are distinguished primarily by the interface for configuring and managing their state.

The effects of GL commands on the default framebuffer are ultimately controlled by the window system, which allocates framebuffer resources, determines which portions of the default framebuffer the GL may access at any given time, and communicates to the GL how those portions are structured. Therefore, there are no GL commands to initialize a GL context or configure the default framebuffer. Similarly, display of framebuffer contents on a physical display device (including the transformation of individual framebuffer values by such techniques as gamma correction) is not addressed by the GL.

Allocation and configuration of the default framebuffer occurs outside of the GL in conjunction with the window system, using companion APIs described in section 1.7.2.
Allocation and initialization of GL contexts is also done using these companion APIs. GL contexts can typically be associated with different default framebuffers, and some context state is determined at the time this association is performed.

It is possible to use a GL context without a default framebuffer, in which case a framebuffer object must be used to perform all rendering. This is useful for applications needing to perform offscreen rendering.

The GL is designed to be run on a range of graphics platforms with varying graphics capabilities and performance. To accommodate this variety, we specify ideal behavior instead of actual behavior for certain GL operations. In cases where deviation from the ideal is allowed, we also specify the rules that an implementation must obey if it is to approximate the ideal behavior usefully. This allowed variation in GL behavior implies that two distinct GL implementations may not agree pixel for pixel when presented with the same input even when run on identical framebuffer configurations.

Finally, command names, constants, and types are prefixed in the GL (by gl, GL_, and GL, respectively in C) to reduce name clashes with other packages. The prefixes are omitted in this document for clarity.

### 2.1.1 Numeric Computation

The GL must perform a number of floating-point operations during the course of its operation.

Implementations will normally perform computations in floating-point, and must meet the range and precision requirements defined under "Floating-Point Computation" below.

These requirements only apply to computations performed in GL operations outside of shader execution, such as texture image specification and per-fragment operations. Range and precision requirements during shader execution differ and are as specified by the OpenGL Shading Language Specification.

In some cases, the representation and/or precision of operations is implicitly limited by the specified format of vertex, texture, or renderbuffer data consumed by the GL. Specific floating-point formats are described later in this section.

**Floating-Point Computation**

We do not specify how floating-point numbers are to be represented, or the details of how operations on them are performed.

We require simply that numbers’ floating-point parts contain enough bits and that their exponent fields are large enough so that individual results of floating-point operations are accurate to about 1 part in \(10^5\). The maximum representable
magnitude for all floating-point values must be at least $2^{32}$. $x \cdot 0 = 0 \cdot x = 0$ for any non-infinite and non-NaN $x$. $1 \cdot x = x \cdot 1 = x$. $x + 0 = 0 + x = x$. $0^0 = 1$. (Occasionally further requirements will be specified.) Most single-precision floating-point formats meet these requirements.

The special values $\text{Inf}$ and $-\text{Inf}$ encode values with magnitudes too large to be represented; the special value $\text{NaN}$ encodes “Not A Number” values resulting from undefined arithmetic operations such as $\frac{0}{0}$. Implementations are permitted, but not required, to support $\text{Inf}$s and $\text{NaN}$s in their floating-point computations.

Any representable floating-point value is legal as input to a GL command that requires floating-point data. The result of providing a value that is not a floating-point number to such a command is unspecified, but must not lead to GL interruption or termination. In IEEE arithmetic, for example, providing a negative zero or a denormalized number to a GL command yields predictable results, while providing a NaN or an infinity yields unspecified results.

### 16-Bit Floating-Point Numbers

A 16-bit floating-point number has a 1-bit sign ($S$), a 5-bit exponent ($E$), and a 10-bit mantissa ($M$). The value $V$ of a 16-bit floating-point number is determined by the following:

$$V = \begin{cases} 
(-1)^S \times 0.0, & E = 0, M = 0 \\
(-1)^S \times 2^{-14} \times \frac{M}{2^{10}}, & E = 0, M \neq 0 \\
(-1)^S \times 2^{E-15} \times (1 + \frac{M}{2^{10}}), & 0 < E < 31 \\
(-1)^S \times \text{Inf}, & E = 31, M = 0 \\
\text{NaN}, & E = 31, M \neq 0
\end{cases}$$

If the floating-point number is interpreted as an unsigned 16-bit integer $N$, then

$$S = \left\lfloor \frac{N \text{ mod } 65536}{32768} \right\rfloor$$

$$E = \left\lfloor \frac{N \text{ mod } 32768}{1024} \right\rfloor$$

$$M = N \text{ mod } 1024.$$

Any representable 16-bit floating-point value is legal as input to a GL command that accepts 16-bit floating-point data. The result of providing a value that is not a floating-point number (such as $\text{Inf}$ or $\text{NaN}$) to such a command is unspecified, but must not lead to GL interruption or termination. Providing a denormalized number or negative zero to GL must yield predictable results.
2.1. OPENGL FUNDAMENTALS

Unsigned 11-Bit Floating-Point Numbers

An unsigned 11-bit floating-point number has no sign bit, a 5-bit exponent \(E\), and a 6-bit mantissa \(M\). The value \(V\) of an unsigned 11-bit floating-point number is determined by the following:

\[
V = \begin{cases} 
0.0, & E = 0, M = 0 \\
2^{-14} \times \frac{M}{64}, & E = 0, M \neq 0 \\
2^{E-15} \times (1 + \frac{M}{64}), & 0 < E < 31 \\
\text{Inf}, & E = 31, M = 0 \\
\text{NaN}, & E = 31, M \neq 0
\end{cases}
\]

If the floating-point number is interpreted as an unsigned 11-bit integer \(N\), then

\[
E = \left\lfloor \frac{N}{64} \right\rfloor \\
M = N \mod 64.
\]

When a floating-point value is converted to an unsigned 11-bit floating-point representation, finite values are rounded to the closest representable finite value. While less accurate, implementations are allowed to always round in the direction of zero. This means negative values are converted to zero. Likewise, finite positive values greater than 65024 (the maximum finite representable unsigned 11-bit floating-point value) are converted to 65024. Additionally: negative infinity is converted to zero; positive infinity is converted to positive infinity; and both positive and negative \(NaN\) are converted to positive \(NaN\).

Any representable unsigned 11-bit floating-point value is legal as input to a GL command that accepts 11-bit floating-point data. The result of providing a value that is not a floating-point number (such as \(Inf\) or \(NaN\)) to such a command is unspecified, but must not lead to GL interruption or termination. Providing a denormalized number to GL must yield predictable results.

Unsigned 10-Bit Floating-Point Numbers

An unsigned 10-bit floating-point number has no sign bit, a 5-bit exponent \(E\), and a 5-bit mantissa \(M\). The value \(V\) of an unsigned 10-bit floating-point number is determined by the following:
2.1. OPENGL FUNDAMENTALS

\[
V = \begin{cases} 
0.0, & E = 0, M = 0 \\
2^{-14} \times \frac{M}{32}, & E = 0, M \neq 0 \\
2^{E-15} \times (1 + \frac{M}{32}), & 0 < E < 31 \\
\text{Inf}, & E = 31, M = 0 \\
\text{NaN}, & E = 31, M \neq 0 
\end{cases}
\]

If the floating-point number is interpreted as an unsigned 10-bit integer \( N \), then

\[
E = \left\lfloor \frac{N}{32} \right\rfloor \\
M = N \mod 32.
\]

When a floating-point value is converted to an unsigned 10-bit floating-point representation, finite values are rounded to the closest representable finite value. While less accurate, implementations are allowed to always round in the direction of zero. This means negative values are converted to zero. Likewise, finite positive values greater than 64512 (the maximum finite representable unsigned 10-bit floating-point value) are converted to 64512. Additionally: negative infinity is converted to zero; positive infinity is converted to positive infinity; and both positive and negative NaN are converted to positive NaN.

Any representable unsigned 10-bit floating-point value is legal as input to a GL command that accepts 10-bit floating-point data. The result of providing a value that is not a floating-point number (such as Inf or NaN) to such a command is unspecified, but must not lead to GL interruption or termination. Providing a denormalized number to GL must yield predictable results.

**Fixed-Point Computation**

Vertex attributes may be specified using a 32-bit two’s-complement signed representation with 16 bits to the right of the binary point (fraction bits).

**General Requirements**

Some calculations require division. In such cases (including implied divisions required by vector normalizations), a division by zero produces an unspecified result but must not lead to GL interruption or termination.
2.1.2 Fixed-Point Data Conversions

When generic vertex attributes and pixel color or depth components are represented as integers, they are often (but not always) considered to be normalized. Normalized integer values are treated specially when being converted to and from floating-point values, and are usually referred to as normalized fixed-point. Such values are always either signed or unsigned.

In the remainder of this section, \( b \) denotes the bit width of the fixed-point integer representation. When the integer is one of the types defined in table 2.2, \( b \) is the minimum required bit width of that type. When the integer is a texture or renderbuffer color or depth component (see section 3.9.3), \( b \) is the number of bits allocated to that component in the internal format of the texture or renderbuffer. When the integer is a framebuffer color or depth component (see section 4), \( b \) is the number of bits allocated to that component in the framebuffer. For framebuffer and renderbuffer A components, \( b \) must be at least 2 if the buffer does not contain an A component, or if there is only 1 bit of A in the buffer.

The signed and unsigned fixed-point representations are assumed to be \( b \)-bit binary twos-complement integers and binary unsigned integers, respectively. The signed fixed-point representation may be treated in one of two ways, as discussed below.

All the conversions described below are performed as defined, even if the implemented range of an integer data type is greater than the minimum required range.

Conversion from Normalized Fixed-Point to Floating-Point

Unsigned normalized fixed-point integers represent numbers in the range \([0, 1]\). The conversion from an unsigned normalized fixed-point value \( c \) to the corresponding floating-point value \( f \) is defined as

\[
f = \frac{c}{2^b - 1}.
\]  

(2.1)

Signed normalized fixed-point integers represent numbers in the range \([-1, 1]\). The conversion from a signed normalized fixed-point value \( c \) to the corresponding floating-point value \( f \) is performed using

\[
f = \max \left\{ \frac{c}{2^b - 1}, -1.0 \right\}.
\]  

(2.2)

Only the range \([-2^{b-1} + 1, 2^{b-1} - 1]\) is used to represent signed fixed-point values in the range \([-1, 1]\). For example, if \( b = 8 \), then the integer value -127 corresponds to -1.0 and the value 127 corresponds to 1.0. Note that while zero can be
2.2. GL STATE

exactly expressed in this representation, one value (-128 in the example) is outside
the representable range, and must be clamped before use. This equation is used ev-
everywhere that signed normalized fixed-point values are converted to floating-point,
including for all signed normalized fixed-point parameters in GL commands, such
as vertex attribute values\(^1\), as well as for specifying texture or framebuffer values
using signed normalized fixed-point.

Conversion from Floating-Point to Normalized Fixed-Point

The conversion from a floating-point value \( f \) to the corresponding unsigned nor-
malized fixed-point value \( c \) is defined by first clamping \( f \) to the range \([0, 1]\), then
computing

\[
f' = f \times (2^b - 1).
\]  

(2.3)

\( f' \) is then cast to an unsigned binary integer value with exactly \( b \) bits.

The conversion from a floating-point value \( f \) to the corresponding signed nor-
malized fixed-point value \( c \) is performed by clamping \( f \) to the range \([-1, 1]\), then
computing

\[
f' = f \times (2^{b-1} - 1).
\]  

(2.4)

After conversion, \( f' \) is then cast to a signed two’s-complement binary integer
value with exactly \( b \) bits.

This equation is used everywhere that floating-point values are converted to
signed normalized fixed-point, including when querying floating-point state (see
section 6) and returning integers\(^2\), as well as for specifying signed normalized tex-
ture or framebuffer values using floating-point.

2.2 GL State

The GL maintains considerable state. This document enumerates each state vari-
able and describes how each variable can be changed. For purposes of discussion,
state variables are categorized somewhat arbitrarily by their function. Although we

\(^1\) This is a behavior change in OpenGL 4.2. In previous versions, a different conversion for
signed normalized values was used in which -128 mapped to -1.0, 127 mapped to 1.0, and 0.0 was
not exactly representable.

\(^2\) This is a behavior change in OpenGL 4.2. In previous versions, a different conversion for
signed normalized values was used in which -128 mapped to -1.0, 127 mapped to 1.0, and 0.0 was
not exactly representable.
describe the operations that the GL performs on the framebuffer, the framebuffer is not a part of GL state.

We distinguish two types of state. The first type of state, called GL server state, resides in the GL server. The majority of GL state falls into this category. The second type of state, called GL client state, resides in the GL client. Unless otherwise specified, all state referred to in this document is GL server state; GL client state is specifically identified. Each instance of a GL context implies one complete set of GL server state; each connection from a client to a server implies a set of both GL client state and GL server state.

While an implementation of the GL may be hardware dependent, this discussion is independent of the specific hardware on which a GL is implemented. We are therefore concerned with the state of graphics hardware only when it corresponds precisely to GL state.

2.2.1 Shared Object State

It is possible for groups of contexts to share certain state. Enabling such sharing between contexts is done through window system binding APIs such as those described in section 1.7.2. These APIs are responsible for creation and management of contexts, and not discussed further here. More detailed discussion of the behavior of shared objects is included in appendix D. Except as defined in this appendix, all state in a context is specific to that context only.

2.3 GL Command Syntax

GL commands are functions or procedures. Various groups of commands perform the same operation but differ in how arguments are supplied to them. To conveniently accommodate this variation, we adopt a notation for describing commands and their arguments.

GL commands are formed from a name which may be followed, depending on the particular command, by a sequence of characters describing a parameter to the command. If present, a digit indicates the required length (number of values) of the indicated type. Next, a string of characters making up one of the type descriptors from table 2.1 indicates the specific size and data type of parameter values. A final \texttt{v} character, if present, indicates that the command takes a pointer to an array (a vector) of values rather than a series of individual arguments. Two specific examples are:

\begin{verbatim}
void Uniform4f( int location, float v0, float v1,
               float v2, float v3);
\end{verbatim}
2.3. GL COMMAND SYNTAX

<table>
<thead>
<tr>
<th>Type Descriptor</th>
<th>Corresponding GL Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>byte</td>
</tr>
<tr>
<td>s</td>
<td>short</td>
</tr>
<tr>
<td>i</td>
<td>int</td>
</tr>
<tr>
<td>i64</td>
<td>int64</td>
</tr>
<tr>
<td>f</td>
<td>float</td>
</tr>
<tr>
<td>d</td>
<td>double</td>
</tr>
<tr>
<td>ub</td>
<td>ubyte</td>
</tr>
<tr>
<td>us</td>
<td>ushort</td>
</tr>
<tr>
<td>ui</td>
<td>uint</td>
</tr>
<tr>
<td>ui64</td>
<td>uint64</td>
</tr>
</tbody>
</table>

Table 2.1: Correspondence of command suffix type descriptors to GL argument types. Refer to table 2.2 for definitions of the GL types.

and

```c
void GetFloatv( enum value, float *data);
```

These examples show the ANSI C declarations for these commands. In general, a command declaration has the form

```c
rtype Name{ε1234}{ε b s i i64 f d ub us ui ui64}{εv}
   ( [args ], T arg1 , . . . , T argN [, args ] );
```

`rtype` is the return type of the function. The braces (`{}`). `} enclose a series of type descriptors (see table 2.1), of which one is selected. `ε` indicates no type descriptor. The arguments enclosed in brackets (`{args },` and `[ args ]`) may or may not be present. The `N` arguments `arg1` through `argN` have type `T`, which corresponds to one of the type descriptors indicated in table 2.1 (if there are no letters, then the arguments’ type is given explicitly). If the final character is not `v`, then `N` is given by the digit `1`, `2`, `3`, or `4` (if there is no digit, then the number of arguments is fixed). If the final character is `v`, then only `arg1` is present and it is an array of `N` values of the indicated type.

For example,

```c
void Uniform{1234}{if}( int location, T value);
```

The declarations shown in this document apply to ANSI C. Languages such as C++ and Ada that allow passing of argument type information admit simpler declarations and fewer entry points.
indicates the eight declarations

```c
void Uniform1i( int location, int value );
void Uniform1f( int location, float value );
void Uniform2i( int location, int v0, int v1 );
void Uniform2f( int location, float v0, float v1 );
void Uniform3i( int location, int v0, int v1, int v2 );
void Uniform3f( int location, float v1, float v2, float v3 );
void Uniform4i( int location, int v0, int v1, int v2, int v3 );
void Uniform4f( int location, float v0, float v1, float v2, float v3 );
```

Arguments whose type is fixed (i.e. not indicated by a suffix on the command) are of one of the GL data types summarized in table 2.2, or pointers to one of these types. Since many GL operations represent bitfields within these types, transfer blocks of data in these types to graphics hardware which uses the same data types, or otherwise requires these sizes, it is not possible to implement the GL API on an architecture which cannot satisfy the exact bit width requirements in table 2.2.

The types clampf and clamped are no longer used, replaced by float and double respectively together with specification language requiring parameter clamping\(^4\).

### 2.3.1 Data Conversion For State-Setting Commands

Many GL commands specify a value or values to which GL state of a specific type (boolean, enum, integer, or floating-point) is to be set. When multiple versions of such a command exist, using the type descriptor syntax described above, any such version may be used to set the state value. When state values are specified using a different parameter type than the actual type of that state, data conversions are performed as follows:

- When the type of internal state is boolean, zero integer or floating-point values are converted to **FALSE** and non-zero values are converted to **TRUE**.

- When the type of internal state is integer or enum, boolean values of **FALSE** and **TRUE** are converted to 0 and 1, respectively. Floating-point values are rounded to the nearest integer.

\(^4\) These changes are completely backwards-compatible and will eventually be propagated to man pages and header files as well.

OpenGL 4.2 (Core Profile) - April 27, 2012
## 2.3. GL COMMAND SYNTAX

<table>
<thead>
<tr>
<th>GL Type</th>
<th>Bit Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1 or more</td>
<td>Boolean</td>
</tr>
<tr>
<td>byte</td>
<td>8</td>
<td>Signed two’s complement binary integer</td>
</tr>
<tr>
<td>ubyte</td>
<td>8</td>
<td>Unsigned binary integer</td>
</tr>
<tr>
<td>char</td>
<td>8</td>
<td>Characters making up strings</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>Signed two’s complement binary integer</td>
</tr>
<tr>
<td>ushort</td>
<td>16</td>
<td>Unsigned binary integer</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>Signed two’s complement binary integer</td>
</tr>
<tr>
<td>uint</td>
<td>32</td>
<td>Unsigned binary integer</td>
</tr>
<tr>
<td>fixed</td>
<td>32</td>
<td>Signed 2’s complement 16.16 scaled integer</td>
</tr>
<tr>
<td>int64</td>
<td>64</td>
<td>Signed two’s complement binary integer</td>
</tr>
<tr>
<td>uint64</td>
<td>64</td>
<td>Unsigned binary integer</td>
</tr>
<tr>
<td>sizei</td>
<td>32</td>
<td>Non-negative binary integer size</td>
</tr>
<tr>
<td>enum</td>
<td>32</td>
<td>Enumerated binary integer value</td>
</tr>
<tr>
<td>intptr</td>
<td>ptrbits</td>
<td>Signed two’s complement binary integer</td>
</tr>
<tr>
<td>sizeiptr</td>
<td>ptrbits</td>
<td>Non-negative binary integer size</td>
</tr>
<tr>
<td>sync</td>
<td>ptrbits</td>
<td>Sync object handle (see section 5.3)</td>
</tr>
<tr>
<td>bitfield</td>
<td>32</td>
<td>Bit field</td>
</tr>
<tr>
<td>half</td>
<td>16</td>
<td>Half-precision floating-point value encoded in an unsigned scalar</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>Floating-point value</td>
</tr>
<tr>
<td>clampf</td>
<td>32</td>
<td>Floating-point value clamped to [0, 1]</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>Floating-point value</td>
</tr>
<tr>
<td>clampd</td>
<td>64</td>
<td>Floating-point value clamped to [0, 1]</td>
</tr>
</tbody>
</table>

Table 2.2: GL data types. GL types are not C types. Thus, for example, GL type `int` is referred to as `GLint` outside this document, and is not necessarily equivalent to the C type `int`. An implementation must use exactly the number of bits indicated in the table to represent a GL type. `ptrbits` is the number of bits required to represent a pointer type; in other words, types `intptr`, `sizeiptr`, and `sync` must be sufficiently large as to store any address.
When the type of internal state is floating-point, boolean values of `FALSE` and `TRUE` are converted to 0.0 and 1.0, respectively. Integer values are converted to floating-point.

For commands taking arrays of the specified type, these conversions are performed for each element of the passed array.

Each command following these conversion rules refers back to this section. Some commands have additional conversion rules specific to certain state values and data types, which are described following the reference.

Validation of values performed by state-setting commands is performed after conversion, unless specified otherwise for a specific command.

### 2.4 Basic GL Operation

Figure 2.1 shows a schematic diagram of the GL. Commands enter the GL on the left. Some commands specify geometric objects to be drawn while others control how the objects are handled by the various stages. Commands are effectively sent through a processing pipeline.

The first stage operates on geometric primitives described by vertices: points, line segments, and polygons. In this stage vertices may be transformed and lit, followed by assembly into geometric primitives, which may optionally be used by the next stage, geometry shading, to generate new primitives. The final resulting primitives are clipped to a clip volume in preparation for the next stage, rasterization. The rasterizer produces a series of framebuffer addresses and values using a two-dimensional description of a point, line segment, or polygon. Each fragment so produced is fed to the next stage that performs operations on individual fragments before they finally alter the framebuffer. These operations include conditional updates into the framebuffer based on incoming and previously stored depth values (to effect depth buffering), blending of incoming fragment colors with stored colors, as well as masking and other logical operations on fragment values.

Finally, values may also be read back from the framebuffer or copied from one portion of the framebuffer to another. These transfers may include some type of decoding or encoding.

This ordering is meant only as a tool for describing the GL, not as a strict rule of how the GL is implemented, and we present it only as a means to organize the various operations of the GL.
Figure 2.1. Block diagram of the GL.
2.5 GL Errors

The GL detects only a subset of those conditions that could be considered errors. This is because in many cases error checking would adversely impact the performance of an error-free program.

The command

```c
enum GetError(void);
```

is used to obtain error information. Each detectable error is assigned a numeric code. When an error is detected, a flag is set and the code is recorded. Further errors, if they occur, do not affect this recorded code. When `GetError` is called, the code is returned and the flag is cleared, so that a further error will again record its code. If a call to `GetError` returns `NO_ERROR`, then there has been no detectable error since the last call to `GetError` (or since the GL was initialized).

To allow for distributed implementations, there may be several flag-code pairs. In this case, after a call to `GetError` returns a value other than `NO_ERROR` each subsequent call returns the non-zero code of a distinct flag-code pair (in unspecified order), until all non-`NO_ERROR` codes have been returned. When there are no more non-`NO_ERROR` error codes, all flags are reset. This scheme requires some positive number of pairs of a flag bit and an integer. The initial state of all flags is cleared and the initial value of all codes is `NO_ERROR`.

Table 2.3 summarizes GL errors. Currently, when an error flag is set, results of GL operation are undefined only if `OUT_OF_MEMORY` has occurred. In other cases, the command generating the error is ignored so that it has no effect on GL state or framebuffer contents. Except where otherwise noted, if the generating command returns a value, it returns zero. If the generating command modifies values through a pointer argument, no change is made to these values. These error semantics apply only to GL errors, not to system errors such as memory access errors. This behavior is the current behavior; the action of the GL in the presence of errors is subject to change.

Several error generation conditions are implicit in the description of every GL command:

- If a command that requires an enumerated value is passed a symbolic constant that is not one of those specified as allowable for that command, the error `INVALID_ENUM` is generated. This is the case even if the argument is a pointer to a symbolic constant, if the value pointed to is not allowable for the given command.

- If a negative number is provided where an argument of type `sizei` or `sizeiptr` is specified, the error `INVALID_VALUE` is generated.

OpenGL 4.2 (Core Profile) - April 27, 2012
### 2.6. PRIMITIVES AND VERTICES

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
<th>Offending command ignored?</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVALID_ENUM</td>
<td>enum argument out of range</td>
<td>Yes</td>
</tr>
<tr>
<td>INVALID_VALUE</td>
<td>Numeric argument out of range</td>
<td>Yes</td>
</tr>
<tr>
<td>INVALID_OPERATION</td>
<td>Operation illegal in current state</td>
<td>Yes</td>
</tr>
<tr>
<td>INVALID_FRAMEBUFFER_OPERATION</td>
<td>Framebuffer object is not complete</td>
<td>Yes</td>
</tr>
<tr>
<td>OUT_OF_MEMORY</td>
<td>Not enough memory left to execute command</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

Table 2.3: Summary of GL errors

- If memory is exhausted as a side effect of the execution of a command, the **error OUT_OF_MEMORY** may be generated.

Otherwise, errors are generated only for conditions that are explicitly described in this specification.

### 2.6 Primitives and Vertices

In the GL, most geometric objects are drawn by specifying a series of generic attribute sets using **DrawArrays** or one of the other drawing commands defined in section 2.8.3. Points, lines, polygons, and a variety of related geometric objects (see section 2.6.1) can be drawn in this way.

Each vertex is specified with one or more generic vertex attributes. Each attribute is specified with one, two, three, or four scalar values. Generic vertex attributes can be accessed from within vertex shaders (section 2.11) and used to compute values for consumption by later processing stages.

The methods by which generic attributes are sent to the GL, as well as how attributes are used by vertex shaders to generate vertices mapped to the two-dimensional screen, are discussed later.

Before vertex shader execution, the state required by a vertex is its generic vertex attributes. Vertex shader execution processes vertices producing a homogeneous vertex position and any outputs explicitly written by the vertex shader.

Figure 2.2 shows the sequence of operations that builds a primitive (point, line segment, or polygon) from a sequence of vertices. After a primitive is formed, it is clipped to a **clip volume**. This may alter the primitive by altering vertex coordinates and vertex shader outputs. In the case of line and polygon primi-
2.6. PRIMITIVES AND VERTICES

Figure 2.2. Vertex processing and primitive assembly.
2.6. PRIMITIVES AND VERTICES

tives, clipping may insert new vertices into the primitive. The vertices defining a primitive to be rasterized have output variables associated with them.

2.6.1 Primitive Types

A sequence of vertices is passed to the GL using DrawArrays or one of the other drawing commands defined in section 2.8.3. There is no limit to the number of vertices that may be specified, other than the size of the vertex arrays. The mode parameter of these commands determines the type of primitives to be drawn using the vertices. The types, and the corresponding mode parameters, are:

Points
A series of individual points may be specified with mode POINTS. Each vertex defines a separate point.

Line Strips
A series of one or more connected line segments may be specified with mode LINE_STRIP. In this case, the first vertex specifies the first segment’s start point while the second vertex specifies the first segment’s endpoint and the second segment’s start point. In general, the ith vertex (for i > 1) specifies the beginning of the ith segment and the end of the i − 1st. The last vertex specifies the end of the last segment. If only one vertex is specified, then no primitive is generated.

The required state consists of the processed vertex produced from the last vertex that was sent (so that a line segment can be generated from it to the current vertex), and a boolean flag indicating if the current vertex is the first vertex.

Line Loops
Line loops may be specified with mode LINE_LOOP. Loops are the same as line strips except that a final segment is added from the final specified vertex to the first vertex. The required state consists of the processed first vertex, in addition to the state required for line strips.

Separate Lines
Individual line segments, each specified by a pair of vertices, may be specified with mode LINES. The first two vertices passed define the first segment, with subsequent pairs of vertices each defining one more segment. If the number of specified vertices is odd, then the last one is ignored. The state required is the same as for line strips but it is used differently: a processed vertex holding the first vertex of the current segment, and a boolean flag indicating whether the current vertex is odd or even (a segment start or end).
2.6. PRIMITIVES AND VERTICES

Triangle Strips

A triangle strip is a series of triangles connected along shared edges, and may be specified with mode TRIANGLE_STRIP. In this case, the first three vertices define the first triangle (and their order is significant). Each subsequent vertex defines a new triangle using that point along with two vertices from the previous triangle. If fewer than three vertices are specified, no primitive is produced. See figure 2.3.

The required state consists of a flag indicating if the first triangle has been completed, two stored processed vertices, (called vertex A and vertex B), and a one bit pointer indicating which stored vertex will be replaced with the next vertex. The pointer is initialized to point to vertex A. Each successive vertex toggles the pointer. Therefore, the first vertex is stored as vertex A, the second stored as vertex B, the third stored as vertex A, and so on. Any vertex after the second one sent forms a triangle from vertex A, vertex B, and the current vertex (in that order).

Triangle Fans

A triangle fan is the same as a triangle strip with one exception: each vertex after the first always replaces vertex B of the two stored vertices. A triangle fan may be specified with mode TRIANGLE_FAN.

Separate Triangles

Separate triangles are specified with mode TRIANGLES. In this case, The \( 3i + \)
2.6. PRIMITIVES AND VERTICES

Figure 2.4. Lines with adjacency (a) and line strips with adjacency (b). The vertices connected with solid lines belong to the main primitives; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

1st, 3i + 2nd, and 3i + 3rd vertices (in that order) determine a triangle for each \( i = 0, 1, \ldots, n - 1 \), where there are \( 3n + k \) vertices drawn. \( k \) is either 0, 1, or 2; if \( k \) is not zero, the final \( k \) vertices are ignored. For each triangle, vertex A is vertex \( 3i \) and vertex B is vertex \( 3i + 1 \). Otherwise, separate triangles are the same as a triangle strip.

**Lines with Adjacency**

Lines with adjacency are independent line segments where each endpoint has a corresponding adjacent vertex that can be accessed by a geometry shader (section 2.13). If a geometry shader is not active, the adjacent vertices are ignored. They are generated with mode `LINES_ADJACENCY`.

A line segment is drawn from the \( 4i + 2 \)nd vertex to the \( 4i + 3 \)rd vertex for each \( i = 0, 1, \ldots, n - 1 \), where there are \( 4n + k \) vertices passed. \( k \) is either 0, 1, 2, or 3; if \( k \) is not zero, the final \( k \) vertices are ignored. For line segment \( i \), the \( 4i + 1 \)st and \( 4i + 4 \)th vertices are considered adjacent to the \( 4i + 2 \)nd and \( 4i + 3 \)rd vertices, respectively (see figure 2.4).

**Line Strips with Adjacency**
Figure 2.5. Triangles with adjacency. The vertices connected with solid lines belong to the main primitive; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

Line strips with adjacency are similar to line strips, except that each line segment has a pair of adjacent vertices that can be accessed by a geometry shader. If a geometry shader is not active, the adjacent vertices are ignored. They are generated with `mode` `LINE_STRIP_ADJACENCY`.

A line segment is drawn from the \( i + 2 \)nd vertex to the \( i + 3 \)rd vertex for each \( i = 0, 1, \ldots, n - 1 \), where there are \( n + 3 \) vertices passed. If there are fewer than four vertices, all vertices are ignored. For line segment \( i \), the \( i + 1 \)st and \( i + 4 \)th vertex are considered adjacent to the \( i + 2 \)nd and \( i + 3 \)rd vertices, respectively (see figure 2.4).

**Triangles with Adjacency**

Triangles with adjacency are similar to separate triangles, except that each triangle edge has an adjacent vertex that can be accessed by a geometry shader. If a geometry shader is not active, the adjacent vertices are ignored. They are generated with `mode` `TRIANGLES_ADJACENCY`.

The \( 6i + 1 \)st, \( 6i + 3 \)rd, and \( 6i + 5 \)th vertices (in that order) determine a triangle for each \( i = 0, 1, \ldots, n - 1 \), where there are \( 6n + k \) vertices passed. \( k \) is either 0, 1, 2, 3, 4, or 5; if \( k \) is non-zero, the final \( k \) vertices are ignored. For triangle \( i \), the \( i + 2 \)nd, \( i + 4 \)th, and \( i + 6 \)th vertices are considered adjacent to edges from the \( i + 1 \)st to the \( i + 3 \)rd, from the \( i + 3 \)rd to the \( i + 5 \)th, and from the \( i + 5 \)th to the
Triangle Strips with Adjacency

Triangle strips with adjacency are similar to triangle strips, except that each line triangle edge has an adjacent vertex that can be accessed by a geometry shader. If a geometry shader is not active, the adjacent vertices are ignored. They are generated with mode **TRIANGLE_STRIP_ADJACENCY**.

In triangle strips with adjacency, *n* triangles are drawn where there are \(2(n + 2) + k\) vertices passed. *k* is either 0 or 1; if *k* is 1, the final vertex is ignored. If there are fewer than 6 vertices, the entire primitive is ignored. Table 2.4 describes the vertices and order used to draw each triangle, and which vertices are considered adjacent to each edge of the triangle (see figure 2.6).
2.6. PRIMITIVES AND VERTICES

Primitive Vertices

<table>
<thead>
<tr>
<th>Primitive</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>only ((i = 0, n = 1))</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>first ((i = 0))</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>middle ((i \text{ odd}))</td>
<td>(2i + 3)</td>
<td>(2i + 1)</td>
<td>(2i + 5)</td>
</tr>
<tr>
<td>middle ((i \text{ even}))</td>
<td>(2i + 1)</td>
<td>(2i + 3)</td>
<td>(2i + 5)</td>
</tr>
<tr>
<td>last ((i = n - 1, i \text{ odd}))</td>
<td>(2i + 3)</td>
<td>(2i + 1)</td>
<td>(2i + 5)</td>
</tr>
<tr>
<td>last ((i = n - 1, i \text{ even}))</td>
<td>(2i + 1)</td>
<td>(2i + 3)</td>
<td>(2i + 5)</td>
</tr>
</tbody>
</table>

Adjacent Vertices

<table>
<thead>
<tr>
<th>Primitive</th>
<th>1/2</th>
<th>2/3</th>
<th>3/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>only ((i = 0, n = 1))</td>
<td>2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>first ((i = 0))</td>
<td>2</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>middle ((i \text{ odd}))</td>
<td>(2i - 1)</td>
<td>(2i + 4)</td>
<td>(2i + 7)</td>
</tr>
<tr>
<td>middle ((i \text{ even}))</td>
<td>(2i - 1)</td>
<td>(2i + 7)</td>
<td>(2i + 4)</td>
</tr>
<tr>
<td>last ((i = n - 1, i \text{ odd}))</td>
<td>(2i - 1)</td>
<td>(2i + 4)</td>
<td>(2i + 6)</td>
</tr>
<tr>
<td>last ((i = n - 1, i \text{ even}))</td>
<td>(2i - 1)</td>
<td>(2i + 6)</td>
<td>(2i + 4)</td>
</tr>
</tbody>
</table>

Table 2.4: Triangles generated by triangle strips with adjacency. Each triangle is drawn using the vertices whose numbers are in the 1st, 2nd, and 3rd columns under primitive vertices, in that order. The vertices in the 1/2, 2/3, and 3/1 columns under adjacent vertices are considered adjacent to the edges from the first to the second, from the second to the third, and from the third to the first vertex of the triangle, respectively. The six rows correspond to six cases: the first and only triangle \((i = 0, n = 1)\), the first triangle of several \((i = 0, n > 0)\), “odd” middle triangles \((i = 1, 3, 5 \ldots)\), “even” middle triangles \((i = 2, 4, 6 \ldots)\), and special cases for the last triangle, when \(i\) is either even or odd. For the purposes of this table, the first vertex passed is numbered 1 and the first triangle is numbered 0.

Separate Patches

A patch is an ordered collection of vertices used for primitive tessellation (section 2.12). The vertices comprising a patch have no implied geometric ordering. The vertices of a patch are used by tessellation shaders and a fixed-function tessellator to generate new point, line, or triangle primitives. Separate patches are generated with mode PATCHES.

Each patch in the series has a fixed number of vertices, which is specified by calling

```c
void PatchParameteri(enum pname, int value);
```

with `pname` set to PATCH_VERTICES. The error INVALID_VALUE is generated if `value` is less than or equal to zero or is greater than the implementation-dependent maximum patch size (the value of MAX_PATCH_VERTICES). The patch size is initially three vertices.

If the number of vertices in a patch is given by \(v\), the \(vi + 1\)st through \(vi + v\)th vertices (in that order) determine a patch for each \(i = 0, 1, \ldots, n - 1\), where there are \(vn + k\) vertices. \(k\) is in the range \([0, v - 1]\); if \(k\) is not zero, the final \(k\) vertices are ignored.
General Considerations For Polygon Primitives

Depending on the current state of the GL, a polygon primitive generated from a drawing command with mode TRIANGLE_FAN, TRIANGLE_STRIP, TRIANGLES, TRIANGLES_ADJACENCY, or TRIANGLE_STRIP_ADJACENCY may be rendered in one of several ways, such as outlining its border or filling its interior. The order of vertices in such a primitive is significant in polygon rasterization and fragment shading (see sections 3.6.1 and 3.10.2).

2.7 Vertex Specification

Vertex shaders (see section 2.11) access an array of 4-component generic vertex attributes. The first slot of this array is numbered 0, and the size of the array is specified by the implementation-dependent constant MAX_VERTEX_ATTRIBS.

Current generic attribute values define generic attributes for a vertex when a vertex array defining that data is not enabled, as described in section 2.8. The current values of a generic shader attribute declared as a floating-point scalar, vector, or matrix may be changed at any time by issuing one of the commands

```c
void VertexAttrib{1234}{sfd}(uint index, T values);
void VertexAttrib{123}{sfd}v(uint index, const T values);
void VertexAttrib4{bsifd ub us ui}v(uint index, const T values);
void VertexAttrib4Nub(uint index, T values);
void VertexAttrib4N{bsi ub us ui}v(uint index, const T values);
void VertexAttribI{1234}{i ui}(uint index, T values);
void VertexAttribI{1234}{i ui}v(uint index, const T values);
void VertexAttribI4{b s ub us}v(uint index, const T values);
void VertexAttribL{1234}d(uint index, T values);
void VertexAttribL{1234}dv(uint index, T values);
void VertexAttribP{1234}ui(uint index, enum type, boolean normalized, uint value);
void VertexAttribP{1234}uiv(uint index, enum type, boolean normalized, const uint *value);
```

The `VertexAttrib4N*` commands specify fixed-point values that are converted
2.7. VERTEX SPECIFICATION

to a normalized $[0,1]$ or $[-1,1]$ range as described in equations 2.1 and 2.2, respectively.

The VertexAttribI* commands specify signed or unsigned fixed-point values that are stored as signed or unsigned integers, respectively. Such values are referred to as pure integers.

The VertexAttribL* commands specify double-precision values that will be stored as double-precision values.

The VertexAttribP* commands specify up to four attribute component values packed into a single natural type as described in section 2.8.2. type must be INT_2_10_10_10_REV or UNSIGNED_INT_2_10_10_10_REV, specifying signed or unsigned data respectively. The first one ($x$), two ($x,y$), three ($x,y,z$), or four ($x,y,z,w$) components of the packed data are consumed by VertexAttribP1ui, VertexAttribP2ui, VertexAttribP3ui, and VertexAttribP4ui, respectively. If normalized is TRUE, signed or unsigned components are converted to floating-point by normalizing to $[-1,1]$ or $[0,1]$ respectively. If normalized is false, components are cast directly to floating-point. For VertexAttribP*uiv, value contains the address of a single uint containing the packed attribute components.

All other VertexAttrib* commands specify values that are converted directly to the internal floating-point representation.

The resulting value(s) are loaded into the generic attribute at slot index, whose components are named $x$, $y$, $z$, and $w$. The VertexAttrib1* family of commands sets the $x$ coordinate to the provided single argument while setting $y$ and $z$ to 0 and $w$ to 1. Similarly, VertexAttrib2* commands set $x$ and $y$ to the specified values, $z$ to 0 and $w$ to 1; VertexAttrib3* commands set $x$, $y$, and $z$, with $w$ set to 1; and VertexAttrib4* commands set all four coordinates.

The VertexAttrib* entry points may also be used to load shader attributes declared as a floating-point matrix. Each column of a matrix takes up one generic 4-component attribute slot out of the MAX_VERTEX_ATTRIBS available slots. Matrices are loaded into these slots in column major order. Matrix columns are loaded in increasing slot numbers.

For all VertexAttrib* commands, the error INVALID_VALUE is generated if index is greater than or equal to the value of MAX_VERTEX_ATTRIBS.

When values for a vertex shader attribute variable are sourced from a current generic attribute value, the attribute must be specified by a command compatible with the data type of the variable. The values loaded into a shader attribute variable bound to generic attribute index are undefined if the current value for attribute index was not specified by

- VertexAttrib[1234]* or VertexAttribP*, for single-precision floating-point
2.8. VERTEX ARRAYS

scalar, vector, and matrix types

- VertexAttribI[1234]i or VertexAttribI[1234]iv, for signed integer scalar and vector types
- VertexAttribI[1234]ui or VertexAttribI[1234]uiv, for unsigned integer scalar and vector types
- VertexAttribL*, for double-precision floating-point scalar and vector types.

The state required to support vertex specification consists of the value of MAX_VERTEX_ATTRIBS four-component vectors to store generic vertex attributes. The initial values for all generic vertex attributes are (0.0, 0.0, 0.0, 1.0).

2.8 Vertex Arrays

Vertex data are placed into arrays that are stored in the server’s address space (described in section 2.9). Blocks of data in these arrays may then be used to specify multiple geometric primitives through the execution of a single GL command. The client may specify up to the value of MAX_VERTEX_ATTRIBS arrays to store one or more generic vertex attributes. The commands

```c
void VertexAttribPointer( uint index, int size, enum type, 
    boolean normalized, sizei stride, const
    void *pointer );
```

```c
void VertexAttribIPointer( uint index, int size, enum type, 
    sizei stride, const void *pointer );
```

```c
void VertexAttribLPointer( uint index, int size, enum type, 
    sizei stride, const void *pointer );
```

describe the locations and organizations of these arrays. For each command, type specifies the data type of the values stored in the array. size indicates the number of values per vertex that are stored in the array as well as their component ordering. Table 2.5 indicates the allowable values for size and type (when present). For type the values BYTE, SHORT, INT, FIXED, FLOAT, HALF_FLOAT, and DOUBLE indicate types byte, short, int, fixed, float, half, and double, respectively; the values UNSIGNED_BYTE, UNSIGNED_SHORT, and UNSIGNED_INT indicate types ubyte, ushort, and uint, respectively; and the values INT_2_10_10_10_REV and UNSIGNED_INT_2_10_10_10_REV indicate respectively four signed or unsigned elements packed into a single uint, both correspond to the term packed in that table.

OpenGL 4.2 (Core Profile) - April 27, 2012
An **INVALID_VALUE** error is generated if `size` is not one of the values allowed in table 2.5 for the corresponding command.

An **INVALID_OPERATION** error is generated under any of the following conditions:

- if no vertex array object is currently bound (see section 2.10);
- `size` is **BGRA** and **type** is not **UNSIGNED_BYTE**, **INT_2_10_10_10_REV** or **UNSIGNED_INT_2_10_10_10_REV**;
- **type** is **INT_2_10_10_10_REV** or **UNSIGNED_INT_2_10_10_10_REV**, and `size` is neither 4 or **BGRA**;
- `size` is **BGRA** and **normalized** is `FALSE`;
- any of the ***Pointer** commands specifying the location and organization of vertex array data are called while `zero` is bound to the **ARRAY_BUFFER** buffer object binding point (see section 2.9.6), and the **pointer** argument is not **NULL**.

The **index** parameter in the **VertexAttrib*Pointer** commands identifies the generic vertex attribute array being described. The error **INVALID_VALUE** is generated if `index` is greater than or equal to the value of **MAX_VERTEX_ATTRIBS**. Generic attribute arrays with integer **type** arguments can be handled in one of three ways: converted to float by normalizing to \([0, 1]\) or \([-1, 1]\) as described in equations 2.1 and 2.2, respectively; converted directly to float, or left as integers. Data for an array specified by **VertexAttribPointer** will be converted to floating-point by normalizing if `normalized` is **TRUE**, and converted directly to floating-point otherwise. Data for an array specified by **VertexAttribIPointer** will always be left as integer values; such data are referred to as **pure** integers. Data for an array specified by **VertexAttribLPointer** must be specified as double-precision floating-point values. An **INVALID_ENUM** error will be generated by **VertexAttribLPointer** if **type** is not **DOUBLE**.

The one, two, three, or four values in an array that correspond to a single vertex comprise an array **element**. When `size` is **BGRA**, it indicates four values. The values within each array element are stored sequentially in memory. However, if `size` is **BGRA**, the first, second, third, and fourth values of each array element are taken from the third, second, first, and fourth values in memory respectively. If `stride` is specified as zero, then array elements are stored sequentially as well. The error **INVALID_VALUE** is generated if `stride` is negative. Otherwise pointers to the **ith** and \((i + 1)st\) elements of an array differ by `stride` basic machine units (typically unsigned bytes), the pointer to the \((i + 1)st\) element being greater. For each
### 2.8. VERTEX ARRAYS

<table>
<thead>
<tr>
<th>Command</th>
<th>Sizes and Component Ordering</th>
<th>Integer Handling</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>VertexAttribPointer</td>
<td>1, 2, 3, 4, BGRA</td>
<td>flag</td>
<td>byte, ubyte, short, ushort, int, uint, fixed, float, half, double, <strong>packed</strong></td>
</tr>
<tr>
<td>VertexAttrib1Pointer</td>
<td>1, 2, 3, 4</td>
<td>integer</td>
<td>byte, ubyte, short, ushort, int, uint</td>
</tr>
<tr>
<td>VertexAttrib1LPointer</td>
<td>1, 2, 3, 4</td>
<td>n/a</td>
<td>double</td>
</tr>
</tbody>
</table>

Table 2.5: Vertex array sizes (values per vertex) and data types. The “Integer Handling” column indicates how fixed-point data types are handled: “integer” means that they remain as integer values, and “flag” means that they are either converted to floating-point directly, or converted by normalizing to [0, 1] (for unsigned types) or [−1, 1] (for signed types), depending on the setting of the normalized flag in `VertexAttribPointer`. If `size` is `BGRA`, vertex array values are always normalized, irrespective of the “normalize” table entry. `packed` is not a GL type, but indicates commands accepting multiple components packed into a single `uint`. 

OpenGL 4.2 (Core Profile) - April 27, 2012
2.8. VERTEX ARRAYS

command, \textit{pointer} specifies the offset within a buffer of the first value of the first element of the array being specified.

When values for a vertex shader attribute variable are sourced from an enabled generic vertex attribute array, the array must be specified by a command compatible with the data type of the variable. The values loaded into a shader attribute variable bound to generic attribute \textit{index} are undefined if the array for \textit{index} was not specified by:

- \textbf{VertexAttribPointer}, for single-precision floating-point scalar, vector, and matrix types
- \textbf{VertexAttribIPointer} with \textit{type} BYTE, SHORT, or INT for signed integer scalar and vector types
- \textbf{VertexAttribIPointer} with \textit{type} UNSIGNED_BYTE, UNSIGNED_SHORT, or UNSIGNED_INT for unsigned integer scalar and vector types
- \textbf{VertexAttribLPointer}, for double-precision floating-point scalar and vector types.

An individual generic vertex attribute array is enabled or disabled by calling one of

\begin{verbatim}
void EnableVertexAttribArray( uint index );
void DisableVertexAttribArray( uint index );
\end{verbatim}

where \textit{index} identifies the generic vertex attribute array to enable or disable.

An \texttt{INVALID\_VALUE} error is generated if \textit{index} is greater than or equal to \texttt{MAX\_VERTEX\_ATTRIBS}.

An \texttt{INVALID\_OPERATION} error is generated if no vertex array object is bound.

The command

\begin{verbatim}
void VertexAttribDivisor( uint index, uint divisor );
\end{verbatim}

modifies the rate at which generic vertex attributes advance when rendering multiple instances of primitives in a single draw call. If \textit{divisor} is zero, the attribute at slot \textit{index} advances once per vertex. If \textit{divisor} is non-zero, the attribute advances once per \textit{divisor} instances of the primitives being rendered. An attribute is referred to as \textit{instanced} if its \textit{divisor} value is non-zero.

An \texttt{INVALID\_VALUE} error is generated if \textit{index} is greater than or equal to the value of \texttt{MAX\_VERTEX\_ATTRIBS}.

An \texttt{INVALID\_OPERATION} error is generated if no vertex array object is bound.
2.8. VERTEX ARRAYS

2.8.1 Transferring Array Elements

When an array element $i$ is transferred to the GL by DrawArrays, DrawElements, or the other Draw* commands described below, each generic attribute is expanded to four components. If size is one then the $x$ component of the attribute is specified by the array; the $y$, $z$, and $w$ components are implicitly set to 0, 0, and 1, respectively. If size is two then the $x$ and $y$ components of the attribute are specified by the array; the $z$ and $w$ components are implicitly set to 0 and 1, respectively. If size is three then $x$, $y$, and $z$ are specified, and $w$ is implicitly set to 1. If size is four then all components are specified.

Primitive restarting is enabled or disabled by calling one of the commands

```c
void Enable( enum target );
```

and

```c
void Disable( enum target );
```

with target PRIMITIVE_RESTART. The command

```c
void PrimitiveRestartIndex( uint index );
```

specifies a vertex array element that is treated specially when primitive restarting is enabled. This value is called the primitive restart index. When one of the Draw* commands transfers a set of generic attribute array elements to the GL, if the index within the vertex arrays corresponding to that set is equal to the primitive restart index, then the GL does not process those elements as a vertex. Instead, it is as if the drawing command ended with the immediately preceding transfer, and another drawing command is immediately started with the same parameters, but only transferring the immediately following element through the end of the originally specified elements.

When one of the *BaseVertex drawing commands specified in section 2.8.3 is used, the primitive restart comparison occurs before the basevertex offset is added to the array index.

2.8.2 Packed Vertex Data Formats

UNSIGNED_INT_2_10_10_10_REV and INT_2_10_10_10_REV vertex data formats describe packed, 4 component formats stored in a single 32-bit word.

For the UNSIGNED_INT_2_10_10_10_REV vertex data format, the first ($x$), second ($y$), and third ($z$) components are represented as 10-bit unsigned integer
2.8. VERTEX ARRAYS

values and the fourth \((w)\) component is represented as a 2-bit unsigned integer value.

For the INT_2_10_10_10_REV vertex data format, the \(x\), \(y\) and \(z\) components are represented as 10-bit signed two’s complement integer values and the \(w\) component is represented as a 2-bit signed two’s complement integer value.

The normalized value is used to indicate whether to normalize the data to \([0, 1]\) (for unsigned types) or \([-1, 1]\) (for signed types). During normalization, the conversion rules specified in equations 2.1 and 2.2 are followed.

Tables 2.6 and 2.7 describe how these components are laid out in a 32-bit word.

Table 2.6: Packed component layout for non-BGRA formats. Bit numbers are indicated for each component.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| \(w\) | \(z\) | \(y\) | \(x\) |

Table 2.7: Packed component layout for BGRA format. Bit numbers are indicated for each component.

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| \(w\) | \(x\) | \(y\) | \(z\) |

2.8.3 Drawing Commands

The command

\[
\text{void DrawArraysOneInstance( enum mode, int first, size\_count, int instance, uint baseinstance );}
\]

does not exist in the GL, but is used to describe functionality in the rest of this section. This command constructs a sequence of geometric primitives by transferring elements \(first\) through \(first + count - 1\) of each enabled non-instanced array to the GL. \(mode\) specifies what kind of primitives are constructed, as defined in section 2.6.1. If \(mode\) is not a valid primitive type, an INVALID_ENUM error is generated. If \(count\) is negative, an INVALID_VALUE error is generated.

The value of \(instance\) may be read by a vertex shader as \(\text{gl\_InstanceID}\), as described in section 2.11.12.
If an enabled vertex attribute array is instanced (it has a non-zero \textit{divisor} as specified by \texttt{VertexAttribDivisor}), the element that is transferred to the GL, for all vertices, is given by

\[
\left\lfloor \frac{\text{instance}}{\text{divisor}} \right\rfloor + \text{baseinstance}
\]

If an array corresponding to a generic attribute required by a vertex shader is not enabled, then the corresponding element is taken from the current generic attribute state (see section 2.7). Otherwise, if an array is enabled, the corresponding current generic attribute value is unaffected by the execution of \texttt{DrawArraysOneInstance}.

If no vertex array object is bound (see section 2.10), an \texttt{INVALID_OPERATION} error is generated.

Specifying \textit{first} < 0 results in undefined behavior. Generating the error \texttt{INVALID_VALUE} is recommended in this case.

The command

\[
\texttt{void DrawArrays( enum mode, int first, sizei count )};
\]

is equivalent to the command sequence

\[
\texttt{DrawArraysOneInstance( mode, first, count, 0, 0) ;}
\]

The command

\[
\texttt{void DrawArraysInstancedBaseInstance( enum mode, int first, sizei count, sizei primcount, uint baseinstance )};
\]

behaves identically to \texttt{DrawArrays} except that \textit{primcount} instances of the range of elements are executed and the value of \textit{instance} advances for each iteration. Those attributes that have positive values for \textit{divisor}, as specified by \texttt{VertexAttribDivisor}, advance once every \textit{divisor} instances. Additionally, the first element within those instanced vertex attributes is specified in \textit{baseinstance}. Thus, the element transferred from instanced vertex attributes is given by:

\[
\left\lfloor \frac{\text{instance}}{\text{divisor}} \right\rfloor + \text{baseinstance}
\]

\texttt{DrawArraysInstancedBaseInstance} has the same effect as:
if (mode, count, or primcount is invalid) 
generate appropriate error 
else {
    for (i = 0; i < primcount; i++) {
        DrawArraysOneInstance (mode, first, count, i, 
                                baseinstance);
    }
}

The command

void DrawArraysInstanced( enum mode, int first, 
                        usize count, usize primcount);

is equivalent to the command sequence

DrawArraysInstancedBaseInstance (mode, first, count, primcount, 0);

The command

void DrawArraysIndirect( enum mode, const 
                         void *indirect);

has the same effect as:

typedef struct {
    uint count;
    uint primCount;
    uint first;
    uint baseInstance;
} DrawArraysIndirectCommand;

DrawArraysIndirectCommand *cmd =
    (DrawArraysIndirectCommand *)indirect;
DrawArraysInstancedBaseInstance (mode, cmd->first, cmd->count, 
                                 cmd->primCount, cmd->baseInstance);

Unlike DrawArraysInstanced, first is unsigned and cannot cause an error. An 
INVALID_OPERATION error is generated if zero is bound to DRAW INDIRECT BUFFER.

All elements of DrawArraysIndirectCommand are tightly packed 32 bit values.

The command
void **MultiDrawArrays**(enum mode, const int *first, const sizei *count, sizei primcount);

behaves identically to **DrawArraysInstanced** except that primcount separate ranges of elements are specified instead, all elements are treated as though they are not instanced, and the value of instance remains zero. It has the same effect as:

```c
if (mode or primcount is invalid)
genenerate appropriate error
else {
    for (i = 0; i < primcount; i++) {
        if (count[i] > 0)
            DrawArraysOneInstance(mode, first[i], count[i], 0, 0);
    }
}
```

The command

```c
void DrawElementsOneInstance( enum mode, sizei count,
    enum type, const void *indices, int instance,
    uint baseinstance);
```

does not exist in the GL, but is used to describe functionality in the rest of this section. This command constructs a sequence of geometric primitives by successively transferring the count elements whose indices are stored in the currently bound element array buffer (see section 2.9.7) at the offset defined by indices to the GL. The ith element transferred by **DrawElementsOneInstance** will be taken from element indices[i] of each enabled non-instanced array. type must be one of UNSIGNED_BYTE, UNSIGNED_SHORT, or UNSIGNED_INT, indicating that the index values are of GL type ubyte, ushort, or uint respectively. mode specifies what kind of primitives are constructed, as defined in section 2.6.1.

The value of instance may be read by a vertex shader as gl_InstanceID, as described in section 2.11.12.

If no vertex array object is bound (see section 2.10), an INVALID_OPERATION error is generated.

If an enabled vertex attribute array is instanced (it has a non-zero attribute divisor as specified by **VertexAttribDivisor**), the element that is transferred to the GL is given by:
\[
\left\lfloor \frac{\text{instance}}{\text{divisor}} \right\rfloor + \text{baseinstance}
\]

If an array corresponding to a generic attribute required by a vertex shader is not enabled, then the corresponding element is taken from the current generic attribute state (see section 2.7). Otherwise, if an array is enabled, the corresponding current generic attribute value is unaffected by the execution of \texttt{DrawElementsOneInstance}.

The command

\begin{verbatim}
void DrawElements( enum mode, sizei count, enum type, 
    const void *indices );
\end{verbatim}

behaves identically to \texttt{DrawElementsOneInstance} with the \texttt{instance} and \texttt{baseinstance} parameters set to zero; the effect of calling

\begin{verbatim}
DrawElements( mode, count, type, indices );
\end{verbatim}

is equivalent to the command sequence:

\begin{verbatim}
if ( mode, count or type is invalid)
    generate appropriate error
else
    DrawElementsOneInstance( mode, count, type, indices, 0, 0);
\end{verbatim}

The command

\begin{verbatim}
void DrawElementsInstancedBaseInstance( enum mode, 
    sizei count, enum type, const void *indices, 
    sizei primcount, uint baseinstance );
\end{verbatim}

behaves identically to \texttt{DrawElements} except that \texttt{primcount} instances of the set of elements are executed and the value of \texttt{instance} advances between each set. Instanced attributes are advanced as they do during execution of \texttt{DrawArraysInstancedBaseInstance}, and \texttt{baseinstance} has the same effect. It has the same effect as:

\begin{verbatim}
if ( mode, count, type, or primcount is invalid)
    generate appropriate error
else {
    for (int i = 0; i < primcount; i++) {
        ...
    }
\end{verbatim}
DrawElementsOneInstance (mode, count, type, indices, i, baseinstance);

The command

void MultiDrawElements( enum mode, constsizei *count, enum type, const void* const *indices, sizei primcount);

behaves identically to DrawElementsInstanced except that primcount separate sets of elements are specified instead, all elements are treated as though they are not instanced, and the value of instance remains zero. It has the same effect as:

if (mode, count, primcount, or type is invalid)
generate appropriate error
else {
  for (int i = 0; i < primcount; i++)
    DrawElementsOneInstance (mode, count[i], type, indices[i], 0, 0);
}

The command

void DrawRangeElements( enum mode, uint start,
  uint end, sizei count, enum type, const void *indices );

is a restricted form of DrawElements. mode, count, type, and indices match the corresponding arguments to DrawElements, with the additional constraint that all index values identified by indices must lie between start and end inclusive.

Implementations denote recommended maximum amounts of vertex and index data, which may be queried by calling GetIntegerv with the symbolic constants MAX_ELEMENTS_VERTICES and MAX_ELEMENTS_INDICES. If end − start + 1 is greater than the value of MAX_ELEMENTS_VERTICES, or if count is greater than the value of MAX_ELEMENTS_INDICES, then the call may operate at reduced performance. There is no requirement that all vertices in the range [start, end] be referenced. However, the implementation may partially process unused vertices, reducing performance from what could be achieved with an optimal index set.
The error INVALID_VALUE is generated if end < start. Invalid mode, count, or type parameters generate the same errors as would the corresponding call to `DrawElements`. It is an error for index values (other than the primitive restart index, when primitive restart is enabled) to lie outside the range \([start, end]\), but implementations are not required to check for this. Such indices will cause implementation-dependent behavior.

The commands

```c
void DrawElementsBaseVertex( enum mode, sizei count, enum type, const void *indices, int basevertex );
void DrawRangeElementsBaseVertex( enum mode, uint start, uint end, sizei count, enum type, const void *indices, int basevertex );
void DrawElementsInstancedBaseVertex( enum mode, sizei count, enum type, const void *indices, sizei primcount, int basevertex );
void DrawElementsInstancedBaseVertexBaseInstance( enum mode, sizei count, enum type, const void *indices, sizei primcount, int basevertex, uint baseinstance );
```

are equivalent to the commands with the same base name (without the `BaseVertex` suffix), except that the \(i\)th element transferred by the corresponding draw call will be taken from element \(indices[i] + basevertex\) of each enabled array. If the resulting value is larger than the maximum value representable by type, it should behave as if the calculation were upconverted to 32-bit unsigned integers (with wrapping on overflow conditions). The operation is undefined if the sum would be negative and should be handled as described in section 2.9.4. For `DrawRangeElementsBaseVertex`, the index values must lie between \(start\) and \(end\) inclusive, prior to adding the `basevertex` offset. Index values lying outside the range \([start, end]\) are treated in the same way as `DrawRangeElements`.

For `DrawElementsInstancedBaseVertexBaseInstance`, `baseinstance` is used to offset the element from which instanced vertex attributes (those with a non-zero divisor as specified by `VertexAttribDivisor`) are taken.

The command

```c
void DrawElementsIndirect( enum mode, enum type, const void *indirect );
```

has the same effect as:
2.8. VERTEX ARRAYS

typedef struct {
  uint count;
  uint primCount;
  uint firstIndex;
  int baseVertex;
  uint baseInstance;
} DrawElementsIndirectCommand;

if (no element array buffer is bound) {
  generate appropriate error
} else {
  DrawElementsIndirectCommand *cmd =
  (DrawElementsIndirectCommand *)indirect;

  DrawElementsInstancedBaseVertexBaseInstance (mode,
  cmd->count, type,
  cmd->firstIndex * size-of-type,
  cmd->primCount, cmd->baseVertex,
  cmd->baseInstance);
}

An INVALID_OPERATION error is generated if zero is bound to DRAW_-
INDIRECT_BUFFER, or if no element array buffer is bound.
All elements of DrawElementsIndirectCommand are tightly packed.
The command

void MultiDrawElementsBaseVertex( enum mode, const
  sizei *count, enum type, const void *const *indices,
  sizei primcount, const int *basevertex);

behaves identically to DrawElementsBaseVertex, except that primcount separate
lists of elements are specified instead. It has the same effect as:

if (mode or primcount is invalid)
generate appropriate error
else {
  for (int i = 0; i < primcount; i++)
    if (count[i] > 0)
      DrawElementsBaseVertex (mode, count[i], type,
                              indices[i], basevertex[i]);
If the number of supported generic vertex attributes (the value of `MAX_VERTEX_ATTRIBS`) is \( n \), then the state required to implement vertex arrays consists of \( n \) boolean values, \( n \) memory pointers, \( n \) integer stride values, \( n \) symbolic constants representing array types, \( n \) integers representing values per element, \( n \) boolean values indicating normalization, \( n \) boolean values indicating whether the attribute values are pure integers, \( n \) integers representing vertex attribute divisors, and an unsigned integer representing the restart index.

In the initial state, the boolean values are each false, the memory pointers are each `NULL`, the strides are each zero, the array types are each `FLOAT`, the integers representing values per element are each four, the normalized and pure integer flags are each false, the divisors are each zero, and the restart index is zero.

## 2.9 Buffer Objects

The GL uses many types of data supplied by the client. Some of this data must be stored in server memory, and it is usually desirable to store other types of frequently used client data, such as vertex array and pixel data, in server memory even if the option to store it in client memory exists. **Buffer objects** provide a mechanism to allocate, initialize, and render from such memory.

The command

```c
void GenBuffers(sizei n, uint *buffers);
```

returns \( n \) previously unused buffer object names in `buffers`. These names are marked as used, for the purposes of `GenBuffers` only, but they acquire buffer state only when they are first bound with `BindBuffer` (see below), just as if they were unused.

Buffer objects are deleted by calling

```c
void DeleteBuffers(sizei n, const uint *buffers);
```

`buffers` contains \( n \) names of buffer objects to be deleted. After a buffer object is deleted it has no contents, and its name is again unused. Unused names in `buffers` that have been marked as used for the purposes of `GenBuffers` are marked as unused again. Unused names in `buffers` are silently ignored, as is the value zero.
### 2.9. BUFFER OBJECTS

<table>
<thead>
<tr>
<th>Target name</th>
<th>Purpose</th>
<th>Described in section(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY_BUFFER</td>
<td>Vertex attributes</td>
<td>2.9.6</td>
</tr>
<tr>
<td>ATOMIC_COUNTER_BUFFER</td>
<td>Atomic counter storage</td>
<td>2.11.7</td>
</tr>
<tr>
<td>COPY_READ_BUFFER_BINDING</td>
<td>Buffer copy source</td>
<td>2.9.5</td>
</tr>
<tr>
<td>COPY_WRITE_BUFFER_BINDING</td>
<td>Buffer copy destination</td>
<td>2.9.5</td>
</tr>
<tr>
<td>DRAW_INDIRECT_BUFFER</td>
<td>Indirect command arguments</td>
<td>2.9.8</td>
</tr>
<tr>
<td>ELEMENT_ARRAY_BUFFER</td>
<td>Vertex array indices</td>
<td>2.9.7</td>
</tr>
<tr>
<td>PIXEL_PACK_BUFFER</td>
<td>Pixel read target</td>
<td>4.3.1, 6.1</td>
</tr>
<tr>
<td>PIXEL_UNPACK_BUFFER</td>
<td>Texture data source</td>
<td>3.7</td>
</tr>
<tr>
<td>TEXTURE_BUFFER</td>
<td>Texture data buffer</td>
<td>3.9.7</td>
</tr>
<tr>
<td>TRANSFORM_FEEDBACK_BUFFER</td>
<td>Transform feedback buffer</td>
<td>2.17</td>
</tr>
<tr>
<td>UNIFORM_BUFFER</td>
<td>Uniform block storage</td>
<td>2.11.7</td>
</tr>
</tbody>
</table>

Table 2.8: Buffer object binding targets.

### 2.9.1 Creating and Binding Buffer Objects

A buffer object is created by binding a name returned by `GenBuffers` to a buffer target. The binding is effected by calling

```c
void BindBuffer( enum target, uint buffer );
```

`target` must be one of the targets listed in table 2.8. If the buffer object named `buffer` has not been previously bound, the GL creates a new state vector, initialized with a zero-sized memory buffer and comprising all the state and with the same initial values listed in table 2.9.

Buffer objects created by binding a name returned by `GenBuffers` to any of the valid `targets` are formally equivalent, but the GL may make different choices about storage location and layout based on the initial binding.

`BindBuffer` may also be used to bind an existing buffer object. If the bind is successful no change is made to the state of the newly bound buffer object, and any previous binding to `target` is broken.

`BindBuffer` fails and an `INVALID_OPERATION` error is generated if `buffer` is not zero or a name returned from a previous call to `GenBuffers`, or if such a name has since been deleted with `DeleteBuffers`.

While a buffer object is bound, GL operations on the target to which it is bound affect the bound buffer object, and queries of the target to which a buffer object is bound return state from the bound object. Operations on the target also affect any other bindings of that object.
2.9. BUFFER OBJECTS

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Initial Value</th>
<th>Legal Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFFER_SIZE</td>
<td>int64</td>
<td>0</td>
<td>any non-negative integer</td>
</tr>
<tr>
<td>BUFFER_USAGE</td>
<td>enum</td>
<td>STATIC_DRAW</td>
<td>STREAM_DRAW, STREAM_READ, STREAM_COPY, STATIC_DRAW, STATIC_READ, STATIC_COPY, DYNAMIC_DRAW, DYNAMIC_READ, DYNAMIC_COPY</td>
</tr>
<tr>
<td>BUFFER_ACCESS</td>
<td>enum</td>
<td>READ_WRITE</td>
<td>READ_ONLY, WRITE_ONLY, READ_WRITE</td>
</tr>
<tr>
<td>BUFFER_ACCESS_FLAGS</td>
<td>int</td>
<td>0</td>
<td>See section 2.9.3</td>
</tr>
<tr>
<td>BUFFER_MAPPED</td>
<td>boolean</td>
<td>FALSE</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>BUFFER_MAP_POINTER</td>
<td>void*</td>
<td>NULL</td>
<td>address</td>
</tr>
<tr>
<td>BUFFER_MAP_OFFSET</td>
<td>int64</td>
<td>0</td>
<td>any non-negative integer</td>
</tr>
<tr>
<td>BUFFER_MAP_LENGTH</td>
<td>int64</td>
<td>0</td>
<td>any non-negative integer</td>
</tr>
</tbody>
</table>

Table 2.9: Buffer object parameters and their values.

If a buffer object is deleted while it is bound, all bindings to that object in the current context (i.e. in the thread that called `DeleteBuffers`) are reset to zero. Bindings to that buffer in other contexts are not affected, and the deleted buffer may continue to be used at any places it remains bound or attached, as described in appendix D.1.

Initially, each buffer object target is bound to zero. There is no buffer object corresponding to the name zero, so client attempts to modify or query buffer object state for a target bound to zero generate an INVALID_OPERATION error.

**Binding Buffer Objects to Indexed Targets**

Buffer objects may be bound to indexed targets by calling one of the commands

```c
void BindBufferRange( enum target, uint index, uint buffer, intptr offset, sizeiptr size );
void BindBufferBase( enum target, uint index, uint buffer );
```

`target` must be one of ATOMIC_COUNTER_BUFFER, TRANSFORM_FEEDBACK_BUFFER or UNIFORM_BUFFER. Additional language specific to each target is included in sections referred to for each target in table 2.8.

Each `target` represents an indexed array of buffer object binding points, as well as a single general binding point that can be used by other buffer object manipulation functions (e.g. `BindBuffer`, `MapBuffer`). Both commands bind the buffer
object named by buffer to both the general binding point, and to the binding point in the array given by index. If the binds are successful no change is made to the state of the bound buffer object, and any previous bindings to the general binding point or to the binding point in the array are broken.

The error INVALID_VALUE is generated if index is greater than or equal to the number of target-specific indexed binding points.

For BindBufferRange, offset specifies a starting offset into the buffer object buffer, and size specifies the amount of data that can be read from the buffer object while used as an indexed target. Both offset and size are in basic machine units. The error INVALID_VALUE is generated if size is less than or equal to zero. Additional errors may be generated if offset violates target-specific alignment requirements.

BindBufferBase binds the entire buffer, even when the size of the buffer is changed after the binding is established. It is equivalent to calling BindBufferRange with offset zero, while size is determined by the size of the bound buffer at the time the binding is used.

Regardless of the size specified with BindBufferRange, or indirectly with BindBufferBase, the GL will never read or write beyond the end of a bound buffer. In some cases this constraint may result in visibly different behavior when a buffer overflow would otherwise result, such as described for transform feedback operations in section 2.17.2.

2.9.2 Creating Buffer Object Data Stores

The data store of a buffer object is created and initialized by calling

```c
void BufferData( enum target, sizeiptr size, const void *data, enum usage );
```

with target set to one of the targets listed in table 2.8, size set to the size of the data store in basic machine units, and data pointing to the source data in client memory. If data is non-NULL, then the source data is copied to the buffer object’s data store. If data is NULL, then the contents of the buffer object’s data store are undefined.

usage is specified as one of nine enumerated values, indicating the expected application usage pattern of the data store. The values are:

STREAM_DRAW The data store contents will be specified once by the application, and used at most a few times as the source for GL drawing and image specification commands.

STREAM_READ The data store contents will be specified once by reading data from the GL, and queried at most a few times by the application.
STREAM_COPY  The data store contents will be specified once by reading data from the GL, and used at most a few times as the source for GL drawing and image specification commands.

STATIC_DRAW  The data store contents will be specified once by the application, and used many times as the source for GL drawing and image specification commands.

STATIC_READ  The data store contents will be specified once by reading data from the GL, and queried many times by the application.

STATIC_COPY  The data store contents will be specified once by reading data from the GL, and used many times as the source for GL drawing and image specification commands.

DYNAMIC_DRAW  The data store contents will be respecified repeatedly by the application, and used many times as the source for GL drawing and image specification commands.

DYNAMIC_READ  The data store contents will be respecified repeatedly by reading data from the GL, and queried many times by the application.

DYNAMIC_COPY  The data store contents will be respecified repeatedly by reading data from the GL, and used many times as the source for GL drawing and image specification commands.

*usage* is provided as a performance hint only. The specified usage value does not constrain the actual usage pattern of the data store.

BufferData deletes any existing data store, and sets the values of the buffer object’s state variables as shown in table 2.10.

Clients must align data elements consistent with the requirements of the client platform, with an additional base-level requirement that an offset within a buffer to a datum comprising $N$ basic machine units be a multiple of $N$.

If the GL is unable to create a data store of the requested size, the error *OUT_OF_MEMORY* is generated.

To modify some or all of the data contained in a buffer object’s data store, the client may use the command

```c
void BufferSubData( enum target, intptr offset,
                    sizeiptr size, const void *data );
```

OpenGL 4.2 (Core Profile) - April 27, 2012
with target set to one of the targets listed in table 2.8. offset and size indicate the range of data in the buffer object that is to be replaced, in terms of basic machine units. data specifies a region of client memory size basic machine units in length, containing the data that replace the specified buffer range. An INVALID_VALUE error is generated if offset or size is less than zero or if offset + size is greater than the value of BUFFER_SIZE. An INVALID_OPERATION error is generated if any part of the specified buffer range is mapped with MapBufferRange or MapBuffer (see section 2.9.3).

### 2.9.3 Mapping and Unmapping Buffer Data

All or part of the data store of a buffer object may be mapped into the client’s address space by calling

```c
void *MapBufferRange( enum target, intptr offset, sizeiptr length, bitfield access );
```

with target set to one of the targets listed in table 2.8. offset and length indicate the range of data in the buffer object that is to be mapped, in terms of basic machine units. access is a bitfield containing flags which describe the requested mapping. These flags are described below.

If no error occurs, a pointer to the beginning of the mapped range is returned once all pending operations on that buffer have completed, and may be used to modify and/or query the corresponding range of the buffer, according to the following flag bits set in access:

- MAP_READ_BIT indicates that the returned pointer may be used to read buffer object data. No GL error is generated if the pointer is used to query

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFFER_SIZE</td>
<td>size</td>
</tr>
<tr>
<td>BUFFER_USAGE</td>
<td>usage</td>
</tr>
<tr>
<td>BUFFER_ACCESS</td>
<td>READ_WRITE</td>
</tr>
<tr>
<td>BUFFER_ACCESS_FLAGS</td>
<td>0</td>
</tr>
<tr>
<td>BUFFER_MAPPED</td>
<td>FALSE</td>
</tr>
<tr>
<td>BUFFER_MAP_POINTER</td>
<td>NULL</td>
</tr>
<tr>
<td>BUFFER_MAP_OFFSET</td>
<td>0</td>
</tr>
<tr>
<td>BUFFER_MAP_LENGTH</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.10: Buffer object initial state.
a mapping which excludes this flag, but the result is undefined and system errors (possibly including program termination) may occur.

- **MAP_WRITE_BIT** indicates that the returned pointer may be used to modify buffer object data. No GL error is generated if the pointer is used to modify a mapping which excludes this flag, but the result is undefined and system errors (possibly including program termination) may occur.

If no error occurs, the pointer value returned by `MapBufferRange` must reflect an allocation aligned to the value of `MIN_MAP_BUFFER_ALIGNMENT` basic machine units. Subtracting `offset` basic machine units from the returned pointer will always produce a multiple of the value of `MIN_MAP_BUFFER_ALIGNMENT`.

Pointer values returned by `MapBufferRange` may not be passed as parameter values to GL commands. For example, they may not be used to specify array pointers, or to specify or query pixel or texture image data; such actions produce undefined results, although implementations may not check for such behavior for performance reasons.

Mappings to the data stores of buffer objects may have nonstandard performance characteristics. For example, such mappings may be marked as uncacheable regions of memory, and in such cases reading from them may be very slow. To ensure optimal performance, the client should use the mapping in a fashion consistent with the values of `BUFFER_USAGE` and `access`. Using a mapping in a fashion inconsistent with these values is liable to be multiple orders of magnitude slower than using normal memory.

The following optional flag bits in `access` may be used to modify the mapping:

- **MAP_INVALIDATE_RANGE_BIT** indicates that the previous contents of the specified range may be discarded. Data within this range are undefined with the exception of subsequently written data. No GL error is generated if subsequent GL operations access unwritten data, but the result is undefined and system errors (possibly including program termination) may occur. This flag may not be used in combination with `MAP_READ_BIT`.

- **MAP_INVALIDATE_BUFFER_BIT** indicates that the previous contents of the entire buffer may be discarded. Data within the entire buffer are undefined with the exception of subsequently written data. No GL error is generated if subsequent GL operations access unwritten data, but the result is undefined and system errors (possibly including program termination) may occur. This flag may not be used in combination with `MAP_READ_BIT`.

- **MAP_FLUSH_EXPLICIT_BIT** indicates that one or more discrete subranges of the mapping may be modified. When this flag is set, modifications to
2.9. BUFFER OBJECTS

Table 2.11: Buffer object state set by `MapBufferRange`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFFER_ACCESS</td>
<td>Depends on <code>access</code>¹</td>
</tr>
<tr>
<td>BUFFER_ACCESS_FLAGS</td>
<td><code>access</code></td>
</tr>
<tr>
<td>BUFFER_MAPPED</td>
<td>TRUE</td>
</tr>
<tr>
<td>BUFFER_MAP_POINTER</td>
<td>pointer to the data store</td>
</tr>
<tr>
<td>BUFFER_MAP_OFFSET</td>
<td><code>offset</code></td>
</tr>
<tr>
<td>BUFFER_MAP_LENGTH</td>
<td><code>length</code></td>
</tr>
</tbody>
</table>

¹ `BUFFER_ACCESS` is set to `READ_ONLY`, `WRITE_ONLY`, or `READ_WRITE` if `access` & (MAP_READ_BIT|MAP_WRITE_BIT) is respectively MAP_READ_BIT, MAP_WRITE_BIT, or MAP_READ_BIT|MAP_WRITE_BIT.

When this option is selected, flushing is strictly limited to regions that are explicitly indicated with calls to `FlushMappedBufferRange` prior to unmap; if this option is not selected `UnmapBuffer` will automatically flush the entire mapped range when called.

- `MAP_UNSYNCHRONIZED_BIT` indicates that the GL should not attempt to synchronize pending operations on the buffer prior to returning from `MapBufferRange`. No GL error is generated if pending operations which source or modify the buffer overlap the mapped region, but the result of such previous and any subsequent operations is undefined.

A successful `MapBufferRange` sets buffer object state values as shown in table 2.11.

Errors

If an error occurs, `MapBufferRange` returns a NULL pointer.

An `INVALID_VALUE` error is generated if `offset` or `length` is negative, if `offset` + `length` is greater than the value of `BUFFER_SIZE`, or if `access` has any bits set other than those defined above.

An `INVALID_OPERATION` error is generated for any of the following conditions:

- `length` is zero.
2.9. BUFFER OBJECTS

- The buffer is already in a mapped state.
- Neither MAP_READ_BIT nor MAP_WRITE_BIT is set.
- MAP_READ_BIT is set and any of MAP_INVALIDATE_RANGE_BIT, MAP_INVALIDATE_BUFFER_BIT, or MAP_UNSYNCHRONIZED_BIT is set.
- MAP_FLUSH_EXPLICIT_BIT is set and MAP_WRITE_BIT is not set.

An OUT_OF_MEMORY error is generated if MapBufferRange fails because memory for the mapping could not be obtained.

No error is generated if memory outside the mapped range is modified or queried, but the result is undefined and system errors (possibly including program termination) may occur.

The entire data store of a buffer object can be mapped into the client’s address space by calling

```c
void *MapBuffer( enum target, enum access );
```

MapBuffer is equivalent to calling MapBufferRange with the same target, offset of zero, length equal to the value of BUFFER_SIZE, and the access bitfield value passed to MapBufferRange equal to

- MAP_READ_BIT, if mbaccess is READ_ONLY
- MAP_WRITE_BIT, if mbaccess is WRITE_ONLY
- MAP_READ_BIT|MAP_WRITE_BIT, if mbaccess is READ_WRITE

where mbaccess is the value of the access enum parameter passed to MapBuffer.

The pointer value returned by MapBuffer must be aligned to the value of MIN_MAP_BUFFER_ALIGNMENT basic machine units.

INVALID_ENUM is generated if access is not one of the values described above. Other errors are generated as described above for MapBufferRange.

If a buffer is mapped with the MAP_FLUSH_EXPLICIT_BIT flag, modifications to the mapped range may be indicated by calling

```c
void FlushMappedBufferRange( enum target, intptr offset, sizeiptr length );
```

with target set to one of the targets listed in table 2.8. offset and length indicate a modified subrange of the mapping, in basic machine units. The specified subrange to flush is relative to the start of the currently mapped range of buffer.

OpenGL 4.2 (Core Profile) - April 27, 2012
FlushMappedBufferRange may be called multiple times to indicate distinct sub-ranges of the mapping which require flushing.

**Errors**

An **INVALID_VALUE** error is generated if offset or length is negative, or if offset + length exceeds the size of the mapping.

An **INVALID_OPERATION** error is generated if zero is bound to target.

An **INVALID_OPERATION** error is generated if the buffer bound to target is not mapped, or is mapped without the MAP_FLUSH_EXPLICIT_BIT flag.

**Unmapping Buffers**

After the client has specified the contents of a mapped buffer range, and before the data in that range are dereferenced by any GL commands, the mapping must be relinquished by calling

```c
boolean UnmapBuffer( enum target );
```

with target set to one of the targets listed in table 2.8. Unmapping a mapped buffer object invalidates the pointer to its data store and sets the object’s BUFFER_MAPPED, BUFFER_MAP_POINTER, BUFFER_ACCESS_FLAGS, BUFFER_MAP_OFFSET, and BUFFER_MAP_LENGTH state variables to the initial values shown in table 2.10.

**UnmapBuffer** returns TRUE unless data values in the buffer’s data store have become corrupted during the period that the buffer was mapped. Such corruption can be the result of a screen resolution change or other window system-dependent event that causes system heaps such as those for high-performance graphics memory to be discarded. GL implementations must guarantee that such corruption can occur only during the periods that a buffer’s data store is mapped. If such corruption has occurred, **UnmapBuffer** returns FALSE, and the contents of the buffer’s data store become undefined.

If the buffer data store is already in the unmapped state, **UnmapBuffer** returns FALSE, and an **INVALID_OPERATION** error is generated. However, unmapping that occurs as a side effect of buffer deletion or reinitialization is not an error.

**Effects of Mapping Buffers on Other GL Commands**

Most, but not all GL commands will detect attempts to read data from a mapped buffer object. When such an attempt is detected, an **INVALID_OPERATION** error will be generated. Any command which does not detect these attempts, and performs such an invalid read, has undefined results and may result in GL interruption or termination.
2.9. BUFFER OBJECTS

2.9.4 Effects of Accessing Outside Buffer Bounds

Most, but not all GL commands operating on buffer objects will detect attempts to read from or write to a location in a bound buffer object at an offset less than zero, or greater than or equal to the buffer’s size. When such an attempt is detected, a GL error will be generated. Any command which does not detect these attempts, and performs such an invalid read or write, has undefined results, and may result in GL interruption or termination.

2.9.5 Copying Between Buffers

All or part of the data store of a buffer object may be copied to the data store of another buffer object by calling

```c
void CopyBufferSubData( enum readtarget, enum writetarget,
    intptr readoffset, intptr writeoffset, sizeiptr size );
```

with `readtarget` and `writetarget` each set to one of the targets listed in table 2.8. While any of these targets may be used, the `COPY_READ_BUFFER_BINDING` and `COPY_WRITE_BUFFER_BINDING` targets are provided specifically for copies, so that they can be done without affecting other buffer binding targets that may be in use. `writeoffset` and `size` specify the range of data in the buffer object bound to `writetarget` that is to be replaced, in terms of basic machine units. `readoffset` and `size` specify the range of data in the buffer object bound to `readtarget` that is to be copied to the corresponding region of `writetarget`.

An `INVALID_VALUE` error is generated if any of `readoffset`, `writeoffset`, or `size` are negative, if `readoffset + size` exceeds the size of the buffer object bound to `readtarget`, or if `writeoffset + size` exceeds the size of the buffer object bound to `writetarget`.

An `INVALID_VALUE` error is generated if the same buffer object is bound to both `readtarget` and `writetarget`, and the ranges `[readoffset, readoffset + size]` and `[writeoffset, writeoffset + size]` overlap.

An `INVALID_OPERATION` error is generated if zero is bound to `readtarget` or `writetarget`.

An `INVALID_OPERATION` error is generated if the buffer objects bound to either `readtarget` or `writetarget` are mapped.

2.9.6 Vertex Arrays in Buffer Objects

Blocks of vertex array data are stored in buffer objects with the same format and layout options described in section 2.8. A buffer object binding point is added to
the client state associated with each vertex array type. The commands that specify
the locations and organizations of vertex arrays copy the buffer object name that
is bound to ARRAY_BUFFER to the binding point corresponding to the vertex array of
the type being specified. For example, the VertexAttribPointer command copies
the value of ARRAY_BUFFER_BINDING (the queriable name of the buffer binding
corresponding to the target ARRAY_BUFFER) to the client state variable VERTEX_-
ATTRIB_ARRAY_BUFFER_BINDING for the specified index.

Rendering commands DrawArrays, and the other drawing commands defined
in section 2.8.3 operate as previously defined, where data for enabled generic at-
ttribute arrays are sourced from buffer objects. When an array is sourced from a
buffer object, the pointer value of that array is used to compute an offset, in basic
machine units, into the data store of the buffer object. This offset is computed by
subtracting a NULL pointer from the pointer value, where both pointers are treated
as pointers to basic machine units.

If any enabled array’s buffer binding is zero when DrawArrays or one of the
other drawing commands defined in section 2.8.3 is called, the result is undefined.

2.9.7 Array Indices in Buffer Objects

Blocks of array indices are stored in buffer objects in the formats described by the
type parameter of DrawElements (see section 2.8.3).

A buffer object is bound to ELEMENT_ARRAY_BUFFER by calling BindBuffer
with target set to ELEMENT_ARRAY_BUFFER, and buffer set to the name of the
buffer object. If no corresponding buffer object exists, one is initialized as defined
in section 2.9.

DrawElements, DrawRangeElements, and DrawElementsInstanced source
their indices from the buffer object whose name is bound to ELEMENT_-ARRAY_BUFFER, using their indices parameters as offsets into the buffer object
in the same fashion as described in section 2.9.6. DrawElementsBaseVertex, DrawRangeElementsBaseVertex, and DrawElementsInstancedBaseVer-
tex also source their indices from that buffer object, adding the basevertex offset to
the appropriate vertex index as a final step before indexing into the vertex buffer;
this does not affect the calculation of the base pointer for the index array. Finally,
MultiDrawElements and MultiDrawElementsBaseVertex also source their in-
dices from that buffer object, using its indices parameter as a pointer to an
array of pointers that represent offsets into the buffer object. If zero is bound to
ELEMENT_ARRAY_BUFFER, the result of these drawing commands is undefined.

In some cases performance will be optimized by storing indices and array data
in separate buffer objects, and by creating those buffer objects with the correspond-
ing binding points.
2.10. VERTEX ARRAY OBJECTS

2.9.8 Indirect Commands in Buffer Objects

Arguments to **DrawArraysIndirect** and **DrawElementsIndirect** commands are stored in buffer objects in the formats described in section 2.8.3 for the **DrawArraysIndirectCommand** and **DrawElementsIndirectCommand** structures, respectively.

A buffer object is bound to **DRAW_INDIRECT_BUFFER** by calling **BindBuffer** with **target** set to **DRAW_INDIRECT_BUFFER**, and **buffer** set to the name of the buffer object. If no corresponding buffer object exists, one is initialized as defined in section 2.9.

**DrawArraysIndirect** and **DrawElementsIndirect** source their arguments from the buffer object whose name is bound to **DRAW_INDIRECT_BUFFER**, using their **indirect** parameters as offsets into the buffer object in the same fashion as described in section 2.9.6. An **INVALID_OPERATION** error is generated if these commands source data beyond the end of the buffer object, if zero is bound to **DRAW_INDIRECT_BUFFER**, or if **indirect** is not aligned to a multiple of the size, in basic machine units, of uint.

2.9.9 Buffer Object State

The state required to support buffer objects consists of binding names for each of the buffer targets in table 2.8, and for each of the indexed buffer targets in section 2.9.1. Additionally, each vertex array has an associated binding so there is a buffer object binding for each of the vertex attribute arrays. The initial values for all buffer object bindings is zero.

The state of each buffer object consists of a buffer size in basic machine units, a usage parameter, an access parameter, a mapped boolean, two integers for the offset and size of the mapped region, a pointer to the mapped buffer (**NULL** if unmapped), and the sized array of basic machine units for the buffer data.

2.10 Vertex Array Objects

The buffer objects that are to be used by the vertex stage of the GL are collected together to form a vertex array object. All state related to the definition of data used by the vertex processor is encapsulated in a vertex array object.

The name space for vertex array objects is the unsigned integers, with zero reserved by the GL.

The command

```c
void GenVertexArrays( sizei n, uint *arrays );
```

OpenGL 4.2 (Core Profile) - April 27, 2012
returns \( n \) previous unused vertex array object names in \( \text{arrays} \). These names are marked as used, for the purposes of \text{GenVertexArrays} only, but they acquire array state only when they are first bound.

Vertex array objects are deleted by calling

$$\text{void DeleteVertexArrays}(\text{sizei } n, \text{const uint } *\text{arrays});$$

\( \text{arrays} \) contains \( n \) names of vertex array objects to be deleted. Once a vertex array object is deleted it has no contents and its name is again unused. If a vertex array object that is currently bound is deleted, the binding for that object reverts to zero and no vertex array object is bound. Unused names in \( \text{arrays} \) that have been marked as used for the purposes of \text{GenVertexArrays} are marked as unused again. Unused names in \( \text{arrays} \) are silently ignored, as is the value zero.

A vertex array object is created by binding a name returned by \text{GenVertexArrays} with the command

$$\text{void BindVertexArray}(\text{uint } \text{array});$$

\( \text{array} \) is the vertex array object name. The resulting vertex array object is a new state vector, comprising all the state and with the same initial values listed in tables 6.5 and 6.6.

\text{BindVertexArray} may also be used to bind an existing vertex array object. If the bind is successful no change is made to the state of the bound vertex array object, and any previous binding is broken.

The currently bound vertex array object is used for all commands which modify vertex array state, such as \text{VertexAttribPointer} and \text{EnableVertexAttribArray}; all commands which draw from vertex arrays, such as \text{DrawArrays} and \text{DrawElements}; and all queries of vertex array state (see chapter 6).

In the initial GL state, or when the current vertex array binding is zero as a result of \text{BindVertexArray} or a side effect of \text{DeleteVertexArrays}, no vertex array object is bound, and commands which modify, draw from, or query vertex array state will fail and generate an \text{INVALID_OPERATION} error.

\text{BindVertexArray} fails and an \text{INVALID_OPERATION} error is generated if \( \text{array} \) is not zero or a name returned from a previous call to \text{GenVertexArrays}, or if such a name has since been deleted with \text{DeleteVertexArrays}.

### 2.11 Vertex Shaders

Vertex shaders describe the operations that occur on vertex values and their associated data.
A vertex shader is an array of strings containing source code for the operations that are meant to occur on each vertex that is processed. The language used for vertex shaders is described in the OpenGL Shading Language Specification.

To use a vertex shader, shader source code is first loaded into a shader object and then compiled. A shader object corresponds to a stage in the rendering pipeline referred to as its shader stage or type. Alternatively, pre-compiled shader binary code may be directly loaded into a shader object. A GL implementation must support shader compilation (the boolean value SHADER_COMPILER must be TRUE). If the integer value of NUM_SHADER_BINARY_FORMATS is greater than zero, then shader binary loading is supported.

One or more vertex shader objects are attached to a program object. The program object is then linked, which generates executable code from all the compiled shader objects attached to the program. Alternatively, pre-compiled program binary code may be directly loaded into a program object (see section 2.11.5).

When program objects are bound to a shader stage, they become the current program object for that stage. When the current program object for the vertex stage includes a vertex shader, it is considered the active program object for the vertex stage. The current program object for all stages may be set at once using a single unified program object, or the current program object may be set for each stage individually using a separable program object where different separable program objects may be current for other stages. The set of separable program objects current for all stages are collected in a program pipeline object that must be bound for use. When a linked program object is made active for the vertex stage, the executable code for the vertex shaders it contains is used to process vertices.

In addition to vertex shaders, tessellation control shaders, tessellation evaluation shaders, geometry shaders and fragment shaders can be created, compiled, and linked into program objects. Tessellation control and evaluation shaders are used to control the operation of the tessellator, and are described in section 2.12. Geometry shaders affect the processing of primitives assembled from vertices (see section 2.13). Fragment shaders affect the processing of fragments during rasterization (see section 3.10). A single program object can contain all of vertex, tessellation control, tessellation evaluation, geometry, and fragment shaders, or any subset thereof.

When the program object currently in use for the vertex stage includes a vertex shader, its vertex shader is considered active and is used to process vertices. If the current vertex stage program object has no vertex shader, or no program object is current for the vertex stage, the results of vertex shader execution are undefined.

A vertex shader can reference a number of variables as it executes. Vertex attributes are the per-vertex values specified in section 2.7. Uniforms are per-program variables that are constant during program execution. Samplers are a spe-
2.11. **VERTEX SHADERS**

A special form of uniform used for texturing (section 3.9). Output variables hold the results of vertex shader execution that are used later in the pipeline. Each of these variable types is described in more detail below.

### 2.11.1 Shader Objects

The source code that makes up a program that gets executed by one of the programmable stages is encapsulated in one or more shader objects.

The name space for shader objects is the unsigned integers, with zero reserved for the GL. This name space is shared with program objects. The following sections define commands that operate on shader and program objects by name. Commands that accept shader or program object names will generate the error INVALID_-

VALUE if the provided name is not the name of either a shader or program object and INVALID_OPERATION if the provided name identifies an object that is not the expected type.

To create a shader object, use the command

```
uint CreateShader( enum type );
```

The shader object is empty when it is created. The type argument specifies the type of shader object to be created. For vertex shaders, type must be VERTEX_SHADER. A non-zero name that can be used to reference the shader object is returned. If an error occurs, zero will be returned.

The command

```
void ShaderSource( uint shader, sizei count, const char * const *string, const int *length );
```

loads source code into the shader object named shader. string is an array of count pointers to optionally null-terminated character strings that make up the source code. The length argument is an array with the number of chars in each string (the string length). If an element in length is negative, its accompanying string is null-terminated. If length is NULL, all strings in the string argument are considered null-terminated. The ShaderSource command sets the source code for the shader to the text strings in the string array. If shader previously had source code loaded into it, the existing source code is completely replaced. Any length passed in excludes the null terminator in its count.

The strings that are loaded into a shader object are expected to form the source code for a valid shader as defined in the OpenGL Shading Language Specification.

Once the source code for a shader has been loaded, a shader object can be compiled with the command

```
OpenGL 4.2 (Core Profile) - April 27, 2012
```
2.11. VERTEX SHADERS

```c
void CompileShader(uint shader);
```

Each shader object has a boolean status, `COMPILATION_STATUS`, that is modified as a result of compilation. This status can be queried with `GetShaderiv` (see section 6.1.12). This status will be set to `TRUE` if `shader` was compiled without errors and is ready for use, and `FALSE` otherwise. Compilation can fail for a variety of reasons as listed in the OpenGL Shading Language Specification. If `CompileShader` failed, any information about a previous compile is lost. Thus a failed compile does not restore the old state of `shader`.

Changing the source code of a shader object with `ShaderSource` does not change its compile status or the compiled shader code.

Each shader object has an information log, which is a text string that is overwritten as a result of compilation. This information log can be queried with `GetShaderInfoLog` to obtain more information about the compilation attempt (see section 6.1.12).

An `INVALID_OPERATION` error is generated if `shader` is not the name of a valid shader object generated by `CreateShader`.

Resources allocated by the shader compiler may be released with the command

```c
void ReleaseShaderCompiler(void);
```

This is a hint from the application, and does not prevent later use of the shader compiler. If shader source is loaded and compiled after `ReleaseShaderCompiler` has been called, `CompileShader` must succeed provided there are no errors in the shader source.

The range and precision for different numeric formats supported by the shader compiler may be determined with the command `GetShaderPrecisionFormat` (see section 6.1.12).

Shader objects can be deleted with the command

```c
void DeleteShader(uint shader);
```

If `shader` is not attached to any program object, it is deleted immediately. Otherwise, `shader` is flagged for deletion and will be deleted when it is no longer attached to any program object. If an object is flagged for deletion, its boolean status bit `DELETE_STATUS` is set to true. The value of `DELETE_STATUS` can be queried with `GetShaderiv` (see section 6.1.12). `DeleteShader` will silently ignore the value zero.
2.11. VERTEX SHADERS

2.11.2 Loading Shader Binaries

Precompiled shader binaries may be loaded with the command

```c
void ShaderBinary( sizei count, const uint *shaders,
    enum binaryformat, const void *binary, sizei length );
```

`shaders` contains a list of `count` shader object handles. Each handle refers to a unique shader type (vertex shader or fragment shader). `binary` points to `length` bytes of pre-compiled binary shader code in client memory, and `binaryformat` denotes the format of the pre-compiled code.

The binary image will be decoded according to the extension specification defining the specified `binaryformat`. OpenGL defines no specific binary formats, but does provide a mechanism to obtain token values for such formats provided by extensions. The number of shader binary formats supported can be obtained by querying the value of `NUM_SHADER_BINARY_FORMATS`. The list of specific binary formats supported can be obtained by querying the value of `SHADER_BINARY_FORMATS`.

Depending on the types of the shader objects in `shaders`, `ShaderBinary` will individually load binary vertex or fragment shaders, or load an executable binary that contains an optimized pair of vertex and fragment shaders stored in the same binary.

An `INVALID_ENUM` error is generated if `binaryformat` is not a supported format returned in `SHADER_BINARY_FORMATS`. An `INVALID_VALUE` error is generated if the data pointed to by `binary` does not match the specified `binaryformat`. Additional errors corresponding to specific binary formats may be generated as specified by the extensions defining those formats. An `INVALID_OPERATION` error is generated if more than one of the handles refers to the same type of shader (vertex or fragment shader).

If `ShaderBinary` fails, the old state of shader objects for which the binary was being loaded will not be restored.

Note that if shader binary interfaces are supported, then a GL implementation may require that an optimized pair of vertex and fragment shader binaries that were compiled together be specified to `LinkProgram`. Not specifying an optimized pair may cause `LinkProgram` to fail.

2.11.3 Program Objects

The shader objects that are to be used by the programmable stages of the GL are collected together to form a program object. The programs that are executed by these programmable stages are called executables. All information necessary for
2.11. VERTEX SHADERS

defining an executable is encapsulated in a program object. A program object is created with the command

```c
uint CreateProgram( void);
```

Program objects are empty when they are created. A non-zero name that can be used to reference the program object is returned. If an error occurs, zero will be returned.

To attach a shader object to a program object, use the command

```c
void AttachShader( uint program, uint shader);
```

The error `INVALID_OPERATION` is generated if `shader` is already attached to `program`.

Shader objects may be attached to program objects before source code has been loaded into the shader object, or before the shader object has been compiled. Multiple shader objects of the same type may be attached to a single program object, and a single shader object may be attached to more than one program object.

To detach a shader object from a program object, use the command

```c
void DetachShader( uint program, uint shader);
```

The error `INVALID_OPERATION` is generated if `shader` is not attached to `program`. If `shader` has been flagged for deletion and is not attached to any other program object, it is deleted.

In order to use the shader objects contained in a program object, the program object must be linked. The command

```c
void LinkProgram( uint program);
```

will link the program object named `program`. Each program object has a boolean status, `LINK_STATUS`, that is modified as a result of linking. This status can be queried with `GetProgramiv` (see section 6.1.12). This status will be set to `TRUE` if a valid executable is created, and `FALSE` otherwise.

Linking can fail for a variety of reasons as specified in the OpenGL Shading Language Specification, as well as any of the following reasons:

- One or more of the shader objects attached to `program` are not compiled successfully.
- More active uniform or active sampler variables are used in `program` than allowed (see sections 2.11.7, 2.11.9, and 2.13.3).
2.11. VERTEX SHADERS

- The program object contains objects to form a tessellation control shader (see section 2.12.1), and
  - the program is not separable and contains no objects to form a vertex shader;
  - the output patch vertex count is not specified in any compiled tessellation control shader object; or
  - the output patch vertex count is specified differently in multiple tessellation control shader objects.

- The program object contains objects to form a tessellation evaluation shader (see section 2.12.3), and
  - the program is not separable and contains no objects to form a vertex shader;
  - the tessellation primitive mode is not specified in any compiled tessellation evaluation shader object; or
  - the tessellation primitive mode, spacing, vertex order, or point mode is specified differently in multiple tessellation evaluation shader objects.

- The program object contains objects to form a geometry shader (see section 2.13), and
  - the program is not separable and contains no objects to form a vertex shader;
  - the input primitive type, output primitive type, or maximum output vertex count is not specified in any compiled geometry shader object; or
  - the input primitive type, output primitive type, or maximum output vertex count is specified differently in multiple geometry shader objects.

If `LinkProgram` failed, any information about a previous link of that program object is lost. Thus, a failed link does not restore the old state of `program`.

When successfully linked program objects are used for rendering operations, they may access GL state and interface with other stages of the GL pipeline through active variables and active interface blocks. The GL provides various commands allowing applications to enumerate and query properties of active variables and interface blocks for a specified program. If one of these commands is called with a program for which `LinkProgram` succeeded, the information recorded when the program was linked is returned. If one of these commands is called with a program...
for which \texttt{LinkProgram} failed, no error is generated unless otherwise noted. Implementations may return information on variables and interface blocks that would have been active had the program been linked successfully. In cases where the link failed because the program required too many resources, these commands may help applications determine why limits were exceeded. However, the information returned in this case is implementation-dependent and may be incomplete. If one of these commands is called with a program for which \texttt{LinkProgram} had never been called, no error will be generated unless otherwise noted, and the program object is considered to have no active variables or interface blocks.

Each program object has an information log that is overwritten as a result of a link operation. This information log can be queried with \texttt{GetProgramInfoLog} to obtain more information about the link operation or the validation information (see section 6.1.12).

If a program has been successfully linked by \texttt{LinkProgram} or \texttt{ProgramBinary} (see section 2.11.5), it can be made part of the current rendering state for all shader stages with the command

\begin{verbatim}
void UseProgram(uint program);
\end{verbatim}

If \texttt{program} is non-zero, this command will make \texttt{program} the current program object. This will install executable code as part of the current rendering state for each shader stage present when the program was last successfully linked. If \texttt{UseProgram} is called with \texttt{program} set to zero, then there is no current program object. If \texttt{program} has not been successfully linked, the error \texttt{INVALID_OPERATION} is generated and the current rendering state is not modified.

While a program object is in use, applications are free to modify attached shader objects, compile attached shader objects, attach additional shader objects, and detach shader objects. These operations do not affect the link status or executable code of the program object.

The executable code for an individual shader stage is taken from the current program for that stage. If there is a current program object established by \texttt{UseProgram}, that program is considered current for all stages. Otherwise, if there is a bound program pipeline object (see section 2.11.4), the program bound to the appropriate stage of the pipeline object is considered current. If there is no current program object or bound program pipeline object, no program is current for any stage. The current program for a stage is considered \textit{active} if it contains executable code for that stage; otherwise, no program is considered active for that stage. If there is no active program for the vertex or fragment shader stages, the results of vertex and/or fragment processing will be undefined. However, this is not an error. If there is no active program for the tessellation control, tessellation evaluation, or geometry shader stages, those stages are ignored.

OpenGL 4.2 (Core Profile) - April 27, 2012
If LinkProgram or ProgramBinary successfully re-links a program object that is active for any shader stage, then the newly generated executable code will be installed as part of the current rendering state for all shader stages where the program is active. Additionally, the newly generated executable code is made part of the state of any program pipeline for all stages where the program is attached.

If a program object that is active for any shader stage is re-linked unsuccessfully, the link status will be set to FALSE, but existing executables and associated state will remain part of the current rendering state until a subsequent call to UseProgram, UseProgramStages, or BindProgramPipeline removes them from use. If such a program is attached to any program pipeline object, the existing executables and associated state will remain part of the program pipeline object until a subsequent call to UseProgramStages removes them from use. An unsuccessfully linked program may not be made part of the current rendering state by UseProgram or added to program pipeline objects by UseProgramStages until it is successfully re-linked. If such a program was attached to a program pipeline at the time of a failed link, its existing executable may still be made part of the current rendering state indirectly by BindProgramPipeline.

To set a program object parameter, call

\[
\text{void ProgramParameteri}(\text{uint program, enum pname, int value})
\]

\(pname\) identifies which parameter to set for \(program\). \(value\) holds the value being set.

If \(pname\) is PROGRAM_SEPARABLE, \(value\) must be TRUE or FALSE, and indicates whether \(program\) can be bound for individual pipeline stages using UseProgramStages after it is next linked. Other legal values for \(pname\) and \(value\) are discussed in section 2.11.5.

Program objects can be deleted with the command

\[
\text{void DeleteProgram( uint program )};
\]

If \(program\) is not current for any GL context, is not the active program for any program pipeline object, and is not the current program for any stage of any program pipeline object, it is deleted immediately. Otherwise, \(program\) is flagged for deletion and will be deleted after all of these conditions become true. When a program object is deleted, all shader objects attached to it are detached. DeleteProgram will silently ignore the value zero.

The command

\[
\text{uint CreateShaderProgramv( enum type, sizei count, const char * const *strings)};
\]

OpenGL 4.2 (Core Profile) - April 27, 2012
creates a stand-alone program from an array of null-terminated source code strings for a single shader type. `CreateShaderProgramv` is equivalent to the following command sequence:

```c
const uint shader = CreateShader(type);
if (shader) {
    ShaderSource(shader, count, strings, NULL);
    CompileShader(shader);
    const uint program = CreateProgram();
    if (program) {
        int compiled = FALSE;
        GetShaderiv(shader, COMPILE_STATUS, &compiled);
        ProgramParameteri(program, PROGRAM_SEPARABLE, TRUE);
        if (compiled) {
            AttachShader(program, shader);
            LinkProgram(program);
            DetachShader(program, shader);
        }
    }
    append-shader-info-log-to-program-info-log
}
DeleteShader(shader);
return program;
} else {
    return 0;
}
```

The program may not actually link if the output variables in the shader attached to the final stage of the linked program take up too many locations. If this situation arises, the info log may explain this.

Because no shader is returned by `CreateShaderProgramv` and the shader that is created is deleted in the course of the command sequence, the info log of the shader object is copied to the program so the shader’s failed info log for the failed compilation is accessible to the application.

### 2.11.4 Program Pipeline Objects

Instead of packaging all shader stages into a single program object, shader types might be contained in multiple program objects each consisting of part of the complete pipeline. A program object may even contain only a single shader stage. This facilitates greater flexibility when combining different shaders in various ways without requiring a program object for each combination.
Program bindings associating program objects with shader types are collected to form a program pipeline object.

The command

```c
void GenProgramPipelines(sizei n, uint *pipelines);
```

returns \( n \) previously unused program pipeline object names in `pipelines`. These names are marked as used, for the purposes of `GenProgramPipelines` only, but they acquire state only when they are first bound.

Program pipeline objects are deleted by calling

```c
void DeleteProgramPipelines(sizei n, const uint *pipelines);
```

`pipelines` contains \( n \) names of program pipeline objects to be deleted. Once a program pipeline object is deleted, it has no contents and its name becomes unused. If an object that is currently bound is deleted, the binding for that object reverts to zero and no program pipeline object becomes current. Unused names in `pipelines` that have been marked as used for the purposes of `GenProgramPipelines` are marked as unused again. Unused names in `pipelines` are silently ignored, as is the value zero.

A program pipeline object is created by binding a name returned by `GenProgramPipelines` with the command

```c
void BindProgramPipeline(uint pipeline);
```

`pipeline` is the program pipeline object name. The resulting program pipeline object is a new state vector, comprising all the state and with the same initial values listed in table 6.31.

`BindProgramPipeline` may also be used to bind an existing program pipeline object. If the bind is successful, no change is made to the state of the bound program pipeline object, and any previous binding is broken. If `BindProgramPipeline` is called with `pipeline` set to zero, then there is no current program pipeline object.

If no current program object has been established by `UseProgram`, the program objects used for each shader stage and for uniform updates are taken from the bound program pipeline object, if any. If there is a current program object established by `UseProgram`, the bound program pipeline object has no effect on rendering or uniform updates. When a bound program pipeline object is used for rendering, individual shader executables are taken from its program objects as described in the discussion of `UseProgram` in section 2.11.3).
2.11. VERTEX SHADERS

BindProgramPipeline fails and an INVALID_OPERATION error is generated if pipeline is not zero or a name returned from a previous call to GenProgramPipelines, or if such a name has since been deleted with DeleteProgramPipelines.

The executables in a program object associated with one or more shader stages can be made part of the program pipeline state for those shader stages with the command:

```c
void UseProgramStages( uint pipeline, bitfield stages, uint program );
```

where pipeline is the program pipeline object to be updated, stages is the bitwise OR of accepted constants representing shader stages, and program is the program object from which the executables are taken. The bits set in stages indicate the program stages for which the program object named by program becomes current. These stages may include tessellation control, tessellation evaluation, vertex, geometry, or fragment indicated by TESS_CONTROL_SHADER_BIT, TESS_EVALUATION_SHADER_BIT, VERTEX_SHADER_BIT, GEOMETRY_SHADER_BIT, or FRAGMENT_SHADER_BIT respectively. The constant ALL_SHADER_BITS indicates program is to be made current for all shader stages.

If program refers to a program object with a valid shader attached for an indicated shader stage, this call installs the executable code for that stage in the indicated program pipeline object state. If UseProgramStages is called with program set to zero or with a program object that contains no executable code for the given stages, it is as if the pipeline object has no programmable stage configured for the indicated shader stages. If stages is not the special value ALL_SHADER_BITS, and has a bit set that is not recognized, the error INVALID_VALUE is generated. If the program object named by program was linked without the PROGRAM_SEPARABLE parameter set, or was not linked successfully, the error INVALID_OPERATION is generated and the corresponding shader stages in the pipeline program pipeline object are not modified.

If pipeline is a name that has been generated (without subsequent deletion) by GenProgramPipelines, but refers to a program pipeline object that has not been previously bound, the GL first creates a new state vector in the same manner as when BindProgramPipeline creates a new program pipeline object. If pipeline is not a name returned from a previous call to GenProgramPipelines or if such a name has since been deleted by DeleteProgramPipelines, an INVALID_OPERATION error is generated.

The command

```c
void ActiveShaderProgram( uint pipeline, uint program );
```
sets the linked program named by `program` to be the active program (discussed later in the section 2.14.4) for the program pipeline object `pipeline`. If `program` has not been successfully linked, the error `INVALID_OPERATION` is generated and active program is not modified.

If `pipeline` is a name that has been generated (without subsequent deletion) by `GenProgramPipelines`, but refers to a program pipeline object that has not been previously bound, the GL first creates a new state vector in the same manner as when `BindProgramPipeline` creates a new program pipeline object. If `pipeline` is not a name returned from a previous call to `GenProgramPipelines` or if such a name has since been deleted by `DeleteProgramPipelines`, an `INVALID_OPERATION` error is generated.

**Shader Interface Matching**

When linking a non-separable program object with multiple shader types, the outputs of one stage form an interface with the inputs of the next stage. These inputs and outputs must typically match in name, type, and qualification. When both sides of an interface are contained in the same program object, `LinkProgram` will detect mismatches on an interface and generate link errors.

When multiple shader stages are active, the outputs of one stage form an interface with the inputs of the next stage. At each such interface, shader inputs are matched up against outputs from the previous stage:

- An output block is considered to match an input block in the subsequent shader if the two blocks have the same block name, and the members of the block match exactly in name, type, qualification, and declaration order.

- An output variable is considered to match an input variable in the subsequent shader if:
  - the two variables match in name, type, and qualification; or
  - the two variables are declared with the same location layout qualifier and match in type and qualification.

Variables or block members declared as structures are considered to match in type if and only if structure members match in name, type, qualification, and declaration order. Variables or block members declared as arrays are considered to match in type only if both declarations specify the same element type and array size. The rules for determining if variables or block members match in qualification are found in the OpenGL Shading Language Specification.
Tessellation control shader per-vertex output variables and blocks and tessellation control, tessellation evaluation, and geometry shader per-vertex input variables and blocks are required to be declared as arrays, with each element representing input or output values for a single vertex of a multi-vertex primitive. For the purposes of interface matching, such variables and blocks are treated as though they were not declared as arrays.

For program objects containing multiple shaders, LinkProgram will check for mismatches on interfaces between shader stages in the program being linked and generate a link error if a mismatch is detected. A link error will be generated if any statically referenced input variable or block does not have a matching output. If either shader redeclares the built-in array gl_ClipDistance[], the array must have the same size in both shaders.

With separable program objects, interfaces between shader stages may involve the outputs from one program object and the inputs from a second program object. For such interfaces, it is not possible to detect mismatches at link time, because the programs are linked separately. When each such program is linked, all inputs or outputs interfacing with another program stage are treated as active. The linker will generate an executable that assumes the presence of a compatible program on the other side of the interface. If a mismatch between programs occurs, no GL error will be generated, but some or all of the inputs on the interface will be undefined.

At an interface between program objects, the set of inputs and outputs are considered to match exactly if and only if:

- Every declared input block or variable must have a matching output, as described above.
- For each user-defined input block declared, there is a matching output block in the previous shader. Two blocks are considered to match if they have the same name, and the members of the block match exactly in name, type, qualification, and declaration order.
- For every user-declared input variable declared, there is an output variable declared in the previous shader matching exactly in name, type, and qualification.
- There are no output blocks or user-defined output variables declared without a matching input block or variable declaration.

When the set of inputs and outputs on an interface between programs matches exactly, all inputs are well-defined unless the corresponding outputs were not written in the previous shader. However, any mismatch between inputs and outputs
results in all inputs being undefined except for cases noted below. Even if an input has a corresponding output that matches exactly, mismatches on other inputs or outputs may adversely affect the executable code generated to read or write the matching variable.

The inputs and outputs on an interface between programs need not match exactly when input and output location qualifiers (sections 4.3.8.1 and 4.3.8.2 of the OpenGL Shading Language Specification) are used. When using location qualifiers, any input with an input location qualifier will be well-defined as long as the other program writes to a matching output, as described above. The names of variables need not match when matching by location.

Additionally, scalar and vector inputs with location layout qualifiers will be well-defined if there is a corresponding output satisfying all of the following conditions:

- the input and output match exactly in qualification, including in the location layout qualifier;
- the output is a vector with the same basic component type and has more components than the input; and
- the common component type of the input and output is `int`, `uint`, or `float` (scalars and vectors with `double` component type are excluded).

In this case, the components of the input will be taken from the first components of the matching output, and the extra components of the output will be ignored.

To use any built-in input or output in the `gl_PerVertex` block in separable program objects, shader code must redeclare that block prior to use. A separable program will fail to link if:

- it contains multiple shaders of a single type with different redeclarations of this built-in block; or
- any shader uses a built-in block member not found in the redeclaration of that block.

There is one exception to this rule described below.

As described above, an exact interface match requires matching built-in input and output blocks. At an interface between two non-fragment shader stages, the `gl_PerVertex` input and output blocks are considered to match if and only if the block members members match exactly in name, type, qualification, and declaration order. At an interface involving the fragment shader stage, the presence or absence of any built-in output does not affect interface matching.
Built-in inputs or outputs not found in blocks do not affect interface matching. Any such built-in inputs are well-defined unless they are derived from built-in outputs not written by the previous shader stage.

**Program Pipeline Object State**

The state required to support program pipeline objects consists of a single binding name of the current program pipeline object. This binding is initially zero indicating no program pipeline object is bound.

The state of each program pipeline object consists of:

- Six unsigned integers (initially all zero) are required to hold each respective name of the current vertex stage program, current geometry stage program, current fragment stage program, current tessellation control stage program, current tessellation evaluation stage program, and active program respectively.

- A boolean holding the status of the last validation attempt, initially false.

- An array of type char containing the information log, initially empty.

- An integer holding the length of the information log.

### 2.11.5 Program Binaries

The command

```c
void GetProgramBinary(uint program, sizei bufSize, sizei *length, enum *binaryFormat, void *binary);
```

returns a binary representation of the program object’s compiled and linked executable source, henceforth referred to as its program binary. The maximum number of bytes that may be written into `binary` is specified by `bufSize`. If `bufSize` is less than the number of bytes in the program binary, then an `INVALID_OPERATION` error is generated. Otherwise, the actual number of bytes written into `binary` is returned in `length` and its format is returned in `binaryFormat`. If `length` is `NULL`, then no length is returned.

The number of bytes in the program binary can be queried by calling `GetProgramiv` with `pname PROGRAM_BINARY_LENGTH`. When a program object’s `LINK_STATUS` is `FALSE`, its program binary length is zero, and a call to `GetProgramBinary` will generate an `INVALID_OPERATION` error.

The command
2.11. VERTEX SHADERS

```c
void ProgramBinary( uint program, enum binaryFormat,
                     const void *binary, sizei length );
```

loads a program object with a program binary previously returned from `GetProgramBinary`. This is useful for future instantiations of the GL to avoid online compilation, while still using OpenGL Shading Language source shaders as a portable initial format. `binaryFormat` and `binary` must be those returned by a previous call to `GetProgramBinary`, and `length` must be the length of the program binary as returned by `GetProgramBinary` or `GetProgramiv` with `pname` `PROGRAM_BINARY_LENGTH`. The program binary will fail, setting the `LINK_STATUS` of `program` to `FALSE`, if these conditions are not met.

A program binary may also fail if the implementation determines that there has been a change in hardware or software configuration from when the program binary was produced such as having been compiled with an incompatible or outdated version of the compiler. In this case the application should fall back to providing the original OpenGL Shading Language source shaders, and perhaps again retrieve the program binary for future use.

A program object’s program binary is replaced by calls to `LinkProgram` or `ProgramBinary`. Where linking success or failure is concerned, `ProgramBinary` can be considered to perform an implicit linking operation. `LinkProgram` and `ProgramBinary` both set the program object’s `LINK_STATUS` to `TRUE` or `FALSE`, as queried with `GetProgramiv`, to reflect success or failure and update the information log, queried with `GetProgramInfoLog`, to provide details about warnings or errors.

A successful call to `ProgramBinary` will reset all uniform variables to their initial values. The initial value is either the value of the variable’s initializer as specified in the original shader source, or zero if no initializer was present.

Additionally, all vertex shader input and fragment shader output assignments that were in effect when the program was linked before saving are restored when `ProgramBinary` is called successfully.

If `ProgramBinary` failed, any information about a previous link or load of that program object is lost. Thus, a failed load does not restore the old state of `program`. The failure does not alter other program state not affected by linking such as the attached shaders, and the vertex attribute and fragment data location bindings as set by `BindAttribLocation` and `BindFragDataLocation`.

OpenGL defines no specific binary formats, but does provide a mechanism to obtain token values for such formats provided by extensions. The number of program binary formats supported can be obtained by querying the value of `NUM_PROGRAM_BINARY_FORMATS`. The list of specific binary formats supported can be obtained by querying the value of `PROGRAM_
2.11. VERTEX SHADERS

Queries of values NUM_PROGRAM_BINARY_FORMATS and PROGRAM_BINARY_FORMATS return the number of program binary formats and the list of program binary format values supported by an implementation. The binaryFormat returned by GetProgramBinary must be present in this list.

Any program binary retrieved using GetProgramBinary and submitted using ProgramBinary under the same configuration must be successful. Any programs loaded successfully by ProgramBinary must be run properly with any legal GL state vector. If an implementation needs to recompile or otherwise modify program executables based on GL state outside the program, GetProgramBinary is required to save enough information to allow such recompilation. To indicate that a program binary is likely to be retrieved, ProgramParameteri should be called with pname PROGRAM_BINARY_RETRIEVABLE_HINT and value GL_TRUE. This setting will not be in effect until the next time LinkProgram or ProgramBinary has been called successfully. Additionally, GetProgramBinary calls may be deferred until after using the program with all non-program state vectors that it is likely to encounter. Such deferral may allow implementations to save additional information in the program binary that would minimize recompilation in future uses of the program binary.

2.11.6 Vertex Attributes

Vertex shaders can define named attribute variables, which are bound to the generic vertex attributes that are set by VertexAttrib*. This binding can be specified by the application before the program is linked, or automatically assigned by the GL when the program is linked.

When an attribute variable declared using one of the scalar or vector data types enumerated in table 2.12 is bound to a generic attribute index i, its value(s) are taken from the components of generic attribute i. Scalars are extracted from the x component; two-, three-, and four-component vectors are extracted from the (x, y), (x, y, z), or (x, y, z, w) components, respectively.

When an attribute variable is declared as a mat2, mat3x2 or mat4x2, its matrix columns are taken from the (x, y) components of generic attributes i and i + 1 (mat2, dmat2), from attributes i through i + 2 (mat3x2), or from attributes i through i + 3 (mat4x2). When an attribute variable is declared as a mat2x3, mat3 or mat4x3, its matrix columns are taken from the (x, y, z) components of generic attributes i and i + 1 (mat2x3), from attributes i through i + 2 (mat3), or from attributes i through i + 3 (mat4x3). When an attribute variable is declared as a mat2x4, mat3x4 or mat4, its matrix columns are taken from the (x, y, z, w) components of generic attributes i and i + 1 (mat2x4), from attributes i through i + 2 (mat3x4), or from attributes i through i + 3 (mat4). When an attribute vari-
2.11. VERTEX SHADERS

<table>
<thead>
<tr>
<th>Data type</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>VertexAttribI1i</td>
</tr>
<tr>
<td>ivec2</td>
<td>VertexAttribI2i</td>
</tr>
<tr>
<td>ivec3</td>
<td>VertexAttribI3i</td>
</tr>
<tr>
<td>ivec4</td>
<td>VertexAttribI4i</td>
</tr>
<tr>
<td>uint</td>
<td>VertexAttribI1ui</td>
</tr>
<tr>
<td>uvec2</td>
<td>VertexAttribI2ui</td>
</tr>
<tr>
<td>uvec3</td>
<td>VertexAttribI3ui</td>
</tr>
<tr>
<td>uvec4</td>
<td>VertexAttribI4ui</td>
</tr>
<tr>
<td>float</td>
<td>VertexAttrib1*</td>
</tr>
<tr>
<td>vec2</td>
<td>VertexAttrib2*</td>
</tr>
<tr>
<td>vec3</td>
<td>VertexAttrib3*</td>
</tr>
<tr>
<td>vec4</td>
<td>VertexAttrib4*</td>
</tr>
<tr>
<td>double</td>
<td>VertexAttribL1d</td>
</tr>
<tr>
<td>dvec2</td>
<td>VertexAttribL2d</td>
</tr>
<tr>
<td>dvec3</td>
<td>VertexAttribL3d</td>
</tr>
<tr>
<td>dvec4</td>
<td>VertexAttribL4d</td>
</tr>
</tbody>
</table>

Table 2.12: Scalar and vector vertex attribute types and `VertexAttrib*` commands used to set the values of the corresponding generic attribute.

able is declared as a double-precision matrix (dmat2, dmat3, dmat4, dmat2x3, dmat2x4, dmat3x2, dmat3x4, dmat4x2, dmat4x3), its matrix columns are taken from the same generic attributes as the equivalent single-precision matrix type, with values specified using the `VertexAttribL*` or `VertexAttribLPointer` commands.

For the 64-bit double precision types listed in table 2.12, no default attribute values are provided if the values of the vertex attribute variable are specified with fewer components than required for the attribute variable. For example, the fourth component of a variable of type `dvec4` will be undefined if specified using `VertexAttribL3dv`, or using a vertex array specified with `VertexAttribLPointer` and a size of three.

A generic attribute variable is considered **active** if it is determined by the compiler and linker that the attribute may be accessed when the shader is executed. Attribute variables that are declared in a vertex shader but never used will not count as **active vertex attributes**. In cases where the compiler and linker cannot make a conclusive determination, an attribute will be considered active. A program object will fail to link if the number of active vertex attributes exceeds...
MAX_VERTEX_ATTRIBS. For the purposes of this comparison, attribute variables of the type dvec3, dvec4, dmat2x3, dmat2x4, dmat3, dmat3x4, dmat4x3, and dmat4 may count as consuming twice as many attributes as equivalent single-precision types. While these types use the same number of generic attributes as their single-precision equivalents, implementations are permitted to consume two single-precision vectors of internal storage for each three- or four-component double-precision vector.

To determine the set of active vertex attributes used by a program, and to determine their types, use the command:

```c
void GetActiveAttrib(uint program, uint index,
sizei bufSize, sizei *length, int*size, enum *type,
char *name );
```

This command provides information about the attribute selected by index. An index of 0 selects the first active attribute, and an index of the value of ACTIVE_ATTRIBUTES minus one selects the last active attribute. The value of ACTIVE_ATTRIBUTES can be queried with GetProgramiv (see section 6.1.12). If index is greater than or equal to ACTIVE_ATTRIBUTES, the error INVALID_VALUE is generated. Note that index simply identifies a member in a list of active attributes, and has no relation to the generic attribute that the corresponding variable is bound to.

The parameter program is the name of a program object for which the command LinkProgram or ProgramBinary has been issued in the past. It is not necessary for program to have been linked successfully. The link could have failed because the number of active attributes exceeded the limit.

The name of the selected attribute is returned as a null-terminated string in name. The actual number of characters written into name, excluding the null terminator, is returned in length. If length is NULL, no length is returned. The maximum number of characters that may be written into name, including the null terminator, is specified by bufSize. The returned attribute name must be the name of a generic attribute. The length of the longest attribute name in program is given by ACTIVE_ATTRIBUTE_MAX_LENGTH, which can be queried with GetProgramiv (see section 6.1.12).

For the selected attribute, the type of the attribute is returned into type. The size of the attribute is returned into size. The value in size is in units of the type returned in type. The type returned can be any of the types whose “Attrib” column is checked in table 2.13.

If an error occurred, the return parameters length, size, type and name will be unmodified.

This command will return as much information about active attributes as possible. If no information is available, length will be set to zero and name will

OpenGL 4.2 (Core Profile) - April 27, 2012
be an empty string. This situation could arise if `GetActiveAttrib` is issued after a failed link.

After a program object has been linked successfully, the bindings of attribute variable names to indices can be queried. The command

```c
int GetAttribLocation(uint program, const char *name);
```

returns the generic attribute index that the attribute variable named `name` was bound to when the program object named `program` was last linked. `name` must be a null-terminated string. If `name` is active and is an attribute matrix, `GetAttribLocation` returns the index of the first column of that matrix. If `program` has not been successfully linked, the error `INVALID_OPERATION` is generated. If `name` is not an active attribute, or if an error occurs, -1 will be returned.

The binding of an attribute variable to a generic attribute index can also be specified explicitly. The command

```c
void BindAttribLocation(uint program, uint index, const char *name);
```

specifies that the attribute variable named `name` in program `program` should be bound to generic vertex attribute `index` when the program is next linked. If `name` was bound previously, its assigned binding is replaced with `index`. `name` must be a null-terminated string. The error `INVALID_VALUE` is generated if `index` is equal or greater than `MAX_VERTEX_ATTRIBS`. `BindAttribLocation` has no effect until the program is linked. In particular, it doesn’t modify the bindings of active attribute variables in a program that has already been linked.

When a program is linked, any active attributes without a binding specified either through `BindAttribLocation` or explicitly set within the shader text will automatically be bound to vertex attributes by the GL. Such bindings can be queried using the command `GetAttribLocation`. `LinkProgram` will fail if the assigned binding of an active attribute variable would cause the GL to reference a non-existent generic attribute (one greater than or equal to the value of `MAX_VERTEX_ATTRIBS`). `LinkProgram` will fail if the attribute bindings specified either by `BindAttribLocation` or explicitly set within the shader text do not leave not enough space to assign a location for an active matrix attribute or an active attribute array, both of which require multiple contiguous generic attributes. If an active attribute has a binding explicitly set within the shader text and a different binding assigned by `BindAttribLocation`, the assignment in the shader text is used.

`BindAttribLocation` may be issued before any vertex shader objects are attached to a program object. Hence it is allowed to bind any name to an index,
including a name that is never used as an attribute in any vertex shader object. Assigned bindings for attribute variables that do not exist or are not active are ignored.

The values of generic attributes sent to generic attribute index \( i \) are part of current state. If a new program object has been made active, then these values will be tracked by the GL in such a way that the same values will be observed by attributes in the new program object that are also bound to index \( i \).

It is possible for an application to bind more than one attribute name to the same location. This is referred to as aliasing. This will only work if only one of the aliased attributes is active in the executable program, or if no path through the shader consumes more than one attribute of a set of attributes aliased to the same location. A link error can occur if the linker determines that every path through the shader consumes multiple aliased attributes, but implementations are not required to generate an error in this case. The compiler and linker are allowed to assume that no aliasing is done, and may employ optimizations that work only in the absence of aliasing.

### 2.11.7 Uniform Variables

Shaders can declare named uniform variables, as described in the OpenGL Shading Language Specification. Values for these uniforms are constant over a primitive, and typically they are constant across many primitives. A uniform is considered active if it is determined by the compiler and linker that the uniform will actually be accessed when the executable code is executed. In cases where the compiler and linker cannot make a conclusive determination, the uniform will be considered active.

Sets of uniforms, except for atomic counters, images, samplers, and subroutine uniforms, can be grouped into uniform blocks. The values of each uniform in such a set are extracted from the data store of a buffer object corresponding to the uniform block. OpenGL Shading Language syntax serves to delimit named blocks of uniforms that can be backed by a buffer object. These are referred to as named uniform blocks, and are assigned a uniform block index. Uniforms that are declared outside of a named uniform block are said to be part of the default uniform block. Default uniform blocks have no name or uniform block index. Uniforms in the default uniform block, except for subroutine uniforms, are program object-specific state. They retain their values once loaded, and their values are restored whenever a program object is used, as long as the program object has not been re-linked. Like uniforms, uniform blocks can be active or inactive. Active uniform blocks are those that contain active uniforms after a program has been compiled and linked.

The amount of storage available for uniform variables, except for subroutine uniforms and atomic counters, in the default uniform block accessed by a
2.11. VERTEX SHADERS

vertex shader is specified by the value of the implementation-dependent con-
stant MAX_VERTEX_UNIFORM_COMPONENTS. The implementation-dependent con-
stant MAX_VERTEX_UNIFORM_VECTORS has a value equal to the value of MAX_-
VERTEX_UNIFORM_COMPONENTS divided by four. The total amount of com-
bined storage available for uniform variables in all uniform blocks accessed by
a vertex shader (including the default uniform block) is specified by the value
of the implementation-dependent constant MAX_COMBINED_VERTEX_UNIFORM_-
COMPONENTS. These values represent the numbers of individual floating-point, in-
teger, or boolean values that can be held in uniform variable storage for a vertex
shader. A uniform matrix in the default uniform block with single- or double-
precision components will consume no more than $4 \times \min(r, c)$ or $8 \times \min(r, c)$
uniform components, respectively. A scalar or vector uniform with double-
precision components will consume no more than $2n$ components, where $n$ is 1
for scalars, and the component count for vectors. A link error is generated if an
attempt is made to utilize more than the space available for vertex shader uniform
variables.

When a program is successfully linked, all active uniforms, except for atomic
counters, belonging to the program object’s default uniform block are initialized
as defined by the version of the OpenGL Shading Language used to compile the
program. A successful link will also generate a location for each active uniform
in the default uniform block. The values of active uniforms in the default uniform
block can be changed using this location and the appropriate Uniform* command
(see below). These locations are invalidated and new ones assigned after each
successful re-link.

Similarly, when a program is successfully linked, all active atomic counters
are assigned bindings, offsets (and strides for arrays of atomic counters) according
to layout rules described below. Atomic counter uniform buffer objects provide
the storage for atomic counters, so the values of atomic counters may be changed
by modifying the contents of the buffer object using commands such as Buffer-
Data, BufferSubData, MapBuffer, and UnmapBuffer. Atomic counters are not
assigned a location and may not be modified using the Uniform* commands. The
bindings, offsets, and strides belonging to atomic counters of a program object are
invalidated and new ones assigned after each successful re-link.

Similarly, when a program is successfully linked, all active uniforms belong-
ing to the program’s named uniform blocks are assigned offsets (and strides for
array and matrix type uniforms) within the uniform block according to layout rules
described below. Uniform buffer objects provide the storage for named uniform
blocks, so the values of active uniforms in named uniform blocks may be changed
by modifying the contents of the buffer object using commands such as Buffer-
Data, BufferSubData, MapBuffer, and UnmapBuffer. Uniforms in a named
uniform block are not assigned a location and may not be modified using the **Uniform** commands. The offsets and strides of all active uniforms belonging to named uniform blocks of a program object are invalidated and new ones assigned after each successful re-link.

To find the location within a program object of an active uniform variable associated with the default uniform block, use the command

```c
int GetUniformLocation(uint program, const char *name);
```

This command will return the location of uniform variable `name` if it is associated with the default uniform block. `name` must be a null-terminated string, without white space. The value -1 will be returned if `name` does not correspond to an active uniform variable name in `program`, if `name` is associated with an atomic counter, or if `name` is associated with a named uniform block.

If `program` has not been successfully linked, the error **INVALID_OPERATION** is generated. After a program is linked, the location of a uniform variable will not change, unless the program is re-linked.

A valid `name` cannot be a structure, an array of structures, or any portion of a single vector or a matrix. In order to identify a valid `name`, the "." (dot) and "[]" operators can be used in `name` to specify a member of a structure or element of an array.

The first element of a uniform array is identified using the name of the uniform array appended with "[0]". Except if the last part of the string `name` indicates a uniform array, then the location of the first element of that array can be retrieved by either using the name of the uniform array, or the name of the uniform array appended with "[0]".

Locations for sequential array indices are not required to be sequential. The location for "a[1]" may or may not be equal to the location for "a[0]" +1. Furthermore, since unused elements at the end of uniform arrays may be trimmed (see the discussion of the `size` parameter of **GetActiveUniform**), the location of the i + 1'th array element may not be valid even if the location of the i’th element is valid. As a direct consequence, the value of the location of "a[0]" +1 may refer to a different uniform entirely. Applications that wish to set individual array elements should query the locations of each element separately.

Named uniform blocks, like uniforms, are identified by name strings. Uniform block indices corresponding to uniform block names can be queried by calling

```c
uint GetUniformBlockIndex(uint program, const char *uniformBlockName);
```
2.11. VERTEX SHADERS

__program__ is the name of a program object for which the command __LinkProgram__ has been issued in the past. It is not necessary for __program__ to have been linked successfully. The link could have failed because the number of active uniforms exceeded the limit.

__uniformBlockName__ must contain a null-terminated string specifying the name of a uniform block in the program object __program__.

__GetUniformBlockIndex__ returns the uniform block index for the uniform block named __uniformBlockName__ of __program__. If __uniformBlockName__ does not identify an active uniform block of __program__, or an error occurred, then __INVALID_INDEX__ is returned. The indices of the active uniform blocks of a program are assigned in consecutive order, beginning with zero.

An active uniform block’s name string can be queried from its uniform block index by calling

```c
void GetActiveUniformBlockName( uint program,
    uint uniformBlockIndex, sizei bufSize, sizei *length,
    char *uniformBlockName );
```

__program__ is the name of a program object for which the command __LinkProgram__ has been issued in the past. It is not necessary for __program__ to have been linked successfully. The link could have failed because the number of active uniforms exceeded the limit.

__uniformBlockIndex__ must be an active uniform block index of the program object __program__, in the range zero to the value of __ACTIVE_UNIFORM_BLOCKS__ minus one. The value of __ACTIVE_UNIFORM_BLOCKS__ can be queried with __GetProgramiv__ (see section 6.1.12). If __uniformBlockIndex__ is greater than or equal to the value of __ACTIVE_UNIFORM_BLOCKS__, the error __INVALID_VALUE__ is generated.

The string name of the uniform block identified by __uniformBlockIndex__ is returned into __uniformBlockName__. The name is null-terminated. The actual number of characters written into __uniformBlockName__, excluding the null terminator, is returned in __length__. If __length__ is __NULL__, no length is returned.

__bufSize__ contains the maximum number of characters (including the null terminator) that will be written back to __uniformBlockName__.

If an error occurs, nothing will be written to __uniformBlockName__ or __length__.

Information about an active uniform block can be queried by calling

```c
void GetActiveUniformBlockiv( uint program,
    uint uniformBlockIndex, enum pname, int *params );
```

__program__ is the name of a program object for which the command __LinkProgram__ has been issued in the past. It is not necessary for __program__ to have been linked successfully.

__uniformBlockIndex__ must be an active uniform block index of the program object __program__, in the range zero to the value of __ACTIVE_UNIFORM_BLOCKS__ minus one. The value of __ACTIVE_UNIFORM_BLOCKS__ can be queried with __GetProgramiv__ (see section 6.1.12). If __uniformBlockIndex__ is greater than or equal to the value of __ACTIVE_UNIFORM_BLOCKS__, the error __INVALID_VALUE__ is generated.

The string name of the uniform block identified by __uniformBlockIndex__ is returned into __uniformBlockName__. The name is null-terminated. The actual number of characters written into __uniformBlockName__, excluding the null terminator, is returned in __length__. If __length__ is __NULL__, no length is returned.

__bufSize__ contains the maximum number of characters (including the null terminator) that will be written back to __uniformBlockName__.

If an error occurs, nothing will be written to __uniformBlockName__ or __length__.

Information about an active uniform block can be queried by calling

```c
void GetActiveUniformBlockiv( uint program,
    uint uniformBlockIndex, enum pname, int *params );
```
been linked successfully. The link could have failed because the number of active uniforms exceeded the limit.

uniformBlockIndex is an active uniform block index of the program object program. If uniformBlockIndex is greater than or equal to the value of ACTIVE_UNIFORM_BLOCKS, or is not the index of an active uniform block in program, the error INVALID_VALUE is generated.

If no error occurs, the uniform block parameter(s) specified by pname are returned in params. Otherwise, nothing will be written to params.

If pname is UNIFORM_BLOCK_BINDING, then the index of the uniform buffer binding point associated with uniformBlockIndex is returned.

If pname is UNIFORM_BLOCK_DATA_SIZE, then the implementation-dependent minimum total buffer object size, in basic machine units, required to hold all active uniforms in the uniform block identified by uniformBlockIndex is returned. It is neither guaranteed nor expected that a given implementation will arrange uniform values as tightly packed in a buffer object. The exception to this is the std140 uniform block layout, which guarantees specific packing behavior and does not require the application to query for offsets and strides. In this case the minimum size may still be queried, even though it is determined in advance based only on the uniform block declaration (see “Standard Uniform Block Layout” in section 2.11.7).

The total amount of buffer object storage available for any given uniform block is subject to an implementation-dependent limit. The maximum amount of available space, in basic machine units, can be queried by calling GetIntegerv with the constant MAX_UNIFORM_BLOCK_SIZE. If the amount of storage required for a uniform block exceeds this limit, a program may fail to link.

If pname is UNIFORM_BLOCK_NAME_LENGTH, then the total length (including the null terminator) of the name of the uniform block identified by uniformBlockIndex is returned.

If pname is UNIFORM_BLOCK_ACTIVE_UNIFORMS, then the number of active uniforms in the uniform block identified by uniformBlockIndex is returned.

If pname is UNIFORM_BLOCK_ACTIVE_UNIFORM_INDICES, then a list of the active uniform indices for the uniform block identified by uniformBlockIndex is returned. The number of elements that will be written to params is the value of UNIFORM_BLOCK_ACTIVE_UNIFORMS for uniformBlockIndex.

If pname is UNIFORM_BLOCK_REFERENCED_BY_VERTEX_SHADER, UNIFORM_BLOCK_REFERENCED_BY_TESS_CONTROL_SHADER, UNIFORM_BLOCK_REFERENCED_BY_TESS_EVALUATION_SHADER, UNIFORM_BLOCK_REFERENCED_BY_GEOMETRY_SHADER, or UNIFORM_BLOCK_REFERENCED_BY_FRAGMENT_SHADER, then a boolean value indicating whether the uniform block identified by uniformBlockIndex is referenced by the vertex, tessellation...
control, tessellation evaluation, geometry, or fragment programming stages of 
*program*, respectively, is returned.

In programs with active atomic counter uniforms, each buffer object binding 
point associated with one or more active atomic counters is considered an active 
atomic counter buffer. Information about the set of active atomic counter buffers 
for a program can be obtained by calling

```c
void GetActiveAtomicCounterBufferiv( uint program, 
  uint bufferIndex, enum pname, int *params );
```

*program* is the name of a program object for which the command 
*LinkProgram* has been issued in the past. It is not necessary for *program* to have 
been linked successfully. The link could have failed because the number of active 
atomic counters exceeded the limits.

*bufferIndex* specifies the index of an active atomic counter buffer in the pro-
gram object *program* and must be in the range zero to the value of 
ACTIVE_ATOMIC_COUNTER_BUFFERS minus one. The value of 
ACTIVE_ATOMIC_COUNTER_BUFFERS for *program* indicates the number of active atomic counter 
buffers and can be queried with *GetProgramiv* (see section 6.1.12). If *bufferIndex* 
is greater than or equal to the value of ACTIVE_ATOMIC_COUNTER_BUFFERS, the 
error INVALID_VALUE is generated.

If no error occurs, the parameter(s) specified by *pname* are returned in *params*. 
Otherwise, nothing will be written to *params*.

If *pname* is ATOMIC_COUNTER_BUFFER_BINDING, then the index of the 
atomic counter buffer binding point associated with the active atomic counter 
buffer *bufferIndex* for *program* is returned.

If *pname* is ATOMIC_COUNTER_BUFFER_DATA_SIZE, then the implementation-dependent 
minimum total buffer object size, in basic machine units, required to hold all active 
atomic counters in the atomic counter binding point identified by *bufferIndex* is 
returned.

The total amount of buffer object storage accessible in any given atomic 
counter buffer is subject to an implementation-dependent limit. The maximum 
amount of storage accessible to atomic counters, in basic machine units, can 
be queried by calling *GetIntegerv* with the constant MAX_ATOMIC_COUNTER_-
BUFFER_SIZE. If the amount of storage required for a atomic counter buffer ex-
cedes this limit, a program may fail to link.

If *pname* is ATOMIC_COUNTER_BUFFER_ACTIVE_ATOMIC_COUNTERS, then 
the number of active atomic counters variables associated with the atomic counter 
buffer identified by *bufferIndex* is returned.
If `pname` is `ATOMIC_COUNTER_BUFFER_ACTIVE_ATOMIC_COUNTER_INDICES`, then a list of the active atomic counter indices for the atomic counter buffer identified by `bufferIndex` is returned. The number of elements that will be written to `params` is the value of `ATOMIC_COUNTER_BUFFER_ACTIVE_ATOMIC+COUNTERS` for `bufferIndex`.

If `pname` is `ATOMIC_COUNTER_BUFFER_REFERENCED_BY_VERTEX_SHADER`, `ATOMIC_COUNTER_BUFFER_REFERENCED_BY_TESS_CONTROL_SHADER`, `UNIFORM_BLOCK_REFERENCED_BY_TESS_EVALUATION_SHADER`, `ATOMIC_COUNTER_BUFFER_REFERENCED_BY_GEOMETRY_SHADER`, or `ATOMIC_COUNTER_BUFFER_REFERENCED_BY_FRAGMENT_SHADER`, then a boolean value indicating whether the atomic counter buffer identified by `bufferIndex` is referenced by the vertex, tessellation control, tessellation evaluation, geometry, or fragment programming stages of `program`, respectively, is returned.

Each active uniform, except for subroutine uniforms, and whether in the default block, in a named uniform block, or an atomic counter, is assigned an index when a program is linked. Indices are assigned in consecutive order, beginning with zero. Structures, arrays of structures, and individual elements of vectors and matrices are not assigned indices. Instead, each uniform variable, declared in the shader, is broken down into one or more strings, using the " . " (dot) and " [ ] " if necessary, to the point that it is a built-in type or an array whose elements are of a built-in type, and each of these is assigned an index. An array whose elements are of a built-in type is assigned only one index (and its elements are not assigned indices), but the index may be queried using either the name of the array, or the name of the array with " [0] " appended. Individual elements of vectors and matrices are never assigned indices.

The indices assigned to a set of uniforms in a program may be queried by calling

```c
void GetUniformIndices( uint program,
  sizei uniformCount, const char *const
  *uniformNames, uint *uniformIndices );
```

- `program` is the name of a program object for which the command `LinkProgram` has been issued in the past. It is not necessary for `program` to have been linked successfully. The link could have failed because the number of active uniforms exceeded the limit.
- `uniformCount` indicates both the number of elements in the array of names `uniformNames` and the number of indices that may be written to `uniformIndices`.
- `uniformNames` contains a list of `uniformCount` null-terminated name strings identifying the uniform names to be queried for indices. For each name string
in \texttt{uniformNames}, the index assigned to the active uniform of that name in the program object \texttt{program} will be written to the corresponding element of \texttt{uniformIndices}. If a string in \texttt{uniformNames} is not the name of an active uniform, which has been assigned an index, the value \texttt{INVALID_INDEX} will be written to the corresponding element of \texttt{uniformIndices}.

If an error occurs, nothing is written to \texttt{uniformIndices}.

The name of an active uniform, except for subroutine uniforms, may be queried from the corresponding uniform index by calling

\begin{verbatim}
void GetActiveUniformName( uint program,
    uint uniformIndex, sizei bufSize, sizei *length,
    char *uniformName );
\end{verbatim}

\texttt{program} is the name of a program object for which the command \texttt{LinkProgram} has been issued in the past. It is not necessary for \texttt{program} to have been linked successfully. The link could have failed because the number of active uniforms exceeded the limit.

\texttt{uniformIndex} must be an active uniform index of the program \texttt{program}, in the range zero to the value of \texttt{ACTIVE_UNIFORMS} minus one. The value of \texttt{ACTIVE_UNIFORMS} can be queried with \texttt{GetProgramiv}. If \texttt{uniformIndex} is greater than or equal to the value of \texttt{ACTIVE_UNIFORMS}, the error \texttt{INVALID_VALUE} is generated.

The name of the uniform identified by \texttt{uniformIndex} is returned as a null-terminated string in \texttt{uniformName}. The actual number of characters written into \texttt{uniformName}, excluding the null terminator, is returned in \texttt{length}. If \texttt{length} is \texttt{NULL}, no length is returned. The maximum number of characters that may be written into \texttt{uniformName}, including the null terminator, is specified by \texttt{bufSize}. The returned uniform name can be the name of built-in uniform state as well. The complete list of built-in uniform state is described in section 7.5 of the OpenGL Shading Language Specification. The length of the longest uniform name in \texttt{program} is given by the value of \texttt{ACTIVE_UNIFORM_MAX_LENGTH}, which can be queried with \texttt{GetProgramiv}.

If \texttt{GetActiveUniformName} is not successful, nothing is written to \texttt{length} or \texttt{uniformName}.

Each active uniform variable, except for subroutine uniforms, is broken down into one or more strings using the "." (dot) and "[ ]" operators, if necessary, to the point that it is legal to pass each string back into \texttt{GetUniformIndices}.

If the active uniform is an array, the uniform name returned in \texttt{name} will always be the name of the uniform array appended with "[0]".

Information about active uniforms, except for subroutine uniforms, can be obtained by calling either

\begin{verbatim}
OpenGL 4.2 (Core Profile) - April 27, 2012
\end{verbatim}
2.11. VERTEX SHADERS

```c
void GetActiveUniform(uint program, uint index,
sizei bufSize, sizei *length, int *size, enum *type,
char *name);
```

or

```c
void GetActiveUniformsiv(uint program,
sizei uniformCount, const uint *uniformIndices,
enum pname, int *params);
```

`program` is the name of a program object for which the command `LinkProgram` has been issued in the past. It is not necessary for `program` to have been linked successfully. The link could have failed because the number of active uniforms exceeded the limit.

These commands provide information about the uniform or uniforms in the program object `program` selected by `index` or `uniformIndices`, respectively. In `GetActiveUniform`, an `index` of 0 selects the first active uniform, and an `index` of the value of `ACTIVE_UNIFORMS` minus one selects the last active uniform. In `GetActiveUniformsiv`, `uniformIndices` is an array of such active uniform indices. If any index is greater than or equal to the value of `ACTIVE_UNIFORMS`, the error `INVALID_VALUE` is generated.

For the selected uniform, `GetActiveUniform` returns the uniform name as a null-terminated string in `name`. The actual number of characters written into `name`, excluding the null terminator, is returned in `length`. If `length` is `NULL`, no length is returned. The maximum number of characters that may be written into `name`, including the null terminator, is specified by `bufSize`. The returned uniform name can be the name of built-in uniform state as well. The complete list of built-in uniform state is described in section 7.5 of the OpenGL Shading Language Specification. The length of the longest uniform name in `program` is given by `ACTIVE_UNIFORM_MAX_LENGTH`.

For the selected uniform, `GetActiveUniform` returns the type of the uniform into `type` and the size of the uniform is into `size`. The value in `size` is in units of the uniform type, which can be any of the type name tokens in table 2.13, corresponding to OpenGL Shading Language type keywords also shown in that table.

If one or more elements of an array are active, `GetActiveUniform` will return the name of the array in `name`, subject to the restrictions listed above. The type of the array is returned in `type`. The `size` parameter contains the highest array element index used, plus one. The compiler or linker determines the highest index used. There will be only one active uniform reported by the GL per uniform array.

`GetActiveUniform` will return as much information about active uniforms as possible. If no information is available, `length` will be set to zero and `name` will be

OpenGL 4.2 (Core Profile) - April 27, 2012
2.11. VERTEX SHADERS

an empty string. This situation could arise if GetActiveUniform is issued after a failed link.

If an error occurs, nothing is written to length, size, type, or name.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name Token</th>
<th>Keyword</th>
<th>Attrib</th>
<th>Xlib</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOAT</td>
<td>float</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>FLOAT_VEC2</td>
<td>vec2</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>FLOAT_VEC3</td>
<td>vec3</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>FLOAT_VEC4</td>
<td>vec4</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td>double</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>DOUBLE_VEC2</td>
<td>dvec2</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>DOUBLE_VEC3</td>
<td>dvec3</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>DOUBLE_VEC4</td>
<td>dvec4</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>INT</td>
<td>int</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>INT_VEC2</td>
<td>ivec2</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>INT_VEC3</td>
<td>ivec3</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>INT_VEC4</td>
<td>ivec4</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT</td>
<td>uint</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT_VEC2</td>
<td>uvec2</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT_VEC3</td>
<td>uvec3</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT_VEC4</td>
<td>uvec4</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>BOOL</td>
<td>bool</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>BOOL_VEC2</td>
<td>bvec2</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>BOOL_VEC3</td>
<td>bvec3</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>BOOL_VEC4</td>
<td>bvec4</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT2</td>
<td>mat2</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT3</td>
<td>mat3</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT4</td>
<td>mat4</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT2x3</td>
<td>mat2x3</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT2x4</td>
<td>mat2x4</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT3x2</td>
<td>mat3x2</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT3x4</td>
<td>mat3x4</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT4x2</td>
<td>mat4x2</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT4x3</td>
<td>mat4x3</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>DOUBLE_MAT2</td>
<td>dmat2</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>DOUBLE_MAT3</td>
<td>dmat3</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>DOUBLE_MAT4</td>
<td>dmat4</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
</tbody>
</table>

(Continued on next page)
### OpenGL Shading Language Type Tokens (continued)

<table>
<thead>
<tr>
<th>Type Name Token</th>
<th>Keyword</th>
<th>Attrib</th>
<th>Xfb</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE_MAT2x3</td>
<td>dmat2x3</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>DOUBLE_MAT2x4</td>
<td>dmat2x4</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>DOUBLE_MAT3x2</td>
<td>dmat3x2</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>DOUBLE_MAT3x4</td>
<td>dmat3x4</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>DOUBLE_MAT4x2</td>
<td>dmat4x2</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>DOUBLE_MAT4x3</td>
<td>dmat4x3</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>SAMPLER_1D</td>
<td>sampler1D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAMPLER_2D</td>
<td>sampler2D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAMPLER_3D</td>
<td>sampler3D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAMPLER_CUBE</td>
<td>samplerCube</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAMPLER_1D_SHADOW</td>
<td>sampler1DShadow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAMPLER_2D_SHADOW</td>
<td>sampler2DShadow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAMPLER_1D_ARRAY</td>
<td>sampler1DArray</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAMPLER_2D_ARRAY</td>
<td>sampler2DArray</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAMPLER_CUBE_MAP_ARRAY</td>
<td>samplerCubeArray</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAMPLER_1D_ARRAY_SHADOW</td>
<td>sampler1DArrayShadow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAMPLER_2D_ARRAY_SHADOW</td>
<td>sampler2DArrayShadow</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAMPLER_2D_MULTISAMPLE</td>
<td>sampler2DMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAMPLER_2D_MULTISAMPLE_ ARRAY</td>
<td>sampler2DMSArray</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT_SAMPLER_1D</td>
<td>isampler1D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT_SAMPLER_2D</td>
<td>isampler2D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT_SAMPLER_3D</td>
<td>isampler3D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT_SAMPLER_CUBE</td>
<td>isamplerCube</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT_SAMPLER_1D_ARRAY</td>
<td>isampler1DArray</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT_SAMPLER_2D_ARRAY</td>
<td>isampler2DArray</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT_SAMPLER_CUBE_MAP_ARRAY</td>
<td>isamplerCubeArray</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Continued on next page)
### 2.11. VERTEX SHADERS

<table>
<thead>
<tr>
<th>Type Name Token</th>
<th>Keyword</th>
<th>Attrib</th>
<th>Xfb</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT_SAMPLER_2D_MULTISAMPLE</td>
<td>isampler2DMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT_SAMPLER_2D_MULTISAMPLE_ARRAY</td>
<td>isampler2DMSArray</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT_SAMPLER_BUFFER</td>
<td>isamplerBuffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT_SAMPLER_2D_RECT</td>
<td>isampler2DRect</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT_SAMPLER_1D</td>
<td>usampler1D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT_SAMPLER_2D</td>
<td>usampler2D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT_SAMPLER_3D</td>
<td>usampler3D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT_SAMPLER_CUBE</td>
<td>usamplerCube</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT_SAMPLER_1D_ARRAY</td>
<td>usampler1DArray</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT_SAMPLER_2D_ARRAY</td>
<td>usampler2DArray</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT_SAMPLER_CUBE_ARRAY</td>
<td>usamplerCubeArray</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT_SAMPLER_CUBE_MAP_ARRAY</td>
<td>usampler2DMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT_SAMPLER_2D_MULTISAMPLE</td>
<td>usampler2DMSArray</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT_SAMPLER_BUFFER</td>
<td>usamplerBuffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT_SAMPLER_2D_RECT</td>
<td>usampler2DRect</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMAGE_1D</td>
<td>image1D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMAGE_2D</td>
<td>image2D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMAGE_3D</td>
<td>image3D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMAGE_2D_RECT</td>
<td>image2DRect</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMAGE_CUBE</td>
<td>imageCube</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMAGE_BUFFER</td>
<td>imageBuffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMAGE_1D_ARRAY</td>
<td>image1DArray</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMAGE_2D_ARRAY</td>
<td>image2DArray</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMAGE_CUBE_MAP_ARRAY</td>
<td>imageCubeArray</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMAGE_2D_MULTISAMPLE</td>
<td>image2DMS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Continued on next page)
Table 2.13: OpenGL Shading Language type tokens returned by `GetActiveUniform` and `GetActiveUniformsiv`, and corresponding shading language keywords declaring each such type. Types whose “Attrib” column are marked may be declared as vertex attributes (see section 2.11.6). Types whose “Xfb” column are marked may be the types of variable returned by transform feedback (see section 2.11.11).
2.11. VERTEX SHADERS

For `GetActiveUniformsiv`, `uniformCount` indicates both the number of elements in the array of indices `uniformIndices` and the number of parameters written to `params` upon successful return. `pname` identifies a property of each uniform in `uniformIndices` that should be written into the corresponding element of `params`. If an error occurs, nothing will be written to `params`.

If `pname` is `UNIFORM_TYPE`, then an array identifying the types of the uniforms specified by the corresponding array of `uniformIndices` is returned. The returned types can be any of the values in table 2.13.

If `pname` is `UNIFORM_SIZE`, then an array identifying the size of the uniforms specified by the corresponding array of `uniformIndices` is returned. The sizes returned are in units of the type returned by a query of `UNIFORM_TYPE`. For active uniforms that are arrays, the size is the number of active elements in the array; for all other uniforms, the size is one.

If `pname` is `UNIFORM_NAME_LENGTH`, then an array identifying the length, including the terminating null character, of the uniform name strings specified by the corresponding array of `uniformIndices` is returned.

If `pname` is `UNIFORM_BLOCK_INDEX`, then an array identifying the uniform block index of each of the uniforms specified by the corresponding array of `uniformIndices` is returned. The index of a uniform associated with the default uniform block is -1.

If `pname` is `UNIFORM_OFFSET`, then an array of buffer offsets is returned. For uniforms in a named uniform block, the returned value will be its offset, in basic machine units, relative to the beginning of the uniform block in the buffer object data store. For atomic counter uniforms, the returned value will be its offset relative to the beginning of its active atomic counter buffer. For all other uniforms, an offset of -1 will be returned.

If `pname` is `UNIFORM_ARRAY_STRIDE`, then an array of strides between array elements in buffer object storage is returned. For uniforms in named uniform blocks and for uniforms declared as atomic counters, the stride is the difference, in basic machine units, of the offsets of consecutive elements in an array, or zero for uniforms not declared as an array. For all other uniforms, a stride of -1 will be returned.

If `pname` is `UNIFORM_MATRIX_STRIDE`, then an array identifying the stride between columns of a column-major matrix or rows of a row-major matrix, in basic machine units, of each of the uniforms specified by the corresponding array of `uniformIndices` is returned. The matrix stride of a uniform associated with the default uniform block is -1. Note that this information only makes sense for uniforms that are matrices. For uniforms that are not matrices, but are declared in a named
uniform block, a matrix stride of zero is returned.

If \textit{pname} is \texttt{UNIFORM\_IS\_ROW\_MAJOR}, then an array identifying whether each of the uniforms specified by the corresponding array of \texttt{uniformIndices} is a row-major matrix or not is returned. A value of one indicates a row-major matrix, and a value of zero indicates a column-major matrix, a matrix in the default uniform block, or a non-matrix.

If \textit{pname} is \texttt{UNIFORM\_ATOMIC\_COUNTER\_BUFFER\_INDEX}, then an array identifying the active atomic counter buffer index of each of the uniforms specified by the corresponding array of \texttt{uniformIndices} is returned. For uniforms other than atomic counters, the returned buffer index is -1. The returned indices can be passed to \texttt{GetActiveAtomicCounterBufferiv} to query properties of the associated buffer, and not necessarily the binding point specified in the uniform declaration.

**Loading Uniform Variables In The Default Uniform Block**

To load values into the uniform variables except for subroutine uniforms and atomic counters, of the default uniform block of the active program object, use the commands

\begin{verbatim}
void Uniform{1234}{ifd}( int location, T value );
void Uniform{1234}{ifd}v( int location, sizei count,  
const T value );
void Uniform{1234}ui( int location, T value );
void Uniform{1234}uiv( int location, sizei count, const  
T value );
void UniformMatrix{234}{fd}v( int location, sizei count,  
boolean transpose, const float *value );
void UniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}{fd}v(  
int location, sizei count, boolean transpose, const  
float *value );
\end{verbatim}

If a non-zero program object is bound by \texttt{UseProgram}, it is the active program object whose uniforms are updated by these commands. If no program object is bound using \texttt{UseProgram}, the active program object of the current program pipeline object set by \texttt{ActiveShaderProgram} is the active program object. If the current program pipeline object has no active program or there is no current program pipeline object, then there is no active program.

The given values are loaded into the default uniform block uniform variable location identified by \textit{location}.
The **Uniform*f{v}** commands will load \( count \) sets of one to four floating-point values into a uniform location defined as a float, a floating-point vector, an array of floats, or an array of floating-point vectors.

The **Uniform*d{v}** commands will load \( count \) sets of one to four double-precision floating-point values into a uniform location defined as a double, a double vector, or an array of double scalars or vectors.

The **Uniform*i{v}** commands will load \( count \) sets of one to four integer values into a uniform location defined as a sampler, an integer, an integer vector, an array of samplers, an array of integers, or an array of integer vectors. Only the **Uniform1i{v}** commands can be used to load sampler values (see below).

The **Uniform*ui{v}** commands will load \( count \) sets of one to four unsigned integer values into a uniform location defined as an unsigned integer, an unsigned integer vector, an array of unsigned integers or an array of unsigned integer vectors.

The **UniformMatrix\{234\}fv** and **UniformMatrix\{234\}dv** commands will load \( count \) \( 2 \times 2, 3 \times 3, \) or \( 4 \times 4 \) matrices (corresponding to 2, 3, or 4 in the command name) of single- or double-precision floating-point values, respectively, into a uniform location defined as a matrix or an array of matrices. If \( transpose \) is \text{FALSE}, the matrix is specified in column major order, otherwise in row major order.

The **UniformMatrix\{2x3,3x2,2x4,4x2,3x4,4x3\}fv** and **UniformMatrix\{2x3,3x2,2x4,4x2,3x4,4x3\}dv** commands will load \( count \) \( 2 \times 3, 3 \times 2, 2 \times 4, 4 \times 2, 3 \times 4, \) or \( 4 \times 3 \) matrices (corresponding to the numbers in the command name) of single- or double-precision floating-point values, respectively, into a uniform location defined as a matrix or an array of matrices. The first number in the command name is the number of columns; the second is the number of rows. For example, **UniformMatrix2x4fv** is used to load a single-precision matrix consisting of two columns and four rows. If \( transpose \) is \text{FALSE}, the matrix is specified in column major order, otherwise in row major order.

When loading values for a uniform declared as a boolean, a boolean vector, an array of booleans, or an array of boolean vectors, the **Uniform*i{v}**, **Uniform*ui{v}**, and **Uniform*f{v}** set of commands can be used to load boolean values. Type conversion is done by the GL. The uniform is set to \text{FALSE} if the input value is 0 or 0.0f, and set to \text{TRUE} otherwise. The **Uniform** command used must match the size of the uniform, as declared in the shader. For example, to load a uniform declared as a \text{bvec2}, any of the **Uniform2{if ui}** commands may be used. An \text{INVALID_OPERATION} error will be generated if an attempt is made to use a non-matching **Uniform** command. In this example using **Uniform1iv** would generate an error.

For all other uniform types, except for subroutine uniforms and atomic counters, the **Uniform** command used must match the size and type of the uniform.
2.11. VERTEX SHADERS

as declared in the shader. No type conversions are done. For example, to load a uniform declared as a \texttt{vec4}, \texttt{Uniform4f\{v\}} must be used, and to load a uniform declared as a \texttt{dmat3}, \texttt{UniformMatrix3dv} must be used. An \texttt{INVALID\_OPERATION} error will be generated if an attempt is made to use a non-matching Uniform* command.

When loading \(N\) elements starting at an arbitrary position \(k\) in a uniform declared as an array, elements \(k\) through \(k + N - 1\) in the array will be replaced with the new values. Values for any array element that exceeds the highest array element index used, as reported by \texttt{GetActiveUniform}, will be ignored by the GL.

If the value of location is -1, the Uniform* commands will silently ignore the data passed in, and the current uniform values will not be changed.

If any of the following conditions occur, an \texttt{INVALID\_OPERATION} error is generated by the Uniform* commands, and no uniform values are changed:

- if the size indicated in the name of the Uniform* command used does not match the size of the uniform declared in the shader,
- if the uniform declared in the shader is not of type boolean and the type indicated in the name of the Uniform* command used does not match the type of the uniform,
- if \(count\) is greater than one, and the uniform declared in the shader is not an array variable,
- if no variable with a location of location exists in the program object currently in use and location is not -1, or
- if there is no active program object in use.

To load values into the uniform variables of the default uniform block of a program which may not necessarily be bound, use the commands

```c
void ProgramUniform\{1234\}{ifd}(uint program,
   int location, T value);
void ProgramUniform\{1234\}{ifd}v(uint program,
   int location, sizei count, const T value);
void ProgramUniform\{1234\}ui(uint program, int location,
   T value);
void ProgramUniform\{1234\}uiv(uint program,
   int location, sizei count, T value);
```
2.11. VERTEX SHADERS

```c
void ProgramUniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}{fd}v( uint program,
   int location, sizei count, boolean transpose, const float *value );
void ProgramUniformMatrix{234}{fd}v( uint program, int location, sizei count,
   boolean transpose, const float *value );
```

These commands operate identically to the corresponding commands above without `Program` in the command name except, rather than updating the currently active program object, these `Program` commands update the program object named by the initial `program` parameter. The remaining parameters following the initial `program` parameter match the parameters for the corresponding non-`Program` uniform command.

If `program` is not the name of a created program or shader object, an `INVALID_VALUE` error is generated. If `program` identifies a shader object or a program object that has not been linked successfully, an `INVALID_OPERATION` error is generated.

**Uniform Blocks**

The values of uniforms arranged in named uniform blocks are extracted from buffer object storage. The mechanisms for placing individual uniforms in a buffer object and connecting a uniform block to an individual buffer object are described below.

There is a set of implementation-dependent maximums for the number of active uniform blocks used by each shader. If the number of uniform blocks used by any shader in the program exceeds its corresponding limit, the program will fail to link. The limits for vertex, tessellation control, tessellation evaluation, geometry, and fragment shaders can be obtained by calling `GetIntegerv` with `pname` values of `MAX_VERTEX_UNIFORM_BLOCKS`, `MAX_TESS_CONTROL_UNIFORM_BLOCKS`, `MAX_TESS_EVALUATION_UNIFORM_BLOCKS`, `MAX_GEOMETRY_UNIFORM_BLOCKS`, and `MAX_FRAGMENT_UNIFORM_BLOCKS`, respectively.

Additionally, there is an implementation-dependent limit on the sum of the number of active uniform blocks used by each shader of a program. If a uniform block is used by multiple shaders, each such use counts separately against this combined limit. The combined uniform block use limit can be obtained by calling `GetIntegerv` with a `pname` of `MAX_COMBINED_UNIFORM_BLOCKS`.

When a named uniform block is declared by multiple shaders in a program, it must be declared identically in each shader. The uniforms within the block must be declared with the same names and types, and in the same order. If a program
2.11. VERTEX SHADERS

contains multiple shaders with different declarations for the same named uniform block differs between shader, the program will fail to link.

Uniform Buffer Object Storage

When stored in buffer objects associated with uniform blocks, uniforms are represented in memory as follows:

- Members of type `bool` are extracted from a buffer object by reading a single uint-typed value at the specified offset. All non-zero values correspond to true, and zero corresponds to false.
- Members of type `int` are extracted from a buffer object by reading a single int-typed value at the specified offset.
- Members of type `uint` are extracted from a buffer object by reading a single uint-typed value at the specified offset.
- Members of type `float` are extracted from a buffer object by reading a single float-typed value at the specified offset.
- Members of type `double` are extracted from a buffer object by reading a single double-typed value at the specified offset.
- Vectors with $N$ elements with basic data types of `bool`, `int`, `uint`, `float`, or `double` are extracted as $N$ values in consecutive memory locations beginning at the specified offset, with components stored in order with the first ($X$) component at the lowest offset. The GL data type used for component extraction is derived according to the rules for scalar members above.
- Column-major matrices with $C$ columns and $R$ rows (using the type `matC x R`, or simply `matC` if $C = R$) are treated as an array of $C$ floating-point column vectors, each consisting of $R$ components. The column vectors will be stored in order, with column zero at the lowest offset. The difference in offsets between consecutive columns of the matrix will be referred to as the column stride, and is constant across the matrix. The column stride, `UNIFORM_MATRIX_STRIDE`, is an implementation-dependent value and may be queried after a program is linked.
- Row-major matrices with $C$ columns and $R$ rows (using the type `matC x R`, or simply `matC` if $C = R$) are treated as an array of $R$ floating-point row vectors, each consisting of $C$ components. The row vectors will be stored in order, with row zero at the lowest offset. The difference in offsets between
2.11. VERTEX SHADERS

consecutive rows of the matrix will be referred to as the row stride, and is constant across the matrix. The row stride, UNIFORM_MATRIX_STRIDE, is an implementation-dependent value and may be queried after a program is linked.

- Arrays of scalars, vectors, and matrices are stored in memory by element order, with array member zero at the lowest offset. The difference in offsets between each pair of elements in the array in basic machine units is referred to as the array stride, and is constant across the entire array. The array stride, UNIFORM_ARRAY_STRIDE, is an implementation-dependent value and may be queried after a program is linked.

**Standard Uniform Block Layout**

By default, uniforms contained within a uniform block are extracted from buffer storage in an implementation-dependent manner. Applications may query the offsets assigned to uniforms inside uniform blocks with query functions provided by the GL.

The layout qualifier provides shaders with control of the layout of uniforms within a uniform block. When the std140 layout is specified, the offset of each uniform in a uniform block can be derived from the definition of the uniform block by applying the set of rules described below.

If a uniform block is declared in multiple shaders linked together into a single program, the link will fail unless the uniform block declaration, including layout qualifier, are identical in all such shaders.

When using the std140 storage layout, structures will be laid out in buffer storage with its members stored in monotonically increasing order based on their location in the declaration. A structure and each structure member have a base offset and a base alignment, from which an aligned offset is computed by rounding the base offset up to a multiple of the base alignment. The base offset of the first member of a structure is taken from the aligned offset of the structure itself. The base offset of all other structure members is derived by taking the offset of the last basic machine unit consumed by the previous member and adding one. Each structure member is stored in memory at its aligned offset. The members of a top-level uniform block are laid out in buffer storage by treating the uniform block as a structure with a base offset of zero.

1. If the member is a scalar consuming $N$ basic machine units, the base alignment is $N$. 
2. If the member is a two- or four-component vector with components consuming $N$ basic machine units, the base alignment is $2N$ or $4N$, respectively.

3. If the member is a three-component vector with components consuming $N$ basic machine units, the base alignment is $4N$.

4. If the member is an array of scalars or vectors, the base alignment and array stride are set to match the base alignment of a single array element, according to rules (1), (2), and (3), and rounded up to the base alignment of a vec4. The array may have padding at the end; the base offset of the member following the array is rounded up to the next multiple of the base alignment.

5. If the member is a column-major matrix with $C$ columns and $R$ rows, the matrix is stored identically to an array of $C$ column vectors with $R$ components each, according to rule (4).

6. If the member is an array of $S$ column-major matrices with $C$ columns and $R$ rows, the matrix is stored identically to a row of $S \times C$ column vectors with $R$ components each, according to rule (4).

7. If the member is a row-major matrix with $C$ columns and $R$ rows, the matrix is stored identically to an array of $R$ row vectors with $C$ components each, according to rule (4).

8. If the member is an array of $S$ row-major matrices with $C$ columns and $R$ rows, the matrix is stored identically to a row of $S \times R$ row vectors with $C$ components each, according to rule (4).

9. If the member is a structure, the base alignment of the structure is $N$, where $N$ is the largest base alignment value of any of its members, and rounded up to the base alignment of a vec4. The individual members of this substructure are then assigned offsets by applying this set of rules recursively, where the base offset of the first member of the sub-structure is equal to the aligned offset of the structure. The structure may have padding at the end; the base offset of the member following the sub-structure is rounded up to the next multiple of the base alignment of the structure.

10. If the member is an array of $S$ structures, the $S$ elements of the array are laid out in order, according to rule (9).
2.11. VERTEX SHADERS

Uniform Buffer Object Bindings

The value an active uniform inside a named uniform block is extracted from the data store of a buffer object bound to one of an array of uniform buffer binding points. The number of binding points can be queried using `GetIntegerv` with the constant `MAX_UNIFORM_BUFFER_BINDINGS`.

Regions of buffer objects are bound as storage for uniform blocks by calling one of the commands `BindBufferRange` or `BindBufferBase` (see section 2.9.1) with `target` set to `UNIFORM_BUFFER`. In addition to the general errors described in section 2.9.1, `BindBufferRange` will generate an `INVALID_VALUE` error if `index` is greater than or equal to the value of `MAX_UNIFORM_BUFFER_BINDINGS`, or if `offset` is not a multiple of the implementation-dependent alignment requirement (the value of `UNIFORM_BUFFER_OFFSET_ALIGNMENT`).

Each of a program’s active uniform blocks has a corresponding uniform buffer object binding point. This binding point can be assigned by calling:

```c
void UniformBlockBinding( uint program, 
    uint uniformBlockIndex, uint uniformBlockBinding );
```

`program` is a name of a program object for which the command `LinkProgram` has been issued in the past.

An `INVALID_VALUE` error is generated if `uniformBlockIndex` is not an active uniform block index of `program`, or if `uniformBlockBinding` is greater than or equal to the value of `MAX_UNIFORM_BUFFER_BINDINGS`.

If successful, `UniformBlockBinding` specifies that `program` will use the data store of the buffer object bound to the binding point `uniformBlockBinding` to extract the values of the uniforms in the uniform block identified by `uniformBlockIndex`.

When executing shaders that access uniform blocks, the binding point corresponding to each active uniform block must be populated with a buffer object with a size no smaller than the minimum required size of the uniform block (the value of `UNIFORM_BLOCK_DATA_SIZE`). For binding points populated by `BindBufferRange`, the size in question is the value of the `size` parameter. If any active uniform block is not backed by a sufficiently large buffer object, the results of shader execution are undefined, and may result in GL interruption or termination. Shaders may be executed to process the primitives and vertices specified by vertex array commands (see section 2.8).

When a program object is linked or re-linked, the uniform buffer object binding point assigned to each of its active uniform blocks is reset to zero.
2.11. VERTEX SHADERS

Atomic Counter Buffers

The values of atomic counters are backed by buffer object storage. The mechanisms for accessing individual atomic counters in a buffer object and connecting to an atomic counter are described in this section.

There is a set of implementation-dependent maximums for the number of active atomic counter buffer referenced by each shader. If the number of atomic counter buffer bindings referenced by any shader in the program exceeds its corresponding limit, the program will fail to link. The limits for vertex, tessellation control, tessellation evaluation, geometry, and fragment shaders can be obtained by calling `GetIntegerv` with `pname` values of `MAX_VERTEX_ATOMIC_COUNTER_BUFFERS`, `MAX_TESS_CONTROL_ATOMIC_COUNTER_BUFFERS`, `MAX_TESS_EVALUATION_ATOMIC_COUNTER_BUFFERS`, `MAX_GEOMETRY_ATOMIC_COUNTER_BUFFERS`, or `MAX_FRAGMENT_ATOMIC_COUNTER_BUFFERS`, respectively.

Additionally, there is an implementation-dependent limit on the sum of the number of active atomic counter buffers used by each shader of a program. If an atomic counter buffer is used by multiple shaders, each such use counts separately against this combined limit. The combined atomic counter buffer use limit can be obtained by calling `GetIntegerv` with a `pname` of `MAX_ATOMIC_COUNTER_BUFFERS`.

Atomic Counter Buffer Object Storage

Atomic counters stored in buffer objects are represented in memory as follows:

- Members of type `atomic_uint` are extracted from a buffer object by reading a single `uint`-typed value at the specified offset.

- Arrays of type `atomic_uint` are stored in memory by element order, with array element member zero at the lowest offset. The difference in offsets between each pair of elements in the array in basic machine units is referred to as the array stride, and is constant across the entire array. The stride can be queried by calling `GetIntegerv` with a `pname` of `ATOMIC_COUNTER_ARRAY_STRIDE` after a program is linked.

Atomic Counter Buffer Bindings

The value of an active atomic counter is extracted from or written to the data store of a buffer object bound to one of an array of atomic counter buffer binding points.
The number of binding points can be queried by calling `GetIntegerv` with a `pname` of `MAX_ATOMIC_COUNTER_BUFFER_BINDINGS`.

Regions of buffer objects are bound as storage for atomic counters by calling one of the commands `BindBufferRange` or `BindBufferBase` (see section 2.9.1) with `target` set to `ATOMIC_COUNTER_BUFFER`. In addition to the general errors described in section 2.9.1, `BindBufferBase` and `BindBufferRange` will generate an `INVALID_VALUE` error if `index` is greater than or equal to the value of `MAX_ATOMIC_COUNTER_BUFFER_BINDINGS`, and `BindBufferRange` will generate an `INVALID_VALUE` error if `offset` is not a multiple of four.

Each of a program’s active atomic counter buffer bindings has a corresponding atomic counter buffer binding point. This binding point is established with the `layout` qualifier in the shader text, either explicitly or implicitly, as described in the OpenGL Shading Language Specification.

When executing shaders that access atomic counters, each active atomic counter buffer must be populated with a buffer object with a size no smaller than the minimum required size for that buffer (the value of `ATOMIC_COUNTER_BUFFER_DATA_SIZE`). For binding points populated by `BindBufferRange`, the size in question is the value of the `size` parameter. If any active atomic counter buffer is not backed by a sufficiently large buffer object, the results of shader execution are undefined, and may result in GL interruption or termination.

### 2.11.8 Subroutine Uniform Variables

Subroutine uniform variables are similar to uniform variables, except they are context state rather than program state. Having subroutine uniforms be context state allows them to have different values if the program is used in multiple contexts simultaneously. There is a set of subroutine uniforms for each shader stage.

The command

```c
int GetSubroutineUniformLocation( uint program,
  enum shadertype, const char *name );
```

will return the location of the subroutine uniform variable `name` in the shader stage of type `shadertype` attached to `program`, with behavior otherwise identical to `GetUniformLocation`. The value -1 will be returned if `name` is not the name of an active subroutine uniform. Active subroutine locations are assigned using consecutive integers in the range from zero to the value of `ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS` minus one for the shader stage. There is an implementation-dependent limit on the number of active subroutine uniform locations in each shader stage; a program will fail to link if the number of subroutine uniform locations required is
greater than the value of \texttt{MAX\_SUBROUTINE\_UNIFORM\_LOCATIONS}. If \texttt{program} has not been successfully linked, the error \texttt{INVALID\_OPERATION} will be generated. For active subroutine uniforms declared as arrays, the declared array elements are assigned consecutive locations.

Each function in a shader associated with a subroutine type is considered an active subroutine, unless the compiler conclusively determines that the function could never be assigned to an active subroutine uniform. Each active subroutine will be assigned an unsigned integer subroutine index that is unique to the shader stage. This index can be queried with the command

\begin{verbatim}
uint GetSubroutineIndex( uint program, enum shadertype, const char *name );
\end{verbatim}

where \texttt{name} is the null-terminated name of a function in the shader stage of type \texttt{shadertype} attached to \texttt{program}. Subroutine indices are assigned using consecutive integers in the range from zero to the value of \texttt{ACTIVE\_SUBROUTINES} minus one for the shader stage. The value \texttt{INVALID\_INDEX} will be returned if \texttt{name} is not the name of an active subroutine in the shader stage. After the program has been linked, the subroutine index will not change unless the program is re-linked.

There is an implementation-dependent limit on the number of active subroutines in each shader stage; a program will fail to link if the number of subroutines is greater than the maximum subroutine count, (the value of \texttt{MAX\_SUBROUTINES}). Information about active subroutine uniforms can be obtained by calling

\begin{verbatim}
void GetActiveSubroutineUniformiv( uint program, enum shadertype, uint index, enum pname, int *values );
void GetActiveSubroutineUniformName( uint program, enum shadertype, uint index, sizei bufsize, sizei *length, char *name );
\end{verbatim}

\texttt{program} and \texttt{shadertype} specify the program and shader stage. \texttt{index} must be an active subroutine uniform index in the range from zero to the value of \texttt{ACTIVE\_SUBROUTINE\_UNIFORMS} minus one for the shader stage. If \texttt{index} is greater than or equal to the value of \texttt{ACTIVE\_SUBROUTINE\_UNIFORMS}, the error \texttt{INVALID\_VALUE} is generated.

For \texttt{GetActiveSubroutineUniformiv}, \texttt{pname} identifies a property of the active subroutine uniform being queried. If \texttt{pname} is \texttt{NUM\_COMPATIBLE\_SUBROUTINES}, a single integer indicating the number of subroutines that can be assigned to the uniform is returned in \texttt{values}. If \texttt{pname} is \texttt{COMPATIBLE\_SUBROUTINES}, an array of integers is returned in \texttt{values}, with each integer specifying the index of an active
subroutine that can be assigned to the selected subroutine uniform. The number of integers returned is the same as the value returned for `NUM_COMPATIBLE_SUBROUTINES`. If `pname` is `UNIFORM_SIZE`, a single integer is returned in `values`. If the selected subroutine uniform is an array, the declared size of the array is returned; otherwise, one is returned. If `pname` is `UNIFORM_NAME_LENGTH`, a single integer specifying the length of the subroutine uniform name (including the terminating null character) is returned in `values`.

For `GetActiveSubroutineUniformName`, the uniform name is returned as a null-terminated string in `name`. The actual number of characters written into `name`, excluding the null terminator is returned in `length`. If `length` is `NULL`, no length is returned. The maximum number of characters that may be written into `name`, including the null terminator, is specified by `bufsize`. The length of the longest subroutine uniform name in `program` and `shadertype` is given by the value of `ACTIVE_SUBROUTINE_UNIFORM_MAX_LENGTH`, which can be queried with `GetProgramStageiv`.

The name of an active subroutine can be queried given its subroutine index with the command:

```c
void GetActiveSubroutineName( uint program,
   enum shadertype, uint index, sizei bufsize,
   sizei *length, char *name );
```

`program` and `shadertype` specify the program and shader stage. `index` must be an active subroutine index in the range from zero to the value of `ACTIVE_SUBROUTINES` minus one for the shader stage. If `index` is greater than or equal to the value of `ACTIVE_SUBROUTINES`, the error `INVALID_VALUE` is generated. The name of the selected subroutine is returned as a null-terminated string in `name`. The actual number of characters written into `name`, excluding the null terminator, is returned in `length`. If `length` is `NULL`, no length is returned. The maximum number of characters that may be written into `name`, including the null terminator, is specified by `bufsize`. The length of the longest subroutine name in `program` and `shadertype` is given by the value of `ACTIVE_SUBROUTINE_MAX_LENGTH`, which can be queried with `GetProgramStageiv`.

The command

```c
void UniformSubroutinesuiv( enum shadertype, sizei count,
   const uint *indices );
```

will load all active subroutine uniforms for shader stage `shadertype` with subroutine indices from `indices`, storing `indices[i]` into the uniform at location `i`. If `count` is
2.11. VERTEX SHADERS

not equal to the value of ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS for the program currently in use at shader stage shadertype, or if any value in indices is greater than or equal to the value of ACTIVE_SUBROUTINES for the shader stage, the error INVALID_VALUE is generated. If, for any subroutine index being loaded to a particular uniform location, the function corresponding to the subroutine index was not associated (as defined in section 6.1.2 of the OpenGL Shading Language Specification) with the type of the subroutine variable at that location, then the error INVALID_OPERATION is generated. If no program is active, the error INVALID_OPERATION is generated. If no program is active, the error INVALID_OPERATION is generated.

Each subroutine uniform must have at least one subroutine to assign to the uniform. A program will fail to link if any stage has one or more subroutine uniforms that has no subroutine associated with the subroutine type of the uniform.

When the active program for a shader stage is re-linked or changed by a call to UseProgram, BindProgramPipeline, or UseProgramStages, subroutine uniforms for that stage are reset to arbitrarily chosen default functions with compatible subroutine types.

2.11.9 Samplers

Samplers are special uniforms used in the OpenGL Shading Language to identify the texture object used for each texture lookup. The value of a sampler indicates the texture image unit being accessed. Setting a sampler’s value to i selects texture image unit number i. The values of i ranges from zero to the implementation-dependent maximum supported number of texture image units.

The type of the sampler identifies the target on the texture image unit. The texture object bound to that texture image unit’s target is then used for the texture lookup. For example, a variable of type sampler2D selects target TEXTURE_2D on its texture image unit. Binding of texture objects to targets is done as usual with BindTexture. Selecting the texture image unit to bind to is done as usual with ActiveTexture.

The location of a sampler needs to be queried with GetUniformLocation, just like any uniform variable. Sampler values need to be set by calling Uniform1i{v}. Loading samplers with any of the other Uniform* entry points is not allowed and will result in an INVALID_OPERATION error.

It is not allowed to have variables of different sampler types pointing to the same texture image unit within a program object. This situation can only be detected at the next rendering command issued, and an INVALID_OPERATION error will then be generated.

Active samplers are samplers actually being used in a program object. The LinkProgram command determines if a sampler is active or not. The LinkPro-
gram command will attempt to determine if the active samplers in the shader(s) contained in the program object exceed the maximum allowable limits. If it determines that the count of active samplers exceeds the allowable limits, then the link fails (these limits can be different for different types of shaders). Each active sampler variable counts against the limit, even if multiple samplers refer to the same texture image unit.

2.11.10 Images

*Images* are special uniforms used in the OpenGL Shading Language to identify a level of a texture to be read or written using built-in image load, store, and atomic functions in the manner described in section 3.9.20. The value of an image uniform is an integer specifying the image unit accessed. Image units are numbered beginning at zero, and there is an implementation-dependent number of available image units (*MAX_IMAGE_UNITS*). The error *INVALID_VALUE* is generated if Uniform1i{v} is used to set an image uniform to a value less than zero or greater than or equal to the value of *MAX_IMAGE_UNITS*. Note that image units used for image variables are independent of the texture image units used for sampler variables; the number of units provided by the implementation may differ. Textures are bound independently and separately to image and texture image units.

The type of an image variable must match the texture target of the image currently bound to the image unit, otherwise the result of a load, store, or atomic operation is undefined (see section 4.1.7.2 of the OpenGL Shading Language specification for more detail).

The location of an image variable needs to be queried with GetUniformLocation, just like any uniform variable. Image values need to be set by calling Uniform1i{v}. Loading image variables with any of the other Uniform entry points is not allowed and will generate an INVALID_OPERATION error.

Unlike samplers, there is no limit on the number of active image variables that may be used by a program or by any particular shader. However, given that there is an implementation-dependent limit on the number of unique image units, the actual number of images that may be used by all shaders in a program is limited.

2.11.11 Output Variables

A vertex shader may define one or more *output variables* or *outputs* (see the OpenGL Shading Language Specification).

The OpenGL Shading Language Specification also defines a set of built-in outputs that vertex shaders can write to (see sections 7.1 and 7.6 of the OpenGL Shading Language Specification). These output variables are either used as the
mechanism to communicate values to the next active stage in the vertex processing pipeline: either the tessellation control shader, the tessellation evaluation shader, the geometry shader, or the fixed-function vertex processing stages leading to rasterization.

If the output variables are passed directly to the vertex processing stages leading to rasterization, the values of all outputs are expected to be interpolated across the primitive being rendered, unless flatshaded. Otherwise the values of all outputs are collected by the primitive assembly stage and passed on to the subsequent pipeline stage once enough data for one primitive has been collected.

The number of components (individual scalar numeric values) of output variables that can be written by the vertex shader, whether or not a tessellation control, tessellation evaluation, or geometry shader is active, is given by the value of the implementation-dependent constant `MAX_VERTEX_OUTPUT_COMPONENTS`. Outputs declared as vectors, matrices, and arrays will all consume multiple components. For the purposes of counting input and output components consumed by a shader, variables declared as vectors, matrices, and arrays will all consume multiple components. Each component of variables declared as double-precision floating-point scalars, vectors, or matrices may be counted as consuming two components.

When a program is linked, all components of any outputs written by a vertex shader will count against this limit. A program whose vertex shader writes more than the value of `MAX_VERTEX_OUTPUT_COMPONENTS` components worth of outputs may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Additionally, when linking a program containing only a vertex and fragment shader, there is a limit on the total number of components used as vertex shader outputs or fragment shader inputs. This limit is given by the value of the implementation-dependent constant `MAX_VARYING_COMPONENTS`. The implementation-dependent constant `MAX_VARYING_VECTORS` has a value equal to the value of `MAX_VARYING_COMPONENTS` divided by four. Each output variable component used as either a vertex shader output or fragment shader input counts against this limit, except for the components of `gl_Position`. A program containing only a vertex and fragment shader that accesses more than this limit’s worth of components of outputs may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Each program object can specify a set of output variables from one shader to be recorded in transform feedback mode (see section 2.17). The variables that can be recorded are those emitted by the first active shader, in order, from the following list:
2.11. VERTEX SHADERS

- geometry shader
- tessellation evaluation shader
- tessellation control shader
- vertex shader

The values to record are specified with the command

```c
void TransformFeedbackVaryings(uint program, sizei count, const char *const *varyings, enum bufferMode);
```

- `program` specifies the program object. `count` specifies the number of output variables used for transform feedback. `varyings` is an array of `count` zero-terminated strings specifying the names of the outputs to use for transform feedback. The variables specified in `varyings` can be either built-in (beginning with "gl_") or user-defined variables. Output variables are written out in the order they appear in the array `varyings`. `bufferMode` is either `INTERLEAVED_ATTRIBS` or `SEPARATE_ATTRIBS`, and identifies the mode used to capture the outputs when transform feedback is active. The error `INVALID_VALUE` is generated if `bufferMode` is `SEPARATE_ATTRIBS` and `count` is greater than the value of the implementation-dependent limit `MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`.

If a string in `varyings` is `gl_NextBuffer`, it does not identify an output, but instead serves as a buffer separator value to direct subsequent outputs at the next transform feedback binding point. If a string in `varyings` is `gl_SkipComponents1`, `gl_SkipComponents2`, `gl_SkipComponents3`, or `gl_SkipComponents4`, it also does not identify a specific output. Instead, such values are treated as requesting that the GL skip the next one to four components of output data. Skipping components this way is equivalent to specifying a one- to four-component output with undefined values, except that the corresponding memory in the buffer object is not modified. Such array entries are counted as being written to the buffer object for the purposes of determining whether the requested attributes exceed per-buffer component count limits and whether recording a new primitive would result in an overflow. Each component skipped is considered to occupy a single float.

The error `INVALID_OPERATION` is generated if any pointer in `varyings` identifies the special names `gl_NextBuffer`, `gl_SkipComponents1`, `gl_SkipComponents2`, `gl_SkipComponents3`, or `gl_SkipComponents4` and...
bufferMode is not INTERLEAVED_ATTRIBS, or if the number of gl_NextBuffer pointers in varyings is greater than or equal to the limit MAX_TRANSFORM_FEEDBACK_BUFFERS.

The state set by TransformFeedbackVaryings has no effect on the execution of the program until program is subsequently linked. When LinkProgram is called, the program is linked so that the values of the specified outputs for the vertices of each primitive generated by the GL are written to a single buffer object (if the buffer mode is INTERLEAVED_ATTRIBS) or multiple buffer objects (if the buffer mode is SEPARATE_ATTRIBS). A program will fail to link if:

- the count specified by TransformFeedbackVaryings is non-zero, but the program object has no vertex, tessellation control, tessellation evaluation, or geometry shader;
- any variable name specified in the varyings array is not one of gl_NextBuffer, gl_SkipComponents1, gl_SkipComponents2, gl_SkipComponents3, or gl_SkipComponents4, and is not declared as a built-in or user-defined output variable in the shader stage whose outputs can be recorded.
- any two entries in the varyings array specify the same output variable;
- the total number of components to capture in any output in varyings is greater than the value of MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS and the buffer mode is SEPARATE_ATTRIBS;
- the total number of components to capture is greater than the constant MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS and the buffer mode is INTERLEAVED_ATTRIBS;
- the set of outputs to capture to any single binding point includes outputs from more than one vertex stream.

For the purposes of counting the total number of components to capture, each component of outputs declared as double-precision floating-point scalars, vectors, or matrices may be counted as consuming two components.

To determine the set of output variables in a linked program object that will be captured in transform feedback mode, the command:

```c
void GetTransformFeedbackVarying( uint program,
    uint index, sizei bufSize, sizei *length, sizei *size,
    enum *type, char *name );
``
provides information about the output variable selected by \textit{index}. An \textit{index} of 0 selects the first output specified in the \textit{varyings} array of \texttt{TransformFeedbackVaryings}, and an \textit{index} of \texttt{TRANSFORM_FEEDBACK_VARYINGS-1} selects the last such output. The value of \texttt{TRANSFORM_FEEDBACK_VARYINGS} can be queried with \texttt{GetProgramiv} (see section 6.1.12). If \textit{index} is greater than or equal to \texttt{TRANSFORM_FEEDBACK_VARYINGS}, the error \texttt{INVALID_VALUE} is generated. The parameter \textit{program} is the name of a program object for which the command \texttt{LinkProgram} has been issued in the past. If \textit{program} has not been linked, the error \texttt{INVALID_OPERATION} is generated. If a new set of outputs is specified by \texttt{TransformFeedbackVaryings} after a program object has been linked, the information returned by \texttt{GetTransformFeedbackVarying} will not reflect those variables until the program is re-linked.

The name of the selected output is returned as a null-terminated string in \textit{name}. The actual number of characters written into \textit{name}, excluding the null terminator, is returned in \textit{length}. If \textit{length} is NULL, no length is returned. The maximum number of characters that may be written into \textit{name}, including the null terminator, is specified by \textit{bufSize}. The returned output name can be the name of either a built-in (beginning with "gl_") or user-defined output variable. See the OpenGL Shading Language Specification for a complete list. The length of the longest output name in \textit{program} is given by \texttt{TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH}, which can be queried with \texttt{GetProgramiv} (see section 6.1.12).

The type of the selected output is returned into \textit{type}. The size of the output is returned into \textit{size}. The value in \textit{size} is in units of the type returned in \textit{type}. The type returned can be any of the types whose “Xfb” column is checked in table 2.13. If an error occurred, the return parameters \textit{length}, \textit{size}, \textit{type} and \textit{name} will be unmodified. This command will return as much information about the outputs as possible. If no information is available, \textit{length} will be set to zero and \textit{name} will be an empty string. This situation could arise if \texttt{GetTransformFeedbackVarying} is called after a failed link.

Special output names (e.g., \texttt{gl\_NextBuffer}, \texttt{gl\_SkipComponents1}) passed to \texttt{TransformFeedbackVaryings} in the \textit{varyings} array are counted as outputs to be recorded for the purposes of determining the value of \texttt{TRANSFORM_FEEDBACK_VARYINGS} and for determining the variable selected by \textit{index} in \texttt{GetTransformFeedbackVarying}. If \textit{index} identifies \texttt{gl\_NextBuffer}, the values zero and \texttt{NONE} will be written to \textit{size} and \textit{type}, respectively. If \textit{index} is of the form \texttt{gl\_SkipComponents\textit{n}}, the value \texttt{NONE} will be written to \textit{type} and the number of components \textit{n} will be written to \textit{size}.  

OpenGL 4.2 (Core Profile) - April 27, 2012
2.11. VERTEX SHADERS

2.11.12 Shader Execution

If there is an active program object present for the vertex, tessellation control, tessellation evaluation, or geometry shader stages, the executable code for these active programs is used to process incoming vertex values. The following sequence of operations is performed:

- Vertices are processed by the vertex shader (see section 2.11) and assembled into primitives as described in sections 2.5 through 2.8.

- If the current program contains a tessellation control shader, each individual patch primitive is processed by the tessellation control shader (section 2.12.1). Otherwise, primitives are passed through unmodified. If active, the tessellation control shader consumes its input patch and produces a new patch primitive, which is passed to subsequent pipeline stages.

- If the current program contains a tessellation evaluation shader, each individual patch primitive is processed by the tessellation primitive generator (section 2.12.2) and tessellation evaluation shader (see section 2.12.3). Otherwise, primitives are passed through unmodified. When a tessellation evaluation shader is active, the tessellation primitive generator produces a new collection of point, line, or triangle primitives to be passed to subsequent pipeline stages. The vertices of these primitives are processed by the tessellation evaluation shader. The patch primitive passed to the tessellation primitive generator is consumed by this process.

- If the current program contains a geometry shader, each individual primitive is processed by the geometry shader (section 2.13). Otherwise, primitives are passed through unmodified. If active, the geometry shader consumes its input patch. However, each geometry shader invocation may emit new vertices, which are arranged into primitives and passed to subsequent pipeline stages.

The following fixed-function operations are the applied to vertices of the resulting primitives:

- Transform feedback (section 2.17).
- Flatshading (section 2.19).
- Clipping, including client-defined half-spaces (section 2.20).
- Perspective division on clip coordinates (section 2.14).
2.11. VERTEX SHADERS

- Viewport mapping, including depth range scaling (section 2.14.1).
- Front face determination (section 3.6.1).
- Generic attribute clipping (section 2.20.1).
- Rasterization (chapter 3).

There are several special considerations for vertex shader execution described in the following sections.

Shader Only Texturing

This section describes texture functionality that is accessible through vertex, tessellation control, tessellation evaluation, geometry, or fragment shaders. Also refer to section 3.9 and to section 8.7 of the OpenGL Shading Language Specification.

Texel Fetches

The OpenGL Shading Language texel fetch functions provide the ability to extract a single texel from a specified texture image. The integer coordinates passed to the texel fetch functions are used as the texel coordinates \((i, j, k)\) into the texture image. This in turn means the texture image is point-sampled (no filtering is performed), but the remaining steps of texture access (described below) are still applied.

The level of detail accessed is computed by adding the specified level-of-detail parameter \(\text{lod}\) to the base level of the texture, \(\text{level}_{\text{base}}\).

The texel fetch functions can not perform depth comparisons or access cube maps. Unlike filtered texel accesses, texel fetches do not support LOD clamping or any texture wrap mode, and require a mipmapped minification filter to access any level of detail other than the base level.

The results of the texel fetch are undefined if any of the following conditions hold:

- the computed level of detail is less than the texture’s base level \((\text{level}_{\text{base}})\) or greater than the maximum defined level, \(q\) (see section 3.8)
- the computed level of detail is not the texture’s base level and the texture’s minification filter is \text{NEAREST} or \text{LINEAR}
- the layer specified for array textures is negative or greater than the number of layers in the array texture,
• the texel coordinates \((i,j,k)\) refer to a texel outside the defined extents of the specified level of detail, where any of

\[
\begin{align*}
  i &< 0 & i &\geq w_s \\
  j &< 0 & j &\geq h_s \\
  k &< 0 & k &\geq d_s
\end{align*}
\]

and the size parameters \(w_s, h_s,\) and \(d_s\) refer to the width, height, and depth of the image, as in equation 3.16

• the texture being accessed is not complete, as defined in section 3.9.14.

Multisample Texel Fetches

Multisample buffers do not have mipmaps, and there is no level of detail parameter for multisample texel fetches. Instead, an integer parameter selects the sample number to be fetched from the buffer. The number identifying the sample is the same as the value used to query the sample location using \texttt{GetMultisamplefv}.

Multisample textures support only \texttt{NEAREST} filtering.

Additionally, this fetch may only be performed on a multisample texture sampler. No other sample or fetch commands may be performed on a multisample texture sampler.

Texture Size Query

The OpenGL Shading Language texture size functions provide the ability to query the size of a texture image. The LOD value \(\texttt{lod}\) passed in as an argument to the texture size functions is added to the \(\texttt{level}_{\text{base}}\) of the texture to determine a texture image level. The dimensions of that image level, excluding a possible border, are then returned. If the computed texture image level is outside the range \([\texttt{level}_{\text{base}}, q]\), the results are undefined. When querying the size of an array texture, both the dimensions and the layer index are returned.

Texture Access

Shaders have the ability to do a lookup into a texture map. The maximum number of texture image units available to shaders are the values of the implementation-dependent constants

• \texttt{MAX_VERTEX_TEXTURE_IMAGE_UNITS} (for vertex shaders),
2.11. VERTEX SHADERS

- **MAX_TESS_CONTROL_TEXTURE_IMAGE_UNITS** (for tessellation control shaders),
- **MAX_TESS_EVALUATION_TEXTURE_IMAGE_UNITS** (for tessellation evaluation shaders),
- **MAX_GEOMETRY_TEXTURE_IMAGE_UNITS** (for geometry shaders), and
- **MAX_TEXTURE_IMAGE_UNITS** (for fragment shaders).

All active shaders combined cannot use more than the value of **MAX_COMBINED_TEXTURE_IMAGE_UNITS** texture image units. If more than one pipeline stage accesses the same texture image unit, each such access counts separately against the **MAX_COMBINED_TEXTURE_IMAGE_UNITS** limit.

When a texture lookup is performed in a shader, the filtered texture value $\tau$ is computed in the manner described in sections 3.9.11 and 3.9.12, and converted to a texture base color $C_b$ as shown in table 3.23, followed by application of the texture swizzle as described in section 3.10.2 to compute the texture source color $C_s$ and $A_s$.

The resulting four-component vector $(R_s, G_s, B_s, A_s)$ is returned to the shader. Texture lookup functions (see section 8.7 of the OpenGL Shading Language Specification) may return floating-point, signed, or unsigned integer values depending on the function and the internal format of the texture.

In shaders other than fragment shaders, it is not possible to perform automatic level-of-detail calculations using partial derivatives of the texture coordinates with respect to window coordinates as described in section 3.9.11. Hence, there is no automatic selection of an image array level. Minification or magnification of a texture map is controlled by a level-of-detail value optionally passed as an argument in the texture lookup functions. If the texture lookup function supplies an explicit level-of-detail value $l$, then the pre-bias level-of-detail value $\lambda_{\text{base}}(x, y) = l$ (replacing equation 3.17). If the texture lookup function does not supply an explicit level-of-detail value, then $\lambda_{\text{base}}(x, y) = 0$. The scale factor $\rho(x, y)$ and its approximation function $f(x, y)$ (see equation 3.21) are ignored.

Texture lookups involving textures with depth component data generate a texture base color $C_b$ either using depth data directly or by performing a comparison with the $D_{\text{ref}}$ value used to perform the lookup, as described in section 3.9.17, and expanding the resulting value $R_t$ to a color $C_b = (R_t, 0, 0, 1)$. Swizzling is then performed as described above, but only the first component $C_s[0]$ is returned to the shader. The comparison operation is requested in the shader by using any of the shadow sampler types (sampler*Shadow), and in the texture using the
TEXTURE_COMPARE_MODE parameter. These requests must be consistent; the results of a texture lookup are undefined if any of the following conditions are true:

- The sampler used in a texture lookup function is not one of the shadow sampler types, the texture object’s internal format is `DEPTH_COMPONENT` or `DEPTH_STENCIL`, and the TEXTURE_COMPARE_MODE is not `NONE`.

- The sampler used in a texture lookup function is one of the shadow sampler types, the texture object’s internal format is `DEPTH_COMPONENT` or `DEPTH_STENCIL`, and the TEXTURE_COMPARE_MODE is `NONE`.

- The sampler used in a texture lookup function is one of the shadow sampler types, and the texture object’s internal format is not `DEPTH_COMPONENT` or `DEPTH_STENCIL`.

The stencil index texture internal component is ignored if the base internal format is `DEPTH_STENCIL`.

If a sampler is used in a shader and the sampler’s associated texture is not complete, as defined in section 3.9.14, (0, 0, 0, 1) will be returned for a non-shadow sampler and 0 for a shadow sampler.

**Atomic Counter Access**

Shaders have the ability to set and get atomic counters. The maximum number of atomic counters available to shaders are the values of the implementation dependent constants

- `MAX_VERTEX_ATOMIC_COUNTERS` (for vertex shaders),
- `MAX_TESS_CONTROL_ATOMIC_COUNTERS` (for tessellation control shaders),
- `MAX_TESS_EVALUATION_ATOMIC_COUNTERS` (for tessellation evaluation shaders),
- `MAX_GEOMETRY_ATOMIC_COUNTERS` (for geometry shaders), and
- `MAX_FRAGMENT_ATOMIC_COUNTERS` (for fragment shaders).

All active shaders combined cannot use more than the value of `MAX_COMBINED_ATOMIC_COUNTERS` atomic counters. If more than one pipeline stage accesses the same atomic counter, each such access counts separately against the `MAX_COMBINED_ATOMIC_COUNTERS` limit.
2.11. VERTEX SHADERS

Image Access

Shaders have the ability to read and write to textures using image uniforms. The maximum number of image uniforms available to individual shader stages are the values of the implementation dependent constants:

- `MAX_VERTEX_IMAGE_UNIFORMS` (vertex shaders),
- `MAX_TESS_CONTROL_IMAGE_UNIFORMS` (tessellation control shaders),
- `MAX_TESS_EVALUATION_IMAGE_UNIFORMS` (tessellation evaluation shaders),
- `MAX_GEOMETRY_IMAGE_UNIFORMS` (geometry shaders), and
- `MAX_FRAGMENT_IMAGE_UNIFORMS` (fragment shaders).

All active shaders combined cannot use more than the value of `MAX_COMBINED_IMAGE_UNIFORMS` atomic counters. If more than one shader stage accesses the same image uniform, each such access counts separately against the `MAX_COMBINED_IMAGE_UNIFORMS` limit.

Shader Inputs

Besides having access to vertex attributes and uniform variables, vertex shaders can access the read-only built-in variables `gl_VertexID` and `gl_InstanceID`.

- `gl_VertexID` holds the integer index $i$ implicitly passed by `DrawArrays` or one of the other drawing commands defined in section 2.8.3.
- `gl_InstanceID` holds the integer instance number of the current primitive in an instanced draw call (see section 2.8.3).

Section 7.1 of the OpenGL Shading Language Specification also describes these variables.

Shader Outputs

A vertex shader can write to user-defined output variables. These values are expected to be interpolated across the primitive it outputs, unless they are specified to be flat shaded. Refer to sections 4.3.6, 7.1, and 7.6 of the OpenGL Shading Language Specification for more detail.

The built-in output `gl_Position` is intended to hold the homogeneous vertex position. Writing `gl_Position` is optional.
The built-in output variable \texttt{gl\_ClipDistance} holds the clip distance(s) used in the clipping stage, as described in section 2.20. If clipping is enabled, \texttt{gl\_ClipDistance} should be written.

The built-in output \texttt{gl\_PointSize}, if written, holds the size of the point to be rasterized, measured in pixels.

Validation

It is not always possible to determine at link time if a program object can execute successfully, given that \texttt{LinkProgram} can not know the state of the remainder of the pipeline. Therefore validation is done when the first rendering command is issued, to determine if the set of active program objects can be executed. If the current set of active program objects cannot be executed, no primitives are processed and the error \texttt{INVALID\_OPERATION} will be generated.

This error is generated by any command that transfers vertices to the GL if:

- A program object is active for at least one, but not all of the shader stages that were present when the program was linked.

- One program object is active for at least two shader stages and a second program is active for a shader stage between two stages for which the first program was active.

- There is an active program for tessellation control, tessellation evaluation, or geometry stages with corresponding executable shader, but there is no active program with executable vertex shader.

- There is no current unified program object and the current program pipeline object includes a program object that was relinked since being applied to the pipeline object via \texttt{UseProgramStages} with the \texttt{PROGRAM\_SEPARABLE} parameter set to \texttt{FALSE}.

- Any two active samplers in the current program object are of different types, but refer to the same texture image unit.

- The number of active samplers in the program exceeds the maximum number of texture image units allowed.

The \texttt{INVALID\_OPERATION} error reported by these rendering commands may not provide enough information to find out why the currently active program object would not execute. No information at all is available about a program object that would still execute, but is inefficient or suboptimal given the current GL state. As a development aid, use the command
void ValidateProgram( uint program );

to validate the program object program against the current GL state. Each program
object has a boolean status, VALIDATE_STATUS, that is modified as a result of
validation. This status can be queried with GetProgramiv (see section 6.1.12).
If validation succeeded this status will be set to TRUE, otherwise it will be set to
FALSE. If validation succeeded the program object is guaranteed to execute, given
the current GL state. If validation failed, the program object is guaranteed to not
execute, given the current GL state.

ValidateProgram will check for all the conditions that could lead to an
INVALID_OPERATION error when rendering commands are issued, and may check
for other conditions as well. For example, it could give a hint on how to optimize
some piece of shader code. The information log of program is overwritten with
information on the results of the validation, which could be an empty string. The
results written to the information log are typically only useful during application
development; an application should not expect different GL implementations to
produce identical information.

A shader should not fail to compile, and a program object should not fail to
link due to lack of instruction space or lack of temporary variables. Implementa-
tions should ensure that all valid shaders and program objects may be successfully
compiled, linked and executed.

Separable program objects may have validation failures that cannot be detected
without the complete program pipeline. Mismatched interfaces, improper usage
of program objects together, and the same state-dependent failures can result in
validation errors for such program objects. As a development aid, use the command

void ValidateProgramPipeline( uint pipeline );

to validate the program pipeline object pipeline against the current GL state. Each
program pipeline object has a boolean status, VALIDATE_STATUS, that is modified
as a result of validation. This status can be queried with GetProgramPipelineiv
(see section 6.1.12). If validation succeeded, the program pipeline object is guar-
anteed to execute given the current GL state.

If pipeline is a name that has been generated (without subsequent deletion)
by GenProgramPipelines, but refers to a program pipeline object that has not
been previously bound, the GL first creates a new state vector in the same man-
ner as when BindProgramPipeline creates a new program pipeline object. If
pipeline is not a name returned from a previous call to GenProgramPipelines or if
such a name has since been deleted by DeleteProgramPipelines, an INVALID_-OPERATION error is generated.
2.11. VERTEX SHADERS

Undefined Behavior

When using array or matrix variables in a shader, it is possible to access a variable with an index computed at run time that is outside the declared extent of the variable. Such out-of-bounds accesses have undefined behavior, and system errors (possibly including program termination) may occur. The level of protection provided against such errors in the shader is implementation-dependent.

2.11.13 Shader Memory Access

Shaders may perform random-access reads and writes to texture or buffer object memory using built-in image load, store, and atomic functions, as described in the OpenGL Shading Language Specification. The ability to perform such random-access reads and writes in systems that may be highly pipelined results in ordering and synchronization issues discussed in the sections below.

Shader Memory Access Ordering

The order in which texture or buffer object memory is read or written by shaders is largely undefined. For some shader types (vertex, tessellation evaluation, and in some cases, fragment), even the number of shader invocations that might perform loads and stores is undefined. In particular, the following rules apply:

- While a vertex or tessellation evaluation shader will be executed at least once for each unique vertex specified by the application (vertex shaders) or generated by the tessellation primitive generator (tessellation evaluation shaders), it may be executed more than once for implementation-dependent reasons. Additionally, if the same vertex is specified multiple times in a collection of primitives (e.g., repeating an index in `DrawElements`), the vertex shader might be run only once.

- For each fragment generated by the GL, the number of fragment shader invocations depends on a number of factors. If the fragment fails the pixel ownership test (see section 4.1.1), the fragment shader may not be executed. Otherwise, if the framebuffer has no multisample buffer (the value of `SAMPLE_BUFFERS` is zero), the fragment shader will be invoked exactly once. If the fragment shader specifies per-sample shading, the fragment shader will be run once per covered sample. Otherwise, the number of fragment shader invocations is undefined, but must be in the range \([1, N]\), where \(N\) is the number of samples covered by the fragment.
If a fragment shader is invoked to process fragments or samples not covered by a primitive being rasterized to facilitate the approximation of derivatives for texture lookups, stores and atomics have no effect.

The relative order of invocations of the same shader type are undefined. A store issued by a shader when working on primitive B might complete prior to a store for primitive A, even if primitive A is specified prior to primitive B. This applies even to fragment shaders; while fragment shader outputs are written to the framebuffer in primitive order, stores executed by fragment shader invocations are not.

The relative order of invocations of different shader types is largely undefined. However, when executing a shader whose inputs are generated from a previous programmable stage, the shader invocations from the previous stage are guaranteed to have executed far enough to generate final values for all next-stage inputs. That implies shader completion for all stages except geometry; geometry shaders are guaranteed only to have executed far enough to emit all needed vertices.

Stores issued to different memory locations within a single shader invocation may not be visible to other invocations in the order they were performed. The built-in function `memoryBarrier()` may be used to provide stronger ordering of reads and writes performed by a single invocation. Calling `memoryBarrier()` guarantees that any memory transactions issued by the shader invocation prior to the call complete prior to the memory transactions issued after the call. Memory barriers may be needed for algorithms that require multiple invocations to access the same memory and require the operations need to be performed in a partially-defined relative order. For example, if one shader invocation does a series of writes, followed by a `memoryBarrier()` call, followed by another write, then another invocation that sees the results of the final write will also see the previous writes. Without the memory barrier, the final write may be visible before the previous writes.

The built-in atomic memory transaction functions may be used to read and write a given memory address atomically. While built-in atomic functions issued
by multiple shader invocations are executed in undefined order relative to each other, these functions perform both a read and a write of a memory address and guarantee that no other memory transaction will write to the underlying memory between the read and write. Atomics allow shaders to use shared global addresses for mutual exclusion or as counters, among other uses.

**Shader Memory Access Synchronization**

Data written to textures or buffer objects by a shader invocation may eventually be read by other shader invocations, sourced by other fixed pipeline stages, or read back by the application. When applications write to buffer objects or textures using API commands such as `TexSubImage*` or `BufferSubData`, the GL implementation knows when and where writes occur and can perform implicit synchronization to ensure that operations requested before the update see the original data and that subsequent operations see the modified data. Without logic to track the target address of each shader instruction performing a store, automatic synchronization of stores performed by a shader invocation would require the GL implementation to make worst-case assumptions at significant performance cost. To permit cases where textures or buffers may be read or written in different pipeline stages without the overhead of automatic synchronization, buffer object and texture stores performed by shaders are not automatically synchronized with other GL operations using the same memory.

Explicit synchronization is required to ensure that the effects of buffer and texture data stores performed by shaders will be visible to subsequent operations using the same objects and will not overwrite data still to be read by previously requested operations. Without manual synchronization, shader stores for a “new” primitive may complete before processing of an “old” primitive completes. Additionally, stores for an “old” primitive might not be completed before processing of a “new” primitive starts. The command

```c
void MemoryBarrier(bitfield barriers);
```

defines a barrier ordering the memory transactions issued prior to the command relative to those issued after the barrier. For the purposes of this ordering, memory transactions performed by shaders are considered to be issued by the rendering command that triggered the execution of the shader. `barriers` is a bitfield indicating the set of operations that are synchronized with shader stores; the bits used in `barriers` are as follows:

- **VERTEX_ATTRIB_ARRAY_BARRIER_BIT**: If set, vertex data sourced from buffer objects after the barrier will reflect data written by shaders prior to the
2.11. VERTEX SHADERS

barrier. The set of buffer objects affected by this bit is derived from the buffer object bindings used for arrays of generic vertex attributes (VERTEX_ATTRIB_ARRAY_BUFFER bindings).

- **ELEMENT_ARRAY_BARRIER_BIT**: If set, vertex array indices sourced from buffer objects after the barrier will reflect data written by shaders prior to the barrier. The buffer objects affected by this bit are derived from the ELEMENT_ARRAY_BUFFER binding.

- **UNIFORM_BARRIER_BIT**: Shader uniforms sourced from buffer objects after the barrier will reflect data written by shaders prior to the barrier.

- **TEXTURE_FETCH_BARRIER_BIT**: Texture fetches from shaders, including fetches from buffer object memory via buffer textures, after the barrier will reflect data written by shaders prior to the barrier.

- **SHADER_IMAGE_ACCESS_BARRIER_BIT**: Memory accesses using shader built-in image load, store, and atomic functions issued after the barrier will reflect data written by shaders prior to the barrier. Additionally, image stores and atomics issued after the barrier will not execute until all memory accesses (e.g., loads, stores, texture fetches, vertex fetches) initiated prior to the barrier complete.

- **COMMAND_BARRIER_BIT**: Command data sourced from buffer objects by Draw*Indirect commands after the barrier will reflect data written by shaders prior to the barrier. The buffer objects affected by this bit are derived from the DRAW_INDIRECT_BUFFER binding.

- **PIXEL_BUFFER_BARRIER_BIT**: Reads/writes of buffer objects via the PIXEL_PACK_BUFFER and PIXEL_UNPACK_BUFFER bindings (ReadPixels, TexSubImage, etc.) after the barrier will reflect data written by shaders prior to the barrier. Additionally, buffer object writes issued after the barrier will wait on the completion of all shader writes initiated prior to the barrier.

- **TEXTURE_UPDATE_BARRIER_BIT**: Writes to a texture via Tex(Sub)Image*, CopyTex* CompressedTex*, and reads via GetTexImage after the barrier will reflect data written by shaders prior to the barrier. Additionally, texture writes from these commands issued after the barrier will not execute until all shader writes initiated prior to the barrier complete.

- **BUFFER_UPDATE_BARRIER_BIT**: Reads/writes via Buffer*Data, CopyBufferSubData, and GetBufferSubData, or to buffer object memory.
mapped by \texttt{MapBuffer*} after the barrier will reflect data written by shaders prior to the barrier. Additionally, writes via these commands issued after the barrier will wait on the completion of any shader writes to the same memory initiated prior to the barrier.

- \texttt{FRAMEBUFFER\_BARRIER\_BIT}: Reads and writes via framebuffer object attachments after the barrier will reflect data written by shaders prior to the barrier. Additionally, framebuffer writes issued after the barrier will wait on the completion of all shader writes issued prior to the barrier.

- \texttt{TRANSFORM\_FEEDBACK\_BARRIER\_BIT}: Writes via transform feedback bindings after the barrier will reflect data written by shaders prior to the barrier. Additionally, transform feedback writes issued after the barrier will wait on the completion of all shader writes issued prior to the barrier.

- \texttt{ATOMIC\_COUNTER\_BARRIER\_BIT}: Accesses to atomic counters after the barrier will reflect writes prior to the barrier.

If \texttt{barriers} is \texttt{ALL\_BARRIER\_BITS}, shader memory accesses will be synchronized relative to all the operations described above.

Implementations may cache buffer object and texture image memory that could be written by shaders in multiple caches; for example, there may be separate caches for texture, vertex fetching, and one or more caches for shader memory accesses. Implementations are not required to keep these caches coherent with shader memory writes. Stores issued by one invocation may not be immediately observable by other pipeline stages or other shader invocations because the value stored may remain in a cache local to the processor executing the store, or because data overwritten by the store is still in a cache elsewhere in the system. When \texttt{MemoryBarrier} is called, the GL flushes and/or invalidates any caches relevant to the operations specified by the \texttt{barriers} parameter to ensure consistent ordering of operations across the barrier.

To allow for independent shader invocations to communicate by reads and writes to a common memory address, image variables in the OpenGL Shading Language may be declared as \texttt{coherent}. Buffer object or texture image memory accessed through such variables may be cached only if caches are automatically updated due to stores issued by any other shader invocation. If the same address is accessed using both coherent and non-coherent variables, the accesses using variables declared as coherent will observe the results stored using coherent variables in other invocations. Using variables declared as \texttt{coherent} guarantees only that the results of stores will be immediately visible to shader invocations using
similarly-declared variables; calling \texttt{MemoryBarrier} is required to ensure that the stores are visible to other operations.

The following guidelines may be helpful in choosing when to use coherent memory accesses and when to use barriers.

- Data that are read-only or constant may be accessed without using coherent variables or calling \texttt{MemoryBarrier}. Updates to the read-only data via commands such as \texttt{BufferSubData} will invalidate shader caches implicitly as required.

- Data that are shared between shader invocations at a fine granularity (e.g., written by one invocation, consumed by another invocation) should use coherent variables to read and write the shared data.

- Data written by one shader invocation and consumed by other shader invocations launched as a result of its execution (dependent invocations) should use coherent variables in the producing shader invocation and call \texttt{memoryBarrier()} after the last write. The consuming shader invocation should also use coherent variables.

- Data written to image variables in one rendering pass and read by the shader in a later pass need not use coherent variables or \texttt{memoryBarrier(). Calling MemoryBarrier with the SHADER_IMAGE_ACCESS_BARRIER_BIT set in barriers between passes is necessary.

- Data written by the shader in one rendering pass and read by another mechanism (e.g., vertex or index buffer pulling) in a later pass need not use coherent variables or \texttt{memoryBarrier(). Calling MemoryBarrier with the appropriate bits set in barriers between passes is necessary.

\section*{2.11.14 Required State}

The GL maintains state to indicate which shader and program object names are in use. Initially, no shader or program objects exist, and no names are in use.

The state required per shader object consists of:

- An unsigned integer specifying the shader object name.

- An integer holding the value of \texttt{SHADER\_TYPE}.

- A boolean holding the delete status, initially \texttt{FALSE}.

- A boolean holding the status of the last compile, initially \texttt{FALSE}.

OpenGL 4.2 (Core Profile) - April 27, 2012
2.11. VERTEX SHADERS

- An array of type `char` containing the information log, initially empty.
- An integer holding the length of the information log.
- An array of type `char` containing the concatenated shader string, initially empty.
- An integer holding the length of the concatenated shader string.

The state required per program object consists of:

- An unsigned integer indicating the program object name.
- A boolean holding the delete status, initially `FALSE`.
- A boolean holding the status of the last link attempt, initially `FALSE`.
- A boolean holding the status of the last validation attempt, initially `FALSE`.
- An integer holding the number of attached shader objects.
- An array of type `char` containing the information log, initially empty.
- An integer holding the length of the information log.
- An integer holding the number of active uniforms.
- For each active uniform, three integers holding its location, size, and type, and an array of type `char` holding its name.
- An array holding the values of each active uniform.
- An integer holding the number of active attributes.
- For each active attribute, three integers holding its location, size, and type, and an array of type `char` holding its name.
- A boolean holding the hint to the retrievability of the program binary, initially `FALSE`.

Additional state required to support vertex shaders consists of:

- A bit indicating whether or not program point size mode (section 3.4.1) is enabled, initially disabled.
2.12. TESSELLATION

Additional state required to support transform feedback consists of:

- An integer holding the transform feedback mode, initially \texttt{INTERLEAVED\_ATTRIBS}.
- An integer holding the number of outputs to be captured, initially zero.
- An integer holding the length of the longest output name being captured, initially zero.
- For each output being captured, two integers holding its size and type, and an array of type \texttt{char} holding its name.

Additionally, one unsigned integer is required to hold the name of the current program object, if any.

This list of program object state is not complete. Tables 6.32-6.40 describe additional program object state specific to program binaries, geometry shaders, tessellation control and evaluation shaders, shader subroutines, and uniform blocks.

2.12 Tessellation

Tessellation is a process that reads a patch primitive and generates new primitives used by subsequent pipeline stages. The generated primitives are formed by subdividing a single triangle or quad primitive according to fixed or shader-computed levels of detail and transforming each of the vertices produced during this subdivision.

Tessellation functionality is controlled by two types of tessellation shaders: tessellation control shaders and tessellation evaluation shaders. Tessellation is considered active if and only if there is an active tessellation control or tessellation evaluation program object.

The tessellation control shader is used to read an input patch provided by the application, and emit an output patch. The tessellation control shader is run once for each vertex in the output patch and computes the attributes of that vertex. Additionally, the tessellation control shader may compute additional per-patch attributes of the output patch. The most important per-patch outputs are the tessellation levels, which are used to control the number of subdivisions performed by the tessellation primitive generator. The tessellation control shader may also write additional per-patch attributes for use by the tessellation evaluation shader. If no tessellation control shader is active, the patch provided is passed through to the tessellation primitive generator stage unmodified.
If a tessellation evaluation shader is active, the tessellation primitive generator subdivides a triangle or quad primitive into a collection of points, lines, or triangles according to the tessellation levels of the patch and the set of layout declarations specified in the tessellation evaluation shader text. The tessellation levels used to control subdivision are normally written by the tessellation control shader. If no tessellation control shader is active, default tessellation levels are instead used.

When a tessellation evaluation shader is active, it is run on each vertex generated by the tessellation primitive generator to compute the final position and other attributes of the vertex. The tessellation evaluation shader can read the relative location of the vertex in the subdivided output primitive, given by an \((u, v)\) or \((u, v, w)\) coordinate, as well as the position and attributes of any or all of the vertices in the input patch.

Tessellation operates only on patch primitives. If tessellation is active, any command that transfers vertices to the GL will generate an `INVALID_OPERATION` error if the primitive mode is not `PATCHES`.

Patch primitives are not supported by pipeline stages below the tessellation evaluation shader. If there is no active tessellation evaluation program, the error `INVALID_OPERATION` is generated by any command that transfers vertices to the GL if the primitive mode is `PATCHES`.

A program object or program pipeline object that includes a tessellation shader of any kind must also include a vertex shader. If the current program state has a tessellation shader but no vertex shader when any command that transfers vertices to the GL is called, an `INVALID_OPERATION` error will be generated.

### 2.12.1 Tessellation Control Shaders

The tessellation control shader consumes an input patch provided by the application and emits a new output patch. The input patch is an array of vertices with attributes corresponding to output variables written by the vertex shader. The output patch consists of an array of vertices with attributes corresponding to per-vertex output variables written by the tessellation control shader and a set of per-patch attributes corresponding to per-patch output variables written by the tessellation control shader. Tessellation control output variables are per-vertex by default, but may be declared as per-patch using the `patch` qualifier.

The number of vertices in the output patch is fixed when the program is linked, and is specified in tessellation control shader source code using the output layout qualifier `vertices`, as described in the OpenGL Shading Language Specification. A program will fail to link if the output patch vertex count is not specified by any tessellation control shader object attached to the program, if it is specified differently by multiple tessellation control shader objects, if it is less than or equal
to zero, or if it is greater than the implementation-dependent maximum patch size. The output patch vertex count may be queried by calling `GetProgramiv` with the symbolic constant `TESS_CONTROL_OUTPUT_VERTICES`.

Tessellation control shaders are created as described in section 2.11.1, using a type of `TESS_CONTROL_SHADER`. When a new input patch is received, the tessellation control shader is run once for each vertex in the output patch. The tessellation control shader invocations collectively specify the per-vertex and per-patch attributes of the output patch. The per-vertex attributes are obtained from the per-vertex output variables written by each invocation. Each tessellation control shader invocation may only write to per-vertex output variables corresponding to its own output patch vertex. The output patch vertex number corresponding to a given tessellation control point shader invocation is given by the built-in variable `gl_InvocationID`. Per-patch attributes are taken from the per-patch output variables, which may be written by any tessellation control shader invocation. While tessellation control shader invocations may read any per-vertex and per-patch output variable and write any per-patch output variable, reading or writing output variables also written by other invocations has ordering hazards discussed below.

**Tessellation Control Shader Variables**

Tessellation control shaders can access uniforms belonging to the current program object. The amount of storage available for uniform variables, except for subroutine uniforms and atomic counters, in the default uniform block accessed by a tessellation control shader is specified by the value of the implementation-dependent constant `MAX_TESS_CONTROL_UNIFORM_COMPONENTS`. The total amount of combined storage available for uniform variables in all uniform blocks accessed by a tessellation control shader (including the default uniform block) is specified by the value of the implementation-dependent constant `MAX_COMBINED_TESS_CONTROL_UNIFORM_COMPONENTS`. These values represent the numbers of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a tessellation control shader. A uniform matrix in the default uniform block with single- or double-precision components will consume no more than $4 \times \min(r,c)$ or $8 \times \min(r,c)$ uniform components, respectively. A scalar or vector uniform with double-precision components will consume no more than $2n$ components, where $n$ is 1 for scalars, and the component count for vectors. A link error is generated if an attempt is made to utilize more than the space available for tessellation control shader uniform variables. Uniforms are manipulated as described in section 2.11.7. Tessellation control shaders also have access to samplers to perform texturing operations, as described in section 2.11.9.

Tessellation control shaders can access the transformed attributes of all vertices.
for their input primitive using *input* variables. A vertex shader writing to output variables generates the values of these input variables. Values for any inputs that are not written by a vertex shader are undefined.

Additionally, tessellation control shaders can write to one or more *output* including per-vertex attributes for the vertices of the output patch and per-patch attributes of the patch. Tessellation control shaders can also write to a set of built-in per-vertex and per-patch outputs defined in the OpenGL Shading Language. The per-vertex and per-patch attributes of the output patch are used by the tessellation primitive generator (section 2.12.2) and may be read by tessellation control shader (section 2.12.3).

**Tessellation Control Shader Execution Environment**

If there is an active program for the tessellation control stage, the executable version of the program’s tessellation control shader is used to process patches resulting from the primitive assembly stage. When tessellation control shader execution completes, the input patch is consumed. A new patch is assembled from the per-vertex and per-patch output variables written by the shader and is passed to subsequent pipeline stages.

There are several special considerations for tessellation control shader execution described in the following sections.

**Texture Access**

The Shader-Only Texturing subsection of section 2.11.12 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to tessellation control shaders.

**Tessellation Control Shader Inputs**

Section 7.1 of the OpenGL Shading Language Specification describes the built-in variable array *gl_in* available as input to a tessellation control shader. *gl_in* receives values from equivalent built-in output variables written by the vertex shader (section 2.11.12). Each array element of *gl_in* is a structure holding values for a specific vertex of the input patch. The length of *gl_in* is equal to the implementation-dependent maximum patch size (*gl_MaxPatchVertices*). Behavior is undefined if *gl_in* is indexed with a vertex index greater than or equal to the current patch size. The members of each element of the *gl_in* array are *gl_Position*, *gl_PointSize*, *gl_ClipDistance*, and *gl_ClipVertex*.

Tessellation control shaders have available several other built-in input variables not replicated per-vertex and not contained in *gl_in*, including:
2.12. TESSELLATION

- The variable `gl_PatchVerticesIn` holds the number of vertices in the input patch being processed by the tessellation control shader.

- The variable `gl_PrimitiveID` is filled with the number of primitives processed by the drawing command which generated the input vertices. The first primitive generated by a drawing command is numbered zero, and the primitive ID counter is incremented after every individual point, line, or triangle primitive is processed. Restarting a primitive topology using the primitive restart index has no effect on the primitive ID counter.

- The variable `gl_InvocationID` holds an invocation number for the current tessellation control shader invocation. Tessellation control shaders are invoked once per output patch vertex, and invocations are numbered beginning with zero.

Similarly to the built-in inputs, each user-defined input variable has a value for each vertex and thus needs to be declared as arrays or inside input blocks declared as arrays. Declaring an array size is optional. If no size is specified, it will be taken from the implementation-dependent maximum patch size (`gl_MaxPatchVertices`). If a size is specified, it must match the maximum patch size; otherwise, a link error will occur. Since the array size may be larger than the number of vertices found in the input patch, behavior is undefined if a per-vertex input variable is accessed using an index greater than or equal to the number of vertices in the input patch. The OpenGL Shading Language doesn’t support multi-dimensional arrays; therefore, user-defined tessellation control shader inputs corresponding to vertex shader outputs declared as arrays must be declared as array members of an input block that is itself declared as an array.

Similarly to the limit on vertex shader output components (see section 2.11.11), there is a limit on the number of components of input variables that can be read by the tessellation control shader, given by the value of the implementation-dependent constant `MAX_TESS_CONTROL_INPUT_COMPONENTS`.

When a program is linked, all components of any input variable read by a tessellation control shader will count against this limit. A program whose tessellation control shader exceeds this limit may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Component counting rules for different variable types and variable declarations are the same as for `MAX_VERTEX_OUTPUT_COMPONENTS`. (see section 2.11.11).
2.12. TESSELLATION

Tessellation Control Shader Outputs

Section 7.1 of the OpenGL Shading Language Specification describes the built-in variable array \texttt{gl\_out} available as an output for a tessellation control shader. \texttt{gl\_out} passes values to equivalent built-in input variables read by subsequent shader stages or to subsequent fixed functionality vertex processing pipeline stages. Each array element of \texttt{gl\_out} is a structure holding values for a specific vertex of the output patch. The length of \texttt{gl\_out} is equal to the output patch size specified in the tessellation control shader output layout declaration. \texttt{(gl\_VerticesOut)} The members of each element of the \texttt{gl\_out} array are \texttt{gl\_Position}, \texttt{gl\_PointSize}, and \texttt{gl\_ClipDistance}, and behave identically to equivalently named vertex shader outputs (section 2.11.12).

Tessellation shaders additionally have two built-in per-patch output arrays, \texttt{gl\_TessLevelOuter} and \texttt{gl\_TessLevelInner}. These arrays are not replicated for each output patch vertex and are not members of \texttt{gl\_out}. \texttt{gl\_TessLevelOuter} is an array of four floating-point values specifying the approximate number of segments that the tessellation primitive generator should use when subdividing each outer edge of the primitive it subdivides. \texttt{gl\_TessLevelInner} is an array of two floating-point values specifying the approximate number of segments used to produce a regularly-subdivided primitive interior. The values written to \texttt{gl\_TessLevelOuter} and \texttt{gl\_TessLevelInner} need not be integers, and their interpretation depends on the type of primitive the tessellation primitive generator will subdivide and other tessellation parameters, as discussed in the following section.

A tessellation control shader may also declare user-defined per-vertex output variables. User-defined per-vertex output variables are declared with the qualifier \texttt{out} and have a value for each vertex in the output patch. Such variables must be declared as arrays or inside output blocks declared as arrays. Declaring an array size is optional. If no size is specified, it will be taken from the output patch size \texttt{(gl\_VerticesOut)} declared in the shader. If a size is specified, it must match the maximum patch size; otherwise, a link error will occur. The OpenGL Shading Language doesn’t support multi-dimensional arrays; therefore, user-defined per-vertex tessellation control shader outputs with multiple elements per vertex must be declared as array members of an output block that is itself declared as an array.

While per-vertex output variables are declared as arrays indexed by vertex number, each tessellation control shader invocation may write only to those outputs corresponding to its output patch vertex. Tessellation control shaders must use the input variable \texttt{gl\_InvocationID} as the vertex number index when writing to per-vertex output variables.

Additionally, a tessellation control shader may declare per-patch output vari-
ables using the qualifier \texttt{patch out}. Unlike per-vertex outputs, per-patch outputs do not correspond to any specific vertex in the patch, and are not indexed by vertex number. Per-patch outputs declared as arrays have multiple values for the output patch; similarly declared per-vertex outputs would indicate a single value for each vertex in the output patch. User-defined per-patch outputs are not used by the tessellation primitive generator, but may be read by tessellation evaluation shaders.

There are several limits on the number of components of output variables that can be written by the tessellation control shader. The number of components of active per-vertex output variables may not exceed the value of \texttt{MAX_TESS\_CONTROL\_OUTPUT\_COMPONENTS}. The number of components of active per-patch output variables may not exceed the value of \texttt{MAX_TESS\_PATCH\_COMPONENTS}. The built-in outputs \texttt{gl\_TessLevelOuter} and \texttt{gl\_TessLevelInner} are not counted against the per-patch limit. The total number of components of active per-vertex and per-patch outputs is derived by multiplying the per-vertex output component count by the output patch size and then adding the per-patch output component count. The total component count may not exceed \texttt{MAX_TESS\_CONTROL\_TOTAL\_OUTPUT\_COMPONENTS}.

When a program is linked, all components of any output variable written by a tessellation control shader will count against this limit. A program exceeding any of these limits may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Counting rules for different variable types and variable declarations are the same as for \texttt{MAX_VERTEX\_OUTPUT\_COMPONENTS}. (see section 2.11.11).

**Tessellation Control Shader Execution Order**

For tessellation control shaders with a declared output patch size greater than one, the shader is invoked more than once for each input patch. The order of execution of one tessellation control shader invocation relative to the other invocations for the same input patch is largely undefined. The built-in function \texttt{barrier} provides some control over relative execution order. When a tessellation control shader calls the \texttt{barrier} function, its execution pauses until all other invocations have also called the same function. Output variable assignments performed by any invocation executed prior to calling \texttt{barrier} will be visible to any other invocation after the call to \texttt{barrier} returns. Shader output values read in one invocation but written by another may be undefined without proper use of \texttt{barrier}; full rules are found in the OpenGL Shading Language Specification.

The \texttt{barrier} function may only be called inside the main entry point of the tessellation control shader and may not be called in potentially divergent flow control. In particular, \texttt{barrier} may not be called inside a switch statement, in either...
sub-statement of an if statement, inside a do, for, or while loop, or at any point after a return statement in the function main.

2.12.2 Tessellation Primitive Generation

If a tessellation evaluation shader is present, the tessellation primitive generator consumes the input patch and produces a new set of basic primitives (points, lines, or triangles). These primitives are produced by subdividing a geometric primitive (rectangle or triangle) according to the per-patch tessellation levels written by the tessellation control shader, if present, or taken from default patch parameter values. This subdivision is performed in an implementation-dependent manner. If no tessellation evaluation shader is present, the tessellation primitive generator passes incoming primitives through without modification.

The type of subdivision performed by the tessellation primitive generator is specified by an input layout declaration in the tessellation evaluation shader using one of the identifiers `triangles`, `quads`, and `isolines`. For `triangles`, the primitive generator subdivides a triangle primitive into smaller triangles. For `quads`, the primitive generator subdivides a rectangle primitive into smaller triangles. For `isolines`, the primitive generator subdivides a rectangle primitive into a collection of line segments arranged in strips stretching horizontally across the rectangle. Each vertex produced by the primitive generator has an associated \((u, v, w)\) or \((u, v)\) position in a normalized parameter space, with parameter values in the range \([0, 1]\), as illustrated in figure 2.7. For `triangles`, the vertex position is a barycentric coordinate \((u, v, w)\), where \(u + v + w = 1\), and indicates the relative influence of the three vertices of the triangle on the position of the vertex. For `quads` and `isolines`, the position is a \((u, v)\) coordinate indicating the relative horizontal and vertical position of the vertex relative to the subdivided rectangle. The subdivision process is explained in more detail in subsequent sections.

When no tessellation control shader is present, the tessellation levels are taken from default patch tessellation levels. These default levels are set by calling

```c
void PatchParameterfv( enum pname, const float *values );
```

If `pname` is `PATCH_DEFAULT_OUTER_LEVEL`, `values` specifies an array of four floating-point values corresponding to the four outer tessellation levels for each subsequent patch. If `pname` is `PATCH_DEFAULT_INNER_LEVEL`, `values` specifies an array of two floating-point values corresponding to the two inner tessellation levels.

A patch is discarded by the tessellation primitive generator if any relevant outer tessellation level is less than or equal to zero. Patches will also be discarded if
Figure 2.7. Domain parameterization for tessellation generator primitive modes (triangles, quads, or isolines). The coordinates illustrate the value of gl_TessCoord at the corners of the domain. The labels on the edges indicate the inner (IL0 and IL1) and outer (OL0 through OL3) tessellation level values used to control the number of subdivisions along each edge of the domain.
any outer tessellation level corresponds to a floating-point NaN (not a number) in
implementations supporting NaN. When patches are discarded, no new primitives
will be generated and the tessellation evaluation program will not be run. For
quads, all four outer levels are relevant. For triangles and isolines, only the
first three or two outer levels, respectively, are relevant. Negative inner levels will
not cause a patch to be discarded; they will be clamped as described below.

Each of the tessellation levels is used to determine the number and spacing
of segments used to subdivide a corresponding edge. The method used to derive
the number and spacing of segments is specified by an input layout declaration
in the tessellation evaluation shader using one of the identifiers equal_spacing,
fractional_even_spacing, or fractional_odd_spacing. If no spacing is
specified in the tessellation evaluation shader, equal_spacing will be used.

If equal_spacing is used, the floating-point tessellation level is first clamped
to the range \([1, \text{max}]\), where \(\text{max}\) is the implementation-dependent maximum tes-
sellation level (the value of \texttt{MAX_TESS_GEN_LEVEL}). The result is rounded up to
the nearest integer \(n\), and the corresponding edge is divided into \(n\) segments of
equal length in \((u,v)\) space.

If fractional_even_spacing is used, the tessellation level is first clamped
to the range \([2, \text{max}]\) and then rounded up to the nearest even integer \(n\). If
fractional_odd_spacing is used, the tessellation level is clamped to the range
\([1, \text{max} - 1]\) and then rounded up to the nearest odd integer \(n\). If \(n\) is one, the edge
will not be subdivided. Otherwise, the corresponding edge will be divided into
\(n - 2\) segments of equal length, and two additional segments of equal length that
are typically shorter than the other segments. The length of the two additional seg-
ments relative to the others will decrease monotonically with the value of \(n - f\),
where \(f\) is the clamped floating-point tessellation level. When \(n - f\) is zero, the
additional segments will have equal length to the other segments. As \(n - f\) ap-
proaches 2.0, the relative length of the additional segments approaches zero. The
two additional segments should be placed symmetrically on opposite sides of the
subdivided edge. The relative location of these two segments is undefined, but
must be identical for any pair of subdivided edges with identical values of \(f\).

When the tessellation primitive generator produces triangles (in the
triangles or quads modes), the orientation of all triangles can be specified by
an input layout declaration in the tessellation evaluation shader using the identifiers
cw and ccw. If the order is cw, the vertices of all generated triangles will have a
clockwise ordering in \((u,v)\) or \((u,v,w)\) space, as illustrated in figure 2.7. If the
order is ccw, the vertices will be specified in counter-clockwise order. If no layout
is specified, ccw will be used.

For all primitive modes, the tessellation primitive generator is capable of gen-
erating points instead of lines or triangles. If an input layout declaration in the

OpenGL 4.2 (Core Profile) - April 27, 2012
2.12. TESSELLATION

tessellation evaluation shader specifies the identifier point_mode, the primitive generator will generate one point for each unique vertex produced by tessellation. Otherwise, the primitive generator will produce a collection of line segments or triangles according to the primitive mode.

The points, lines, or triangles produced by the tessellation primitive generator are passed to subsequent pipeline stages in an implementation-dependent order.

Triangle Tessellation

If the tessellation primitive mode is triangles, an equilateral triangle is subdivided into a collection of triangles covering the area of the original triangle. First, the original triangle is subdivided into a collection of concentric equilateral triangles. The edges of each of these triangles are subdivided, and the area between each triangle pair is filled by triangles produced by joining the vertices on the subdivided edges. The number of concentric triangles and the number of subdivisions along each triangle except the outermost is derived from the first inner tessellation level. The edges of the outermost triangle are subdivided independently, using the first, second, and third outer tessellation levels to control the number of subdivisions of the $u = 0$ (left), $v = 0$ (bottom), and $w = 0$ (right) edges, respectively. The second inner tessellation level and the fourth outer tessellation level have no effect in this mode.

If the first inner tessellation level and all three outer tessellation levels are exactly one after clamping and rounding, only a single triangle with $(u, v, w)$ coordinates of $(0, 0, 1)$, $(1, 0, 0)$, and $(0, 1, 0)$ is generated. If the inner tessellation level is one and any of the outer tessellation levels is greater than one, the inner tessellation level is treated as though it were originally specified as $1 + \epsilon$ and will be rounded up to result in a two- or three-segment subdivision according to the tessellation spacing.

If any tessellation level is greater than one, tessellation begins by producing a set of concentric inner triangles and subdividing their edges. First, the three outer edges are temporarily subdivided using the clamped and rounded first inner tessellation level and the specified tessellation spacing, generating $n$ segments. For the outermost inner triangle, the inner triangle is degenerate – a single point at the center of the triangle – if $n$ is two. Otherwise, for each corner of the outer triangle, an inner triangle corner is produced at the intersection of two lines extended perpendicular to the corner’s two adjacent edges running through the vertex of the subdivided outer edge nearest that corner. If $n$ is three, the edges of the inner triangle are not subdivided and is the final triangle in the set of concentric triangles. Otherwise, each edge of the inner triangle is divided into $n - 2$ segments, with the $n - 1$ vertices of this subdivision produced by intersecting the inner edge with
2.12. TESSELLATION

Figure 2.8. Inner triangle tessellation with inner tessellation levels of (a) five and (b) four, respectively (not to scale). Solid black circles depict vertices along the edges of the concentric triangles. The edges of inner triangles are subdivided by intersecting the edge with segments perpendicular to the edge passing through each inner vertex of the subdivided outer edge. Dotted lines depict edges connecting corresponding vertices on the inner and outer triangle edges.

Once all the concentric triangles are produced and their edges are subdivided, the area between each pair of adjacent inner triangles is filled completely with a set of non-overlapping triangles. In this subdivision, two of the three vertices of each triangle are taken from adjacent vertices on a subdivided edge of one triangle; the third is one of the vertices on the corresponding edge of the other triangle. If the innermost triangle is degenerate (i.e., a point), the triangle containing it is subdivided into six triangles by connecting each of the six vertices on that triangle with the center point. If the innermost triangle is not degenerate, that triangle is added to the set of generated triangles as-is.

After the area corresponding to any inner triangles is filled, the primitive generator generates triangles to cover area between the outermost triangle and the outer...
2.12. TESSELLATION

 ermost inner triangle. To do this, the temporary subdivision of the outer triangle edge above is discarded. Instead, the $u = 0, v = 0,$ and $w = 0$ edges are subdivided according to the first, second, and third outer tessellation levels, respectively, and the tessellation spacing. The original subdivision of the first inner triangle is retained. The area between the outer and first inner triangles is completely filled by non-overlapping triangles as described above. If the first (and only) inner triangle is degenerate, a set of triangles is produced by connecting each vertex on the outer triangle edges with the center point.

After all triangles are generated, each vertex in the subdivided triangle is assigned a barycentric $(u, v, w)$ coordinate based on its location relative to the three vertices of the outer triangle.

The algorithm used to subdivide the triangular domain in $(u, v, w)$ space into individual triangles is implementation-dependent. However, the set of triangles produced will completely cover the domain, and no portion of the domain will be covered by multiple triangles. The order in which the generated triangles passed to subsequent pipeline stages and the order of the vertices in those triangles are both implementation-dependent. However, when depicted in a manner similar to figure 2.8, the order of the vertices in the generated triangles will be either all clockwise or all counter-clockwise, according to the vertex order layout declaration.

Quad Tessellation

If the tessellation primitive mode is quads, a rectangle is subdivided into a collection of triangles covering the area of the original rectangle. First, the original rectangle is subdivided into a regular mesh of rectangles, where the number of rectangles along the $u = 0$ and $u = 1$ (vertical) and $v = 0$ and $v = 1$ (horizontal) edges are derived from the first and second inner tessellation levels, respectively. All rectangles, except those adjacent to one of the outer rectangle edges, are decomposed into triangle pairs. The outermost rectangle edges are subdivided independently, using the first, second, third, and fourth outer tessellation levels to control the number of subdivisions of the $u = 0$ (left), $v = 0$ (bottom), $u = 1$ (right), and $v = 1$ (top) edges, respectively. The area between the inner rectangles of the mesh and the outer rectangle edges are filled by triangles produced by joining the vertices on the subdivided outer edges to the vertices on the edge of the inner rectangle mesh.

If both clamped inner tessellation levels and all four clamped outer tessellation levels are exactly one, only a single triangle pair covering the outer rectangle is generated. Otherwise, if either clamped inner tessellation level is one, that tessellation level is treated as though it were originally specified as $1 + \epsilon$, which would rounded up to result in a two- or three-segment subdivision according to the tessel-
If any tessellation level is greater than one, tessellation begins by subdividing the $u = 0$ and $u = 1$ edges of the outer rectangle into $m$ segments using the clamped and rounded first inner tessellation level and the tessellation spacing. The $v = 0$ and $v = 1$ edges are subdivided into $n$ segments using the second inner tessellation level. Each vertex on the $u = 0$ and $v = 0$ edges are joined with the corresponding vertex on the $u = 1$ and $v = 1$ edges to produce a set of vertical and horizontal lines that divide the rectangle into a grid of smaller rectangles. The primitive generator emits a pair of non-overlapping triangles covering each such rectangle not adjacent to an edge of the outer rectangle. The boundary of the region covered by these triangles forms an inner rectangle, the edges of which are subdivided by the grid vertices that lie on the edge. If either $m$ or $n$ is two, the inner rectangle is degenerate, and one or both of the rectangle’s “edges” consist of a single point. This subdivision is illustrated in figure 2.9.

After the area corresponding to the inner rectangle is filled, the primitive generator must produce triangles to cover area between the inner and outer rectangles. To do this, the subdivision of the outer rectangle edge above is discarded. Instead, the $u = 0$, $v = 0$, $u = 1$, and $v = 1$ edges are subdivided according to the first, second, third, and fourth outer tessellation levels, respectively, and the tessellation spacing. The original subdivision of the inner rectangle is retained. The area between the outer and inner rectangles is completely filled by non-overlapping triangles. Two of the three vertices of each triangle are adjacent vertices on a subdivided edge of one rectangle; the third is one of the vertices on the corresponding edge of the other triangle. If either edge of the innermost rectangle is degenerate, the area near the corresponding outer edges is filled by connecting each vertex on the outer edge with the single vertex making up the inner “edge”.

The algorithm used to subdivide the rectangular domain in $(u, v)$ space into individual triangles is implementation-dependent. However, the set of triangles produced will completely cover the domain, and no portion of the domain will be covered by multiple triangles. The order in which the generated triangles passed to subsequent pipeline stages and the order of the vertices in those triangles are both implementation-dependent. However, when depicted in a manner similar to figure 2.9, the order of the vertices in the generated triangles will be either all clockwise or all counter-clockwise, according to the vertex order layout declaration.

**Isoline Tessellation**

If the tessellation primitive mode is *isolines*, a set of independent horizontal line segments is drawn. The segments are arranged into connected strips, where each strip has a constant $v$ coordinate, and the $u$ coordinates of the strip cover the
Figure 2.9. Inner quad tessellation with inner tessellation levels of (a) \((4, 2)\) and (b) \((7, 4)\), respectively. Gray regions on the bottom figure depict the 10 inner rectangles, each of which will be subdivided into two triangles. Solid black circles depict vertices on the boundary of the outer and inner rectangles, where the inner rectangle on the top figure is degenerate (a single line segment). Dotted lines depict the horizontal and vertical edges connecting corresponding vertices on the inner and outer rectangle edges.
2.12. TESSELLATION

full range \([0, 1]\). The number of segments in each strip is derived from the first outer tessellation level; the number of line strips drawn is derived from the second outer tessellation level. Both inner tessellation levels and the third and fourth outer tessellation levels have no effect in this mode.

As with quad tessellation above, isoline tessellation begins with a rectangle. The \(u = 0\) and \(u = 1\) edges of the rectangle are subdivided according to the second outer tessellation level. For the purposes of this subdivision, the tessellation spacing is ignored and treated as \textsc{equal}. A line is drawn from each vertex on the \(u = 0\) rectangle edge with the corresponding vertex on the \(u = 1\) rectangle edge, except that no line is drawn between \((0,1)\) and \((1,1)\). If the number of segments on the subdivided \(u = 0\) and \(u = 1\) edges is \(n\), this process will result in \(n\) equally spaced lines with constant \(v\) coordinates of \(0, \frac{1}{n}, \frac{2}{n}, \ldots, \frac{n-1}{n}\).

Each of the \(n\) lines is then subdivided according to the first outer tessellation level and the tessellation spacing, resulting in \(m\) line segments. Each segment of each line is emitted by the tessellation primitive generator, as illustrated in figure 2.10.

The order in which the generated line segments are passed to subsequent pipeline stages and the order of the vertices in each generated line segment are both implementation-dependent.

2.12.3 Tessellation Evaluation Shaders

If active, the tessellation evaluation shader takes the \((u, v)\) or \((u, v, w)\) location of each vertex in the primitive subdivided by the tessellation primitive generator, and generates a vertex with a position and associated attributes. The tessellation evaluation shader can read any of the vertices of its input patch, which is the output patch produced by the tessellation control shader (if present) or provided by the application and transformed by the vertex shader (if no control shader is used). Tessellation evaluation shaders are created as described in section 2.11.1, using a type of \textsc{tess_evaluation_shader}.

Each invocation of the tessellation evaluation shader writes the attributes of exactly one vertex. The number of vertices evaluated per patch depends on the tessellation level values computed by the tessellation control shaders (if present) or specified as patch parameters. Tessellation evaluation shader invocations run independently, and no invocation can access the variables belonging to another invocation. All invocations are capable of accessing all the vertices of their corresponding input patch.

If a tessellation control shader is present, the number of the vertices in the input patch is fixed and is equal to the tessellation control shader output patch size parameter in effect when the program was last linked. If no tessellation control

OpenGL 4.2 (Core Profile) - April 27, 2012
Figure 2.10. Isoline tessellation with the first two outer tessellation levels of (a) $(3, 1)$ and (b) $(6, 4)$, respectively. Line segments connecting the vertices marked with solid black circles are emitted by the primitive generator. Vertices marked with empty circles correspond to $(u, v)$ coordinates of $(0, 1)$ and $(1, 1)$, where no line segments are generated.
2.12. TESSELLATION

shader is present, the input patch is provided by the application can have a variable number of vertices, as specified by PatchParameteri.

Tessellation Evaluation Shader Variables

Tessellation evaluation shaders can access uniforms belonging to the current program object. The amount of storage available for uniform variables, except for subroutine uniforms and atomic counters, in the default uniform block accessed by a tessellation evaluation shader is specified by the value of the implementation-dependent constant MAX_TESS_EVALUATION_UNIFORM_COMPONENTS. The total amount of combined storage available for uniform variables in all uniform blocks accessed by a tessellation evaluation shader (including the default uniform block) is specified by the value of the implementation-dependent constant MAX_COMBINED_TESS_EVALUATION_UNIFORM_COMPONENTS. These values represent the numbers of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a tessellation evaluation shader.

A uniform matrix in the default uniform block with single- or double-precision components will consume no more than $4 \times \min(r,c)$ or $8 \times \min(r,c)$ uniform components, respectively. A scalar or vector uniform with double-precision components will consume no more than $2n$ components, where $n$ is 1 for scalars, and the component count for vectors. A link error is generated if an attempt is made to utilize more than the space available for tessellation evaluation shader uniform variables. Uniforms are manipulated as described in section 2.11.7. Tessellation evaluation shaders also have access to samplers to perform texturing operations, as described in section 2.11.9.

Tessellation evaluation shaders can access the transformed attributes of all vertices for their input primitive using input variables. If active, a tessellation control shader writing to output variables generates the values of these input variables. If no tessellation control shader is active, input variables will be obtained from vertex shader outputs. Values for any input variable that are not written by a vertex or tessellation control shader are undefined.

Additionally, tessellation evaluation shaders can write to one or more output variables that will be passed to subsequent programmable shader stages or fixed functionality vertex pipeline stages.

Tessellation Evaluation Shader Execution Environment

If there is an active program for the tessellation evaluation stage, the executable version of the program’s tessellation evaluation shader is used to process vertices produced by the tessellation primitive generator. During this processing, the shader
may access the input patch processed by the primitive generator. When tessellation evaluation shader execution completes, a new vertex is assembled from the output variables written by the shader and is passed to subsequent pipeline stages.

There are several special considerations for tessellation evaluation shader execution described in the following sections.

**Texture Access**

The Shader-Only Texturing subsection of section 2.11.12 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to tessellation evaluation shaders.

**Tessellation Evaluation Shader Inputs**

Section 7.1 of the OpenGL Shading Language Specification describes the built-in variable array `gl_in` available as input to a tessellation evaluation shader. `gl_in` receives values from equivalent built-in output variables written by a previous shader (section 2.11.12). If a tessellation control shader active, the values of `gl_in` will be taken from tessellation control shader outputs. Otherwise, they will be taken from vertex shader outputs. Each array element of `gl_in` is a structure holding values for a specific vertex of the input patch. The length of `gl_in` is equal to the implementation-dependent maximum patch size (`gl_MaxPatchVertices`). Behavior is undefined if `gl_in` is indexed with a vertex index greater than or equal to the current patch size. The members of each element of the `gl_in` array are `gl_Position`, `gl_PointSize`, and `gl_ClipDistance`.

Tessellation evaluation shaders have available several other built-in input variables not replicated per-vertex and not contained in `gl_in`, including:

- The variables `gl_PatchVerticesIn` and `gl_PrimitiveID` are filled with the number of the vertices in the input patch and a primitive number, respectively. They behave exactly as the identically named inputs for tessellation control shaders.

- The variable `gl_TessCoord` is a three-component floating-point vector consisting of the \((u, v, w)\) coordinate of the vertex being processed by the tessellation evaluation shader. The values of \(u, v,\) and \(w\) are in the range \([0, 1]\), and vary linearly across the primitive being subdivided. For tessellation primitive modes of `quads` or `isolines`, the \(w\) value is always zero. The \((u, v, w)\) coordinates are generated by the tessellation primitive generator in a manner dependent on the primitive mode, as described in section 2.12.2. `gl_TessCoord` is not an array; it specifies the location of the
vertex being processed by the tessellation evaluation shader, not of any vertex in the input patch.

- The variables `gl_TessLevelOuter` and `gl_TessLevelInner` are arrays holding outer and inner tessellation levels of the patch, as used by the tessellation primitive generator. If a tessellation control shader is active, the tessellation levels will be taken from the corresponding outputs of the tessellation control shader. Otherwise, the default levels provided as patch parameters are used. Tessellation level values loaded in these variables will be prior to the clamping and rounding operations performed by the primitive generator as described in section 2.12.2. For triangular tessellation, `gl_TessLevelOuter[3]` and `gl_TessLevelInner[1]` will be undefined. For isoline tessellation, `gl_TessLevelOuter[2]`, `gl_TessLevelOuter[3]`, and both values in `gl_TessLevelInner` are undefined.

A tessellation evaluation shader may also declare user-defined per-vertex input variables. User-defined per-vertex input variables are declared with the qualifier `in` and have a value for each vertex in the input patch. User-defined per-vertex input variables have a value for each vertex and thus need to be declared as arrays or inside input blocks declared as arrays. Declaring an array size is optional. If no size is specified, it will be taken from the implementation-dependent maximum patch size (`gl_MaxPatchVertices`). If a size is specified, it must match the maximum patch size; otherwise, a link error will occur. Since the array size may be larger than the number of vertices found in the input patch, behavior is undefined if a per-vertex input variable is accessed using an index greater than or equal to the number of vertices in the input patch. The OpenGL Shading Language doesn’t support multi-dimensional arrays; therefore, user-defined tessellation evaluation shader inputs corresponding to vertex shader outputs declared as arrays must be declared as array members of an input block that is itself declared as an array.

Additionally, a tessellation evaluation shader may declare per-patch input variables using the qualifier `patch in`. Unlike per-vertex inputs, per-patch inputs do not correspond to any specific vertex in the patch, and are not indexed by vertex number. Per-patch inputs declared as arrays have multiple values for the input patch; similarly declared per-vertex inputs would indicate a single value for each vertex in the output patch. User-defined per-patch input variables are filled with corresponding per-patch output values written by the tessellation control shader. If no tessellation control shader is active, all such variables are undefined.

Similarly to the limit on vertex shader output components (see section 2.11.11), there is a limit on the number of components of per-vertex and per-patch input
variables that can be read by the tessellation evaluation shader, given by the values of the implementation-dependent constants\texttt{MAX_TESS_EVALUATION_INPUT_COMPONENTS} and \texttt{MAX_TESS_PATCH_COMPONENTS}, respectively. The built-in inputs \texttt{gl_TessLevelOuter} and \texttt{gl_TessLevelInner} are not counted against the per-patch limit.

When a program is linked, all components of any input variable read by a tessellation evaluation shader will count against this limit. A program whose tessellation evaluation shader exceeds this limit may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Component counting rules for different variable types and variable declarations are the same as for \texttt{MAX_VERTEX_OUTPUT_COMPONENTS}. (see section 2.11.11).

\textbf{Tessellation Evaluation Shader Outputs}

Tessellation evaluation shaders have a number of built-in output variables used to pass values to equivalent built-in input variables read by subsequent shader stages or to subsequent fixed functionality vertex processing pipeline stages. These variables are \texttt{gl_Position}, \texttt{gl_PointSize}, and \texttt{gl_ClipDistance}, and all behave identically to equivalently named vertex shader outputs (see section 2.11.12). A tessellation evaluation shader may also declare user-defined per-vertex output variables.

Similarly to the limit on vertex shader output components (see section 2.11.11), there is a limit on the number of components of output output variables that can be written by the tessellation evaluation shader, given by the values of the implementation-dependent constant \texttt{MAX_TESS_EVALUATION_OUTPUT_COMPONENTS}.

When a program is linked, all components of any output variable written by a tessellation evaluation shader will count against this limit. A program whose tessellation evaluation shader exceeds this limit may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Counting rules for different variable types and variable declarations are the same as for \texttt{MAX_VERTEX_OUTPUT_COMPONENTS}. (see section 2.11.11).

\section{Geometry Shaders}

After vertices are processed, they are arranged into primitives, as described in section 2.6.1. This section describes optional \textit{geometry shaders}, an additional pipeline stage defining operations to further process those primitives. Geometry shaders are
2.13. GEOMETRY SHADERS

defined by source code in the OpenGL Shading Language, in the same manner as vertex shaders. They operate on a single primitive at a time and emit one or more output primitives, all of the same type, which are then processed like an equivalent OpenGL primitive specified by the application. The original primitive is discarded after geometry shader execution. The inputs available to a geometry shader are the transformed attributes of all the vertices that belong to the primitive. Additional adjacency primitives are available which also make the transformed attributes of neighboring vertices available to the shader. The results of the shader are a new set of transformed vertices, arranged into primitives by the shader.

The geometry shader pipeline stage is inserted after primitive assembly, prior to transform feedback (section 2.17).

Geometry shaders are created as described in section 2.11.1 using a type of GEOMETRY_SHADER. They are attached to and used in program objects as described in section 2.11.3. When the program object currently in use includes a geometry shader, its geometry shader is considered active, and is used to process primitives. If the program object has no geometry shader, this stage is bypassed.

A program object or program pipeline object that includes a geometry shader must also include a vertex shader. If the current program state has a geometry shader but no vertex shader when any command that transfers vertices to the GL is called, an INVALID_OPERATION error will be generated.

2.13.1 Geometry Shader Input Primitives

A geometry shader can operate on one of five input primitive types. Depending on the input primitive type, one to six input vertices are available when the shader is executed. Each input primitive type supports a subset of the primitives provided by the GL. If a geometry shader is active, any command that transfers vertices to the GL will generate an INVALID_OPERATION error if the primitive mode parameter is incompatible with the input primitive type of the geometry shader of the active geometry program object, as discussed below.

A geometry shader that accesses more input vertices than are available for a given input primitive type can be successfully compiled, because the input primitive type is not part of the shader object. However, a program object containing a shader object that accesses more input vertices than are available for the input primitive type of the program object will not link.

The input primitive type is specified in the geometry shader source code using an input layout qualifier, as described in the OpenGL Shading Language Specification. A program will fail to link if the input primitive type is not specified by any geometry shader object attached to the program, or if it is specified differently by multiple geometry shader objects. The input primitive type may be queried by
calling `GetProgramiv` with the symbolic constant `GEOMETRY_INPUT_TYPE`. The supported types and the corresponding OpenGL Shading Language input layout qualifier keywords are:

**Points** *(points)*

Geometry shaders that operate on points are valid only for the `POINTS` primitive type. There is only a single vertex available for each geometry shader invocation.

**Lines** *(lines)*

Geometry shaders that operate on line segments are valid only for the `LINES`, `LINE_STRIP`, and `LINE_LOOP` primitive types. There are two vertices available for each geometry shader invocation. The first vertex refers to the vertex at the beginning of the line segment and the second vertex refers to the vertex at the end of the line segment. See also section 2.13.4.

**Lines with Adjacency** *(lines_adjacency)*

Geometry shaders that operate on line segments with adjacent vertices are valid only for the `LINES_ADJACENCY` and `LINE_STRIP_ADJACENCY` primitive types. There are four vertices available for each program invocation. The second vertex refers to attributes of the vertex at the beginning of the line segment and the third vertex refers to the vertex at the end of the line segment. The first and fourth vertices refer to the vertices adjacent to the beginning and end of the line segment, respectively.

**Triangles** *(triangles)*

Geometry shaders that operate on triangles are valid for the `TRIANGLES`, `TRIANGLE_STRIP` and `TRIANGLE_FAN` primitive types. There are three vertices available for each program invocation. The first, second and third vertices refer to attributes of the first, second and third vertex of the triangle, respectively.

**Triangles with Adjacency** *(triangles_adjacency)*

Geometry shaders that operate on triangles with adjacent vertices are valid for the `TRIANGLES_ADJACENCY` and `TRIANGLE_STRIP_ADJACENCY` primitive types. There are six vertices available for each program invocation. The first, third and fifth vertices refer to attributes of the first, second and third vertex of the triangle, respectively. The second, fourth and sixth vertices refer to attributes of the vertices adjacent to the edges from the first to the second vertex, from the second to the third vertex, and from the third to the first vertex, respectively.
2.13. GEOMETRY SHADERS

2.13.2 Geometry Shader Output Primitives

A geometry shader can generate primitives of one of three types. The supported output primitive types are points (POINTS), line strips (LINE_STRIP), and triangle strips (TRIANGLE_STRIP). The vertices output by the geometry shader are assembled into points, lines, or triangles based on the output primitive type in the manner described in section 2.6.1. The resulting primitives are then further processed as described in section 2.13.4. If the number of vertices emitted by the geometry shader is not sufficient to produce a single primitive, nothing is drawn. The number of vertices output by the geometry shader is limited to a maximum count specified in the shader.

The output primitive type and maximum output vertex count are specified in the geometry shader source code using an output layout qualifier, as described in section 4.3.8.1 of the OpenGL Shading Language Specification. A program will fail to link if either the output primitive type or maximum output vertex count are not specified by any geometry shader object attached to the program, or if they are specified differently by multiple geometry shader objects. The output primitive type and maximum output vertex count of a linked program may be queried by calling GetProgramiv with the symbolic constants GEOMETRY_OUTPUT_TYPE and GEOMETRY_VERTICES_OUT, respectively.

2.13.3 Geometry Shader Variables

Geometry shaders can access uniforms belonging to the current program object. The amount of storage available for uniform variables, except for subroutine uniforms and atomic counters, in the default uniform block accessed by a geometry shader is specified by the value of the implementation-dependent constant MAX_-GEOMETRY_UNIFORM_COMPONENTS. The total amount of combined storage available for uniform variables in all uniform blocks accessed by a geometry shader (including the default uniform block) is specified by the value of the implementation-dependent constant MAX_COMBINED_GEOMETRY_UNIFORM_COMPONENTS. These values represent the numbers of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a geometry shader. A uniform matrix in the default uniform block with single- or double-precision components will consume no more than $4 \times \min(r, c)$ or $8 \times \min(r, c)$ uniform components, respectively. A scalar or vector uniform with double-precision components will consume no more than $2n$ components, where $n$ is 1 for scalars, and the component count for vectors. A link error is generated if an attempt is made to utilize more than the space available for geometry shader uniform variables. Uniforms are manipulated as described in section 2.11.7. Geometry shaders also have access
to samplers to perform texturing operations, as described in section 2.11.9.

Geometry shaders can access the transformed attributes of all vertices for their input primitive type using input variables. A vertex shader writing to output variables generates the values of these input variables. Values for any inputs that are not written by a vertex shader are undefined. Additionally, a geometry shader has access to a built-in input that holds the ID of the current primitive. This ID is generated by the primitive assembly stage that sits in between the vertex and geometry shader.

Additionally, geometry shaders can write to one or more output variables for each vertex they output. These values are optionally flatshaded (using the OpenGL Shading Language qualifier `flat`) and clipped, then the clipped values interpolated across the primitive (if not flatshaded). The results of these interpolations are available to the fragment shader.

### 2.13.4 Geometry Shader Execution Environment

If there is an active program for the geometry stage, the executable version of the program’s geometry shader is used to process primitives resulting from the primitive assembly stage.

The following operations are applied to the primitives that are the result of executing a geometry shader:

- Perspective division on clip coordinates (section 2.14).
- Viewport mapping, including depth range scaling (section 2.14.1).
- Flatshading (section 2.19).
- Clipping, including client-defined half-spaces (section 2.20).
- Front face determination (section 3.6.1).
- Generic attribute clipping (section 2.20.1).

There are several special considerations for geometry shader execution described in the following sections.

### Texture Access

The **Shader Only Texturing** subsection of section 2.11.12 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to geometry shaders.
2.13. GEOMETRY SHADERS

Instanced Geometry Shaders

For each input primitive received by the geometry shader pipeline stage, the geometry shader may be run once or multiple times. The number of times a geometry shader should be executed for each input primitive may be specified using a layout qualifier in a geometry shader of a linked program. If the invocation count is not specified in any layout qualifier, the invocation count will be one.

Each separate geometry shader invocation is assigned a unique invocation number. For a geometry shader with \( N \) invocations, each input primitive spawns \( N \) invocations, numbered 0 through \( N - 1 \). The built-in uniform \texttt{gl_InvocationID} may be used by a geometry shader invocation to determine its invocation number.

When executing instanced geometry shaders, the output primitives generated from each input primitive are passed to subsequent pipeline stages using the shader invocation number to order the output. The first primitives received by the subsequent pipeline stages are those emitted by the shader invocation numbered zero, followed by those from the shader invocation numbered one, and so forth. Additionally, all output primitives generated from a given input primitive are passed to subsequent pipeline stages before any output primitives generated from subsequent input primitives.

Geometry Shader Vertex Streams

Geometry shaders may emit primitives to multiple independent vertex streams. Each vertex emitted by the geometry shader is directed at one of the vertex streams. As vertices are received on each stream, they are arranged into primitives of the type specified by the geometry shader output primitive type. The shading language built-in functions \texttt{EndPrimitive} and \texttt{EndStreamPrimitive} may be used to end the primitive being assembled on a given vertex stream and start a new empty primitive of the same type. If an implementation supports \( N \) vertex streams, the individual streams are numbered 0 through \( N - 1 \). There is no requirement on the order of the streams to which vertices are emitted, and the number of vertices emitted to each stream may be completely independent, subject only to implementation-dependent output limits.

The primitives emitted to all vertex streams are passed to the transform feedback stage to be captured and written to buffer objects in the manner specified by the transform feedback state. The primitives emitted to all streams but stream zero are discarded after transform feedback. Primitives emitted to stream zero are passed to subsequent pipeline stages for clipping, rasterization, and subsequent fragment processing.

Geometry shaders that emit vertices to multiple vertex streams are currently
limited to using only the `points` output primitive type. A program will fail to link if it includes a geometry shader that calls the `EmitStreamVertex` built-in function and has any other output primitive type parameter.

**Geometry Shader Inputs**

Section 7.1 of the OpenGL Shading Language Specification describes the built-in variable array `gl_in[]` available as input to a geometry shader. `gl_in[]` receives values from equivalent built-in output variables written by the vertex shader, and each array element of `gl_in[]` is a structure holding values for a specific vertex of the input primitive. The length of `gl_in[]` is determined by the geometry shader input type (see section 2.13.1). The members of each element of the `gl_in[]` array are:

- Structure member `gl_ClipDistance[]` holds the per-vertex array of clip distances, as written by the vertex shader to its built-in output variable `gl_ClipDistance[]`.
- Structure member `gl_PointSize` holds the per-vertex point size written by the vertex shader to its built-in output variable `gl_PointSize`. If the vertex shader does not write `gl_PointSize`, the value of `gl_PointSize` is undefined, regardless of the value of the enable `PROGRAM_POINT_SIZE`.
- Structure member `gl_Position` holds the per-vertex position, as written by the vertex shader to its built-in output variable `gl_Position`. Note that writing to `gl_Position` from either the vertex or geometry shader is optional (also see section 7.1 of the OpenGL Shading Language Specification).

Geometry shaders also have available the built-in input variable `gl_PrimitiveIDIn`, which is not an array and has no vertex shader equivalent. It is filled with the number of primitives processed by the drawing command which generated the input vertices. The first primitive generated by a drawing command is numbered zero, and the primitive ID counter is incremented after every individual point, line, or triangle primitive is processed. For triangles drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may eventually be drawn. Restarting a primitive topology using the primitive restart index has no effect on the primitive ID counter.

Similarly to the built-in inputs, each user-defined input has a value for each vertex and thus needs to be declared as arrays or inside input blocks declared as arrays. Declaring an array size is optional. If no size is specified, it will be inferred by the linker from the input primitive type. If a size is specified, it must match the
number of vertices for the input primitive type; otherwise, a link error will occur. The OpenGL Shading Language doesn’t support multi-dimensional arrays; therefore, user-defined geometry shader inputs corresponding to vertex shader outputs declared as arrays must be declared as array members of an input block that is itself declared as an array. See sections 4.3.6 and 7.6 of the OpenGL Shading Language Specification for more information.

Similarly to the limit on vertex shader output components (see section 2.11.11), there is a limit on the number of components of input variables that can be read by the geometry shader, given by the value of the implementation-dependent constant `MAX_GEOMETRY_INPUT_COMPONENTS`.

When a program is linked, all components of any input read by a geometry shader will count against this limit. A program whose geometry shader exceeds this limit may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Component counting rules for different variable types and variable declarations are the same as for `MAX_VERTEX_OUTPUT_COMPONENTS`. (see section 2.11.11).

Geometry Shader Outputs

A geometry shader is limited in the number of vertices it may emit per invocation. The maximum number of vertices a geometry shader can possibly emit is specified in the geometry shader source and may be queried after linking by calling `GetProgramiv` with the symbolic constant `GEOMETRY_VERTICES_OUT`. If a single invocation of a geometry shader emits more vertices than this value, the emitted vertices may have no effect.

There are two implementation-dependent limits on the value of `GEOMETRY_VERTICES_OUT`; it may not exceed the value of `MAX_GEOMETRY_OUTPUT_VERTICES`, and the product of the total number of vertices and the sum of all components of all active output variables may not exceed the value of `MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS`. `LinkProgram` will fail if it determines that the total component limit would be violated.

A geometry shader can write to built-in as well as user-defined output variables. These values are expected to be interpolated across the primitive it outputs, unless they are specified to be flat shaded. To enable seamlessly inserting or removing a geometry shader from a program object, the rules, names and types of the built-in and user-defined output variables are the same as for the vertex shader. Refer to section 2.11.11, and sections 4.3.6, 7.1, and 7.6 of the OpenGL Shading Language Specification for more detail.

After a geometry shader emits a vertex, all output variables are undefined, as described in section 8.10 of the OpenGL Shading Language Specification.
2.13. GEOMETRY SHADERS

The built-in output `gl_Position` is intended to hold the homogeneous vertex position. Writing `gl_Position` is optional.

The built-in output `gl_ClipDistance` holds the clip distance used in the clipping stage, as described in section 2.20.

The built-in output `gl_PointSize`, if written, holds the size of the point to be rasterized, measured in pixels.

The built-in output `gl_PrimitiveID` holds the primitive ID counter read by the fragment shader, replacing the value of `gl_PrimitiveID` generated by drawing commands when no geometry shader is active. The geometry shader must write to `gl_PrimitiveID` for the provoking vertex (see section 2.19) of a primitive being generated, or the primitive ID counter read by the fragment shader for that primitive is undefined.

The built-in output `gl_Layer` is used in layered rendering, and discussed further in the next section.

The built-in output `gl_ViewPortIndex` is used to direct rendering to one of several viewports and is discussed further in the next section.

Similarly to the limit on vertex shader output components (see section 2.11.11), there is a limit on the number of components of output variables that can be written by the geometry shader, given by the value of the implementation-dependent constant `MAX_GEOMETRY_OUTPUT_COMPONENTS`.

When a program is linked, all components of any output variable written by a geometry shader will count against this limit. A program whose geometry shader exceeds this limit may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Component counting rules for different variable types and variable declarations are the same as for `MAX_VERTEX_OUTPUT_COMPONENTS`. (see section 2.11.11).

Layer and Viewport Selection

Geometry shaders can be used to render to one of several different layers of cube map textures, three-dimensional textures, or one-or two-dimensional texture arrays. This functionality allows an application to bind an entire complex texture to a framebuffer object, and render primitives to arbitrary layers computed at run time. For example, it can be used to project and render a scene onto all six faces of a cubemap texture in one pass. The layer to render to is specified by writing to the built-in output variable `gl_Layer`. Layered rendering requires the use of framebuffer objects (see section 4.4.7).

Geometry shaders may also select the destination viewport for each output primitive. The destination viewport for a primitive may be selected in the geometry shader by writing to the built-in output variable `gl_ViewPortIndex`. This
2.13. GEOMETRY SHADERS

functionality allows a geometry shader to direct its output to a different viewport for each primitive, or to draw multiple versions of a primitive into several different viewports.

The specific vertex of a primitive that is used to select the rendering layer or viewport index is implementation-dependent and thus portable applications will assign the same layer and viewport index for all vertices in a primitive. The vertex conventions followed for \texttt{gl\_Layer} and \texttt{gl\_ViewportIndex} may be determined by calling \texttt{GetIntegerv} with the symbolic constants \texttt{LAYER\_PROVOKING\_VERTEX} and \texttt{VIEWPORT\_INDEX\_PROVOKING\_VERTEX}, respectively. For either query, if the value returned is \texttt{PROVOKING\_VERTEX}, then vertex selection follows the convention specified by \texttt{ProvokingVertex} (see section 2.19). If the value returned is \texttt{FIRST\_VERTEX\_CONVENTION}, selection is always taken from the first vertex of a primitive. If the value returned is \texttt{LAST\_VERTEX\_CONVENTION}, the selection is always taken from the last vertex of a primitive. If the value returned is \texttt{UNDEFINED\_VERTEX}, the selection is not guaranteed to be taken from any specific vertex in the primitive. The vertex considered the provoking vertex for particular primitive types is given in table 2.15.

**Primitive Type Mismatches and Drawing Commands**

A geometry shader will fail to execute if a mismatch exists between the type of primitive being drawn and the input primitive type of the shader. If it cannot be executed then no fragments will be rendered, and the error \texttt{INVALID\_OPERATION} will be generated.

This error is generated by any command that transfers vertices to the GL if a geometry shader is active and:

- the input primitive type of the current geometry shader is \texttt{POINTS} and \texttt{mode} is not \texttt{POINTS};
- the input primitive type of the current geometry shader is \texttt{LINES} and \texttt{mode} is not \texttt{LINES}, \texttt{LINE\_STRIP}, or \texttt{LINE\_LOOP};
- the input primitive type of the current geometry shader is \texttt{TRIANGLES} and \texttt{mode} is not \texttt{TRIANGLES}, \texttt{TRIANGLE\_STRIP} or \texttt{TRIANGLE\_FAN};
- the input primitive type of the current geometry shader is \texttt{LINES\_-\_ADJACENCY} and \texttt{mode} is not \texttt{LINES\_-\_ADJACENCY} or \texttt{LINE\_STRIP\_-\_ADJACENCY}; or,
2.14 Coordinate Transformations

Clip coordinates for a vertex result from vertex or, if active, geometry shader execution, which yields a vertex coordinate \( \text{gl\_Position} \). Perspective division on clip coordinates yields normalized device coordinates, followed by a viewport transformation to convert these coordinates into window coordinates.

If a vertex in clip coordinates is given by
\[
\begin{pmatrix}
  x_c \\
  y_c \\
  z_c \\
  w_c
\end{pmatrix}
\]

then the vertex’s normalized device coordinates are

\[
\begin{pmatrix}
  x_d \\
  y_d \\
  z_d
\end{pmatrix} = \begin{pmatrix}
  \frac{x_c}{w_c} \\
  \frac{y_c}{w_c} \\
  \frac{z_c}{w_c}
\end{pmatrix}.
\]

2.14.1 Controlling the Viewport

The viewport transformation is determined by the selected viewport’s width and height in pixels, \( p_x \) and \( p_y \), respectively, and its center \((o_x, o_y)\) (also in pixels).

The vertex’s window coordinates, \( \begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} \), are given by

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

Multiple viewports are available and are numbered zero through the value of \( \text{MAX\_VIEWPORTS} \) minus one. If a geometry shader is active and writes to \( \text{gl\_ViewportIndex} \), the viewport transformation uses the viewport corresponding to the value assigned to \( \text{gl\_ViewportIndex} \) taken from an implementation-dependent primitive vertex. If the value of the viewport index is outside the range zero to the value of \( \text{MAX\_VIEWPORTS} \) minus one, the results of the viewport transformation are undefined. If no geometry shader is active, or if the active geometry shader does not write to \( \text{gl\_ViewportIndex} \), the viewport numbered zero is used by the viewport transformation.
A single vertex may be used in more than one individual primitive, in primitives such as `TRIANGLE_STRIP`. In this case, the viewport transformation is applied separately for each primitive.

The factor and offset applied to $z_d$ for each viewport encoded by $n$ and $f$ are set using

```c
void DepthRangeArrayv(uint first, sizei count, const double *v);
void DepthRangeIndexed(uint index, double n, double f);
void DepthRange(double n, double f);
void DepthRangef(float n, float f);
```

`DepthRangeArrayv` is used to specify the depth range for multiple viewports simultaneously. `first` specifies the index of the first viewport to modify and `count` specifies the number of viewports. If $(first + count)$ is greater than the value of `MAX_VIEWPORTS` then an INVALID_VALUE error will be generated. Viewports whose indices lie outside the range $[first, first + count)$ are not modified. The `v` parameter contains the address of an array of `double` types specifying near ($n$) and far ($f$) for each viewport in that order.

`DepthRangeIndexed` specifies the depth range for a single viewport and is equivalent (assuming no errors are generated) to:

```c
double v[] = { n, f };
DepthRangeArrayv(index, 1, v);
```

`DepthRange` sets the depth range for all viewports to the same values and is equivalent (assuming no errors are generated) to:

```c
for (uint i = 0; i < MAX_VIEWPORTS; i++)
    DepthRangeIndexed(i, n, f);
```

$z_w$ may be represented using either a fixed-point or floating-point representation. However, a floating-point representation must be used if the draw framebuffer has a floating-point depth buffer. If an $m$-bit fixed-point representation is used, we assume that it represents each value $k/(2^m - 1)$, where $k \in \{0, 1, \ldots, 2^m - 1\}$, as $k$ (e.g. 1.0 is represented in binary as a string of all ones). If a fixed-point representation is used, the parameters $n$ and $f$ are clamped to the range $[0, 1]$ when computing $z_w$.

Viewport transformation parameters are specified using

OpenGL 4.2 (Core Profile) - April 27, 2012
2.14. COORDINATE TRANSFORMATIONS

void ViewportArrayv(uint first, sizei count, const float *v);
void ViewportIndexedf(uint index, float x, float y, float w, float h);
void ViewportIndexedfv(uint index, const float *v);
void Viewport(int x, int y, sizei w, sizei h);

ViewportArrayv specifies parameters for multiple viewports simultaneously. first specifies the index of the first viewport to modify and count specifies the number of viewports. If first + count is greater than the value of MAX_VIEWPORTS then an INVALID_VALUE error will be generated. Viewports whose indices lie outside the range [first, first + count) are not modified. v contains the address of an array of floating-point values specifying the left (x), bottom (y), width (w) and height (h) of each viewport, in that order. x and y give the location of the viewport’s lower left corner and w and h give the viewport’s width and height, respectively.

ViewportIndexedf and ViewportIndexedfv specify parameters for a single viewport and are equivalent (assuming no errors are generated) to:

float v[4] = { x, y, w, h };
ViewportArrayv(index, 1, v);

and

ViewportArrayv(index, 1, v);

respectively.

Viewport sets the parameters for all viewports to the same values and is equivalent (assuming no errors are generated) to:

for (uint i = 0; i < MAX_VIEWPORTS; i++)
    ViewportIndexedf(i, 1, (float)x, (float)y, (float)w, (float)h);

The viewport parameters shown in the above equations are found from these values as

\[ o_x = x + \frac{w}{2} \]
\[ o_y = y + \frac{h}{2} \]
\[ p_x = w \]
\[ p_y = h. \]

The location of the viewport’s bottom-left corner, given by (x, y), are clamped to be within the implementation-dependent viewport bounds range. The viewport
bounds range \([min, max]\) tuple may be determined by calling \texttt{GetFloatv} with the symbolic constant \texttt{VIEWPORT_BOUNDS_RANGE} (see section 6.1).

Viewport width and height are clamped to implementation-dependent maximums when specified. The maximum width and height may be found by calling \texttt{GetFloatv} with the symbolic constant \texttt{MAX_VIEWPORT_DIMS}. The maximum viewport dimensions must be greater than or equal to the larger of the visible dimensions of the display being rendered to (if a display exists), and the largest renderbuffer image which can be successfully created and attached to a framebuffer object (see chapter 4). \texttt{INVALID_VALUE} is generated if either \(w\) or \(h\) is negative.

The state required to implement the viewport transformation is four integers and two clamped floating-point values for each viewport. In the initial state, \(w\) and \(h\) for each viewport are set to the width and height, respectively, of the window into which the GL is to do its rendering. If the default framebuffer is bound but no default framebuffer is associated with the GL context (see chapter 4), then \(w\) and \(h\) are initially set to zero. \(o_x\), \(o_y\), \(n\), and \(f\) are set to \(\frac{w}{2}\), \(\frac{h}{2}\), 0.0, and 1.0, respectively.

The precision with which the GL interprets the floating-point viewport bounds is implementation-dependent and may be determined by querying the implementation-defined constant \texttt{VIEWPORT_SUBPIXEL_BITS}.

### 2.15 Asynchronous Queries

Asynchronous queries provide a mechanism to return information about the processing of a sequence of GL commands. There are four query types supported by the GL. Primitive queries with a target of \texttt{PRIMITIVES_GENERATED} (see section 2.18) return information on the number of primitives processed by the GL. Primitive queries with a target of \texttt{TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN} (see section 2.18) return information on the number of primitives written to one or more buffer objects. Occlusion queries (see section 4.1.6) count the number of fragments or samples that pass the depth test, or set a boolean to true when any fragments or samples pass the depth test. Timer queries (see section 5.1) record the amount of time needed to fully process these commands or the current time of the GL.

The results of asynchronous queries are not returned by the GL immediately after the completion of the last command in the set; subsequent commands can be processed while the query results are not complete. When available, the query results are stored in an associated query object. The commands described in section 6.1.7 provide mechanisms to determine when query results are available and return the actual results of the query. The name space for query objects is the unsigned integers, with zero reserved by the GL.

OpenGL 4.2 (Core Profile) - April 27, 2012
Each type of query supported by the GL has an active query object name. If the active query object name for a query type is non-zero, the GL is currently tracking the information corresponding to that query type and the query results will be written into the corresponding query object. If the active query object for a query type name is zero, no such information is being tracked.

A query object is created and made active by calling

```c
void BeginQuery( enum target, uint id );
```

target indicates the type of query to be performed; valid values of target are defined in subsequent sections. If id is an unused query object name, the name is marked as used and associated with a new query object of the type specified by target. Otherwise id must be the name of an existing query object of that type.

BeginQuery fails and an INVALID_OPERATION error is generated if id is not a name returned from a previous call to GenQueries, or if such a name has since been deleted with DeleteQueries.

BeginQuery sets the active query object name for the query type given by target to id. If BeginQuery is called with an id of zero, if the active query object name for target is non-zero (for the targets SAMPLES_PASSED and ANY_SAMPLES_PASSED, if the active query for either target is non-zero), if id is the name of an existing query object whose type does not match target, if id is the active query object name for any query type, or if id is the active query object for conditional rendering (see section 2.16), the error INVALID_OPERATION is generated.

Query targets also support multiple indexed queries. A query object may be created and made active on an indexed query target by calling:

```c
void BeginQueryIndexed( enum target, uint index, uint id );
```

target indicates the type of query to be performed as in BeginQuery. index is the index of the query and must be between 0 and a target-specific maximum. If index is outside of this range, the error INVALID_VALUE is generated. The number of indexed queries supported by specific targets is one, unless indicated otherwise in following sections. Calling BeginQuery is equivalent to calling BeginQueryIndexed with index set to zero.

The command

```c
void EndQuery( enum target );
```
marks the end of the sequence of commands to be tracked for the query type given by target. The active query object for target is updated to indicate that query results
are not available, and the active query object name for \textit{target} is reset to zero. When the commands issued prior to \texttt{EndQuery} have completed and a final query result is available, the query object active when \texttt{EndQuery} is called is updated by the GL. The query object is updated to indicate that the query results are available and to contain the query result. If the active query object name for \textit{target} is zero when \texttt{EndQuery} is called, the error \texttt{INVALID\_OPERATION} is generated.

The command

\begin{verbatim}
void EndQueryIndexed( enum target, uint index );
\end{verbatim}

may be used to mark the end of the query currently active at index \textit{index} of \textit{target}, and must be between zero and the \textit{target}-specific maximum. If \textit{index} is outside of this range, the error \texttt{INVALID\_VALUE} is generated. Calling \texttt{EndQuery} is equivalent to calling \texttt{EndQueryIndexed} with \textit{index} set to zero.

The command

\begin{verbatim}
void GenQueries( sizei n, uint *ids );
\end{verbatim}

returns \textit{n} previously unused query object names in \textit{ids}. These names are marked as used, for the purposes of \texttt{GenQueries} only, but no object is associated with them until the first time they are used by \texttt{BeginQuery}, \texttt{BeginQueryIndexed}, or \texttt{QueryCounter} (see section 5.1).

Query objects are deleted by calling

\begin{verbatim}
void DeleteQueries( sizei n, const uint *ids );
\end{verbatim}

\textit{ids} contains \textit{n} names of query objects to be deleted. After a query object is deleted, its name is again unused. Unused names in \textit{ids} are silently ignored. If an active query object is deleted its name immediately becomes unused, but the underlying object is not deleted until it is no longer active (see section D.1). Unused names in \textit{ids} that have been marked as used for the purposes of \texttt{GenQueries} are marked as unused again. Unused names in \textit{ids} are silently ignored, as is the value zero.

Query objects contain two pieces of state: a single bit indicating whether a query result is available, and an integer containing the query result value. The number of bits, \textit{n}, used to represent the query result is implementation-dependent and may be determined as described in section 6.1.7. In the initial state of a query object, the result is not available (the flag is \texttt{FALSE}), and the result value is zero.

If the query result overflows (exceeds the value $2^n - 1$), its value becomes undefined. It is recommended, but not required, that implementations handle this overflow case by saturating at $2^n - 1$ and incrementing no further.
2.16  CONDITIONAL RENDERING

The necessary state for each query type is an unsigned integer holding the active query object name (zero if no query object is active), and any state necessary to keep the current results of an asynchronous query in progress. Only a single type of occlusion query can be active at one time, so the required state for occlusion queries is shared.

2.16 Conditional Rendering

Conditional rendering can be used to discard rendering commands based on the result of an occlusion query. Conditional rendering is started and stopped using the commands

```c
void BeginConditionalRender( uint id, enum mode );
void EndConditionalRender( void );
```

`id` specifies the name of an occlusion query object whose results are used to determine if the rendering commands are discarded. If the result (SAMPLES_PASSED) of the query is zero, or if the result (ANY_SAMPLES_PASSED) is false, all rendering commands between `BeginConditionalRender` and the corresponding `EndConditionalRender` are discarded. In this case, all drawing commands (see section 2.8.3), as well as `Clear` and `ClearBuffer*` (see section 4.2.3), have no effect. The effect of commands setting current vertex state, such as `VertexAttrib`, are undefined. If the result (SAMPLES_PASSED) of the query is non-zero, or if the result (ANY_SAMPLES_PASSED) is true, such commands are not discarded.

`mode` specifies how `BeginConditionalRender` interprets the results of the occlusion query given by `id`. If `mode` is QUERY_WAIT, the GL waits for the results of the query to be available and then uses the results to determine if subsequent rendering commands are discarded. If `mode` is QUERY_NO_WAIT, the GL may choose to unconditionally execute the subsequent rendering commands without waiting for the query to complete.

If `mode` is QUERY_BY_REGION_WAIT, the GL will also wait for occlusion query results and discard rendering commands if the result of the occlusion query is zero. If the query result is non-zero, subsequent rendering commands are executed, but the GL may discard the results of the commands for any region of the framebuffer that did not contribute to the sample count in the specified occlusion query. Any such discarding is done in an implementation-dependent manner, but the rendering command results may not be discarded for any samples that contributed to the occlusion query sample count. If `mode` is QUERY_BY_REGION_NO_WAIT, the GL operates as in QUERY_BY_REGION_WAIT, but may choose to uncondition-
2.17. TRANSFORM FEEDBACK

ally execute the subsequent rendering commands without waiting for the query to complete.

If BeginConditionalRender is called while conditional rendering is in progress, the error INVALID_OPERATION is generated. If id is not the name of an existing query object, the error INVALID_VALUE is generated. If id is the name of a query object with a target other than SAMPLES_PASSED or ANY_SAMPLES_PASSED, or if id is the name of a query currently in progress, the error INVALID_OPERATION is generated. If EndConditionalRender is called while conditional rendering is not in progress, the error INVALID_OPERATION is generated.

2.17 Transform Feedback

In transform feedback mode, attributes of the vertices of transformed primitives passed to the transform feedback stage are written out to one or more buffer objects. The vertices are fed back before flatshading and clipping. The transformed vertices may be optionally discarded after being stored into one or more buffer objects, or they can be passed on down to the clipping stage for further processing. The set of attributes captured is determined when a program is linked.

The data captured in transform feedback mode depends on the active programs on each of the shader stages. If a program is active for the geometry shader stage, transform feedback captures the vertices of each primitive emitted by the geometry shader. Otherwise, if a program is active for the tessellation evaluation shader stage, transform feedback captures each primitive produced by the tessellation primitive generator, whose vertices are processed by the tessellation evaluation shader. Otherwise, transform feedback captures each primitive processed by the vertex shader.

If separable program objects are in use, the set of attributes captured is taken from the program object active on the last shader stage processing the primitives captured by transform feedback. The set of attributes to capture in transform feedback mode for any other program active on a previous shader stage is ignored.

2.17.1 Transform Feedback Objects

The set of buffer objects used to capture vertex attributes and related state are stored in a transform feedback object. The set of attributes captured in transform feedback mode is determined using the state of the active program object. The name space for transform feedback objects is the unsigned integers. The name zero designates the default transform feedback object.

The command

OpenGL 4.2 (Core Profile) - April 27, 2012
2.17. TRANSFORM FEEDBACK

```c
void GenTransformFeedbacks(sizei n, uint *ids);
```

returns \( n \) previously unused transform feedback object names in \( ids \). These names are marked as used, for the purposes of `GenTransformFeedbacks` only, but they acquire transform feedback state only when they are first bound.

Transform feedback objects are deleted by calling

```c
void DeleteTransformFeedbacks(sizei n, const uint *ids);
```

\( ids \) contains \( n \) names of transform feedback objects to be deleted. After a transform feedback object is deleted it has no contents, and its name is again unused.

Unused names in \( ids \) that have been marked as used for the purposes of `GenTransformFeedbacks` are marked as unused again. Unused names in \( ids \) are silently ignored, as is the value zero. The default transform feedback object cannot be deleted. If an active transform feedback object is deleted its name immediately becomes unused, but the underlying object is not deleted until it is no longer active (see section D.1).

The error `INVALID_OPERATION` is generated by `DeleteTransformFeedbacks` if the transform feedback operation for any object named by \( ids \) is currently active.

A transform feedback object is created by binding a name returned by `GenTransformFeedbacks` with the command

```c
void BindTransformFeedback(enum target, uint id);
```

target must be `TRANSFORM_FEEDBACK` and \( id \) is the transform feedback object name. The resulting transform feedback object is a new state vector, comprising all the state and with the same initial values listed in table 6.43. Additionally, the new object is bound to the GL state vector and is used for subsequent transform feedback operations.

`BindTransformFeedback` can also be used to bind an existing transform feedback object to the GL state for subsequent use. If the bind is successful, no change is made to the state of the newly bound transform feedback object and any previous binding to target is broken.

While a transform feedback buffer object is bound, GL operations on the target to which it is bound affect the bound transform feedback object, and queries of the target to which a transform feedback object is bound return state from the bound object. When buffer objects are bound for transform feedback, they are attached to the currently bound transform feedback object. Buffer objects are used for transform feedback only if they are attached to the currently bound transform feedback object.
In the initial state, a default transform feedback object is bound and treated as a transform feedback object with a name of zero. That object is bound any time `BindTransformFeedback` is called with `id` of zero.

The error INVALID_OPERATION is generated by `BindTransformFeedback` if the transform feedback operation is active on the currently bound transform feedback object, and that operation is not paused (as described below).

`BindTransformFeedback` fails and an INVALID_OPERATION error is generated if `id` is not zero or a name returned from a previous call to `GenTransformFeedbacks`, or if such a name has since been deleted with `DeleteTransformFeedback`.

### 2.17.2 Transform Feedback Primitive Capture

Transform feedback for the currently bound transform feedback object is started and finished by calling

```c
void BeginTransformFeedback( enum primitiveMode );
```

and

```c
void EndTransformFeedback( void );
```

respectively. Transform feedback is said to be active after a call to `BeginTransformFeedback` and inactive after a call to `EndTransformFeedback`. `EndTransformFeedback` first performs an implicit `ResumeTransformFeedback` (see below) if transform feedback is active and not paused. `primitiveMode` is one of TRIANGLES, LINES, or POINTS, and specifies the output type of primitives that will be recorded into the buffer objects bound for transform feedback (see below). `primitiveMode` restricts the primitive types that may be rendered while transform feedback is active, as shown in table 2.14.

Transform feedback commands must be paired; the error INVALID_OPERATION is generated by `BeginTransformFeedback` if transform feedback is active for the current transform feedback object, and by `EndTransformFeedback` if transform feedback is inactive. Transform feedback is initially inactive.

Transform feedback operations for the currently bound transform feedback object may be paused and resumed by calling

```c
void PauseTransformFeedback( void );
```

and
2.17. TRANSFORM FEEDBACK

<table>
<thead>
<tr>
<th>Transform Feedback primitiveMode</th>
<th>Allowed render primitive modes</th>
</tr>
</thead>
<tbody>
<tr>
<td>POINTS</td>
<td>POINTS</td>
</tr>
<tr>
<td>LINES</td>
<td>LINES, LINE_LOOP, LINE_STRIP</td>
</tr>
<tr>
<td>TRIANGLES</td>
<td>TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN</td>
</tr>
</tbody>
</table>

Table 2.14: Legal combinations of the transform feedback primitive mode, as passed to `BeginTransformFeedback`, and the current primitive mode.

```c
void ResumeTransformFeedback(void);
```

respectively. When transform feedback operations are paused, transform feedback is still considered active and changing most transform feedback state related to the object results in an error. However, a new transform feedback object may be bound while transform feedback is paused. The error `INVALID_OPERATION` is generated by `PauseTransformFeedback` if the currently bound transform feedback is not active or is paused. The error `INVALID_OPERATION` is generated by `ResumeTransformFeedback` if the currently bound transform feedback is not active or is not paused.

When transform feedback is active and not paused, all geometric primitives generated must be compatible with the value of `primitiveMode` passed to `BeginTransformFeedback`. The error `INVALID_OPERATION` is generated by `DrawArrays` and the other drawing commands defined in section 2.8.3 if `mode` is not one of the allowed modes in table 2.14. If a tessellation evaluation or geometry shader is active, the type of primitive emitted by that shader is used instead of of the `mode` parameter passed to drawing commands for the purposes of this error check. If tessellation evaluation and geometry shaders are both active, the output primitive type of the geometry shader will be used for the purposes of this error. Any primitive type may be used while transform feedback is paused.

Transform feedback mode captures the values of output variables written by the vertex shader (or, if active, geometry shader).

Regions of buffer objects are bound as the targets of transform feedback by calling one of the commands `BindBufferRange` or `BindBufferBase` (see section 2.9.1) with `target` set to `TRANSFORM_FEEDBACK_BUFFER`. In addition to the general errors described in section 2.9.1, `BindBufferRange` will generate an `INVALID_VALUE` error if `index` is greater than or equal to the value of `MAX_TRANSFORM_FEEDBACK_BUFFERS`, or if either `offset` or `size` is not a multiple of 4.
When an individual point, line, or triangle primitive reaches the transform feedback stage while transform feedback is active and not paused, the values of the specified output variables of the vertex are appended to the buffer objects bound to the transform feedback binding points. The attributes of the first vertex received after `BeginTransformFeedback` are written at the starting offsets of the bound buffer objects set by `BindBufferRange`, and subsequent vertex attributes are appended to the buffer object. When capturing line and triangle primitives, all attributes of the first vertex are written first, followed by attributes of the subsequent vertices.

When writing output variables that are arrays, individual array elements are written in order. For multi-component output variables, elements of output arrays, or transformed vertex attributes, the individual components are written in order. The value for any attribute specified to be streamed to a buffer object but not actually written by a vertex or geometry shader is undefined. The results of appending an output variable to a transform feedback buffer are undefined if any component of that variable would be written at an offset not aligned to the size of the component.

When transform feedback is paused, no vertices are recorded. When transform feedback is resumed, subsequent vertices are appended to the bound buffer objects immediately following the last vertex written before transform feedback was paused.

Individual lines or triangles of a strip or fan primitive will be extracted and recorded separately. Incomplete primitives are not recorded.

Transform feedback can operate in either `INTERLEAVED_ATTRIBS` or `SEPARATE_ATTRIBS` mode.

In `INTERLEAVED_ATTRIBS` mode, the values of one or more output variables written by a vertex or geometry shader are written, interleaved, into the buffer objects bound to one or more transform feedback binding points. The list of outputs provided for capture in interleaved mode may include special separator values, which can be used to direct subsequent outputs to the next binding point. Each non-separator output is written to the binding point numbered $n$, where $n$ is the number of separator values preceding it in the list. If more than one output variable is written to a buffer object, they will be recorded in the order specified by `TransformFeedbackVaryings` (see section 2.11.11).

In `SEPARATE_ATTRIBS` mode, the first output variable or transformed vertex attribute specified by `TransformFeedbackVaryings` is written to the first transform feedback binding point; subsequent output variables are written to the subsequent transform feedback binding points. The total number of variables that may be captured in separate mode is given by `MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`.

When using a geometry shader or program that writes vertices to multiple vertex streams, each vertex emitted may trigger a new primitive in the vertex stream to
which it was emitted. If transform feedback is active, the outputs of the primitive are written to a transform feedback binding point if and only if the outputs directed at that binding point belong to the vertex stream in question. All outputs assigned to a given binding point are required to come from a single vertex stream.

If recording the vertices of a primitive to the buffer objects being used for transform feedback purposes would result in either exceeding the limits of any buffer object’s size, or in exceeding the end position \( \text{offset} + \text{size} - 1 \), as set by `BindBufferRange`, then no vertices of that primitive are recorded in any buffer object, and the counter corresponding to the asynchronous query target `TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN` (see section 2.18) is not incremented. For the purposes of this test, `gl_SkipComponents` variables are counted as recording data to a buffer object.

Transform feedback binding points zero through \( \text{count} \) minus one must have buffer objects bound when `BeginTransformFeedback` is called, where \( \text{count} \) is the parameter passed to `TransformFeedbackVaryings` in separate mode, or one more than the number of `gl_NextBuffer` elements in the `varyings` parameter to `TransformFeedbackVaryings` in interleaved mode. The error `INVALID_OPERATION` is generated by `BeginTransformFeedback` if any of these binding points does not have a buffer object bound. The error `INVALID_OPERATION` is also generated by `BeginTransformFeedback` if no binding points would be used, either because no program object is active or because the active program object has specified no output variables to record.

When `BeginTransformFeedback` is called with an active program object containing a vertex or geometry shader, the set of output variables captured during transform feedback is taken from the active program object and may not be changed while transform feedback is active. That program object must be active until the `EndTransformFeedback` is called, except while the transform feedback object is paused. The error `INVALID_OPERATION` is generated:

- by `UseProgram` if the current transform feedback object is active and not paused;
- by `UseProgramStages` if the program pipeline object it refers to is current and the current transform feedback object is active and not paused;
- by `BindProgramPipeline` if the current transform feedback object is active and not paused;
- by `LinkProgram` if `program` is the name of a program being used by one or more transform feedback objects, even if the objects are not currently bound or are paused;
2.17. TRANSFORM FEEDBACK

- by **ResumeTransformFeedback** if the program object being used by the current transform feedback object is not active, or has been re-linked since transform feedback became active for the current transform feedback object.

- by **ResumeTransformFeedback** if the program pipeline object being used by the current transform feedback object is not bound, if any of its shader stage bindings has changed, or if a single program object is active and overriding it; and

- by **BindBufferRange** or **BindBufferBase** if target is **TRANSFORM_FEEDBACK_BUFFER** and transform feedback is currently active.

Buffers should not be bound or in use for both transform feedback and other purposes in the GL. Specifically, if a buffer object is simultaneously bound to a transform feedback buffer binding point and elsewhere in the GL, any writes to or reads from the buffer generate undefined values. Examples of such bindings include **ReadPixels** to a pixel buffer object binding point and client access to a buffer mapped with **MapBuffer**.

However, if a buffer object is written and read sequentially by transform feedback and other mechanisms, it is the responsibility of the GL to ensure that data are accessed consistently, even if the implementation performs the operations in a pipelined manner. For example, **MapBuffer** may need to block pending the completion of a previous transform feedback operation.

### 2.17.3 Transform Feedback Draw Operations

When transform feedback is active, the values of output variables or transformed vertex attributes are captured into the buffer objects attached to the current transform feedback object. After transform feedback is complete, subsequent rendering operations may use the contents of these buffer objects (see section 2.9). The number of vertices captured from each vertex stream during transform feedback is stored in the corresponding transform feedback object and may be used in conjunction with the commands

```cpp
void DrawTransformFeedback( enum mode, uint id );
void DrawTransformFeedbackInstanced( enum mode,
                                      uint id, sizei primcount );
void DrawTransformFeedbackStream( enum mode, uint id,
                                       uint stream );
``
void DrawTransformFeedbackStreamInstanced(enum mode, uint id, uint stream, sizei primcount);

to replay the captured vertices.

DrawTransformFeedbackStreamInstanced is equivalent to calling DrawArraysInstanced with mode as specified, first set to zero, count set to the number of vertices captured from the vertex stream numbered stream the last time transform feedback was active on the transform feedback object named id, and primcount as specified.

Calling DrawTransformFeedbackInstanced is equivalent to calling DrawTransformFeedbackStreamInstanced with stream set to zero.

Calling DrawTransformFeedbackStream is equivalent to calling DrawTransformFeedbackStreamInstanced with primcount set to one.

Finally, calling DrawTransformFeedback is equivalent to calling DrawTransformFeedbackStreamInstanced with stream set to zero and primcount set to one.

The error INVALID_VALUE is generated if stream is greater than or equal to the value of MAX_VERTEX_STREAMS. The error INVALID_VALUE is generated if id is not the name of a transform feedback object. The error INVALID_OPERATION is generated if EndTransformFeedback has never been called while the object named by id was bound. No error is generated if the transform feedback object named by id is active; the vertex count used for the rendering operation is set by the previous EndTransformFeedback command.

Note that the vertex count is from the number of vertices recorded to the selected vertex stream during the transform feedback operation. If no outputs belonging to the selected vertex stream are recorded, the corresponding vertex count will be zero even if complete primitives were emitted to the selected stream.

2.18 Primitive Queries

Primitive queries use query objects to track the number of primitives in each vertex stream that are generated by the GL and the number of primitives in each vertex stream that are written to buffer objects in transform feedback mode.

When BeginQueryIndexed is called with a target of PRIMITIVES_GENERATED, the primitives generated count maintained by the GL for the vertex stream index is set to zero. There is a separate query and counter for each vertex stream. The number of vertex streams is given by the value of the implementation-dependent constant MAX_VERTEX_STREAMS. If index is not an integer in the range zero to the value of MAX_VERTEX_STREAMS minus one, the error INVALID_
2.19. FLATSHADING

VALUE is generated. When a generated primitive query for a vertex stream is active, the primitives-generated count is incremented every time a primitive emitted to that stream reaches the transform feedback stage (see section 2.17), whether or not transform feedback is active. This counter counts the number of primitives emitted by a geometry shader, if active, possibly further tessellated into separate primitives during the transform feedback stage, if active.

When BeginQueryIndexed is called with a target of TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, the transform feedback primitives written count maintained by the GL for vertex stream index is set to zero. There is a separate query and counter for each vertex stream. If index is not an integer in the range zero to the value of MAX_VERTEX_STREAMS minus one, the error INVALID_VALUE is generated. When a transform feedback primitives written query for a vertex stream is active, the counter for that vertex stream is incremented every time the vertices of a primitive written to that stream are recorded into one or more buffer objects. If transform feedback is not active or if a primitive to be recorded does not fit in a buffer object, the counter is not incremented.

These two types of queries can be used together to determine if all primitives in a given vertex stream have been written to the bound feedback buffers; if both queries are run simultaneously and the query results are equal, all primitives have been written to the buffer(s). If the number of primitives written is less than the number of primitives generated, one or more buffers overflowed.

2.19 Flatshading

Flatshading a vertex shader output means to assign all vertices of the primitive the same value for that output.

The output values assigned are those of the provoking vertex of the primitive. The provoking vertex is controlled with the command

```c
void ProvokingVertex( enum provokeMode );
```

provokeMode must be either FIRST_VERTEX_CONVENTION or LAST_VERTEX_CONVENTION, and controls selection of the vertex whose values are assigned to flatshaded colors and outputs, as shown in table 2.15

If a vertex or geometry shader is active, user-defined output variables may be flatshaded by using the flat qualifier when declaring the output, as described in section 4.3.6 of the OpenGL Shading Language Specification

The state required for flatshading is one bit for the provoking vertex mode, and one implementation-dependent bit for the provoking vertex behavior of quad
2.20. **PRIMITIVE CLIPPING**

Primitives are clipped to the *clip volume*. In clip coordinates, the *view volume* is defined by

\[
-\omega_c \leq x_c \leq \omega_c \\
-\omega_c \leq y_c \leq \omega_c \\
-\omega_c \leq z_c \leq \omega_c.
\]

This view volume may be further restricted by as many as \( n \) client-defined half-spaces. \((n \text{ is an implementation-dependent maximum that must be at least 8.})\) The clip volume is the intersection of all such half-spaces with the view volume (if no client-defined half-spaces are enabled, the clip volume is the view volume).

A vertex shader may write a single clip distance for each supported half-space to elements of the `gl_ClipDistance[]` array. Half-space \( n \) is then given by the set of points satisfying the inequality

![Table 2.15: Provoking vertex selection. The output values used for flatshading the \( i \)th primitive generated by drawing commands with the indicated primitive type are derived from the corresponding values of the vertex whose index is shown in the table. Vertices are numbered 1 through \( n \), where \( n \) is the number of vertices drawn.](image-url)

primitives. The initial value of the provoking vertex mode is `LAST_VERTEX_-CONVENTION`.

**2.20 Primitive Clipping**

<table>
<thead>
<tr>
<th>Primitive type of polygon ( i )</th>
<th>First vertex convention</th>
<th>Last vertex convention</th>
</tr>
</thead>
<tbody>
<tr>
<td>point</td>
<td>( i )</td>
<td>( i )</td>
</tr>
<tr>
<td>independent line</td>
<td>( 2i - 1 )</td>
<td>( 2i )</td>
</tr>
<tr>
<td>line loop</td>
<td>( i )</td>
<td>( i + 1 ), if ( i &lt; n ) ( 1 ), if ( i = n )</td>
</tr>
<tr>
<td>line strip</td>
<td>( i )</td>
<td>( i + 1 )</td>
</tr>
<tr>
<td>independent triangle</td>
<td>( 3i - 2 )</td>
<td>( 3i )</td>
</tr>
<tr>
<td>triangle strip</td>
<td>( i )</td>
<td>( i + 2 )</td>
</tr>
<tr>
<td>triangle fan</td>
<td>( i + 1 )</td>
<td>( i + 2 )</td>
</tr>
<tr>
<td>line adjacency</td>
<td>( 4i - 2 )</td>
<td>( 4i - 1 )</td>
</tr>
<tr>
<td>line strip adjacency</td>
<td>( i + 1 )</td>
<td>( i + 2 )</td>
</tr>
<tr>
<td>triangle adjacency</td>
<td>( 6i - 5 )</td>
<td>( 6i - 1 )</td>
</tr>
<tr>
<td>triangle strip adjacency</td>
<td>( 2i - 1 )</td>
<td>( 2i + 3 )</td>
</tr>
</tbody>
</table>
2.20. PRIMITIVE CLIPPING

\[ c_n(P) \geq 0, \]

where \( c_n(P) \) is the value of clip distance \( n \) at point \( P \). For point primitives, \( c_n(P) \) is simply the clip distance for the vertex in question. For line and triangle primitives, per-vertex clip distances are interpolated using a weighted mean, with weights derived according to the algorithms described in sections 3.5 and 3.6.

Client-defined half-spaces are enabled with the generic \texttt{Enable} command and disabled with the \texttt{Disable} command. The value of the argument to either command is \texttt{CLIP\_DISTANCE}\( i \), where \( i \) is an integer between 0 and \( n - 1 \); specifying a value of \( i \) enables or disables the plane equation with index \( i \). The constants obey \texttt{CLIP\_DISTANCE}\( i \) = \texttt{CLIP\_DISTANCE0} + \( i \).

Depth clamping is enabled with the generic \texttt{Enable} command and disabled with the \texttt{Disable} command. The value of the argument to either command is \texttt{DEPTH\_CLAMP}. If depth clamping is enabled, the

\[-w_c \leq z_c \leq w_c\]

plane equation is ignored by view volume clipping (effectively, there is no near or far plane clipping).

If the primitive under consideration is a point, then clipping passes it unchanged if it lies within the clip volume; otherwise, it is discarded.

If the primitive is a line segment, then clipping does nothing to it if it lies entirely within the clip volume, and discards it if it lies entirely outside the volume.

If part of the line segment lies in the volume and part lies outside, then the line segment is clipped and new vertex coordinates are computed for one or both vertices. A clipped line segment endpoint lies on both the original line segment and the boundary of the clip volume.

This clipping produces a value, \( 0 \leq t \leq 1 \), for each clipped vertex. If the coordinates of a clipped vertex are \( \mathbf{P} \) and the original vertices’ coordinates are \( \mathbf{P}_1 \) and \( \mathbf{P}_2 \), then \( t \) is given by

\[ \mathbf{P} = t\mathbf{P}_1 + (1 - t)\mathbf{P}_2. \]

The value of \( t \) is used to clip vertex shader outputs as described in section 2.20.1.

If the primitive is a polygon, then it is passed if every one of its edges lies entirely inside the clip volume and either clipped or discarded otherwise. Polygon clipping may cause polygon edges to be clipped, but because polygon connectivity must be maintained, these clipped edges are connected by new edges that lie along the clip volume’s boundary. Thus, clipping may require the introduction of new vertices into a polygon.
If it happens that a polygon intersects an edge of the clip volume’s boundary, then the clipped polygon must include a point on this boundary edge.

Primitives rendered with user-defined half-spaces must satisfy a complementarity criterion. Suppose a series of primitives is drawn where each vertex $i$ has a single specified clip distance $d_i$ (or a number of similarly specified clip distances, if multiple half-spaces are enabled). Next, suppose that the same series of primitives are drawn again with each such clip distance replaced by $-d_i$ (and the GL is otherwise in the same state). In this case, primitives must not be missing any pixels, nor may any pixels be drawn twice in regions where those primitives are cut by the clip planes.

The state required for clipping is at least 8 bits indicating which of the client-defined half-spaces are enabled. In the initial state, all half-spaces are disabled.

### 2.20.1 Clipping Shader Outputs

Next, vertex shader outputs are clipped. The output values associated with a vertex that lies within the clip volume are unaffected by clipping. If a primitive is clipped, however, the output values assigned to vertices produced by clipping are clipped.

Let the output values assigned to the two vertices $P_1$ and $P_2$ of an unclipped edge be $c_1$ and $c_2$. The value of $t$ (section 2.20) for a clipped point $P$ is used to obtain the output value associated with $P$ as

$$c = tc_1 + (1-t)c_2.$$  

(Multiplying an output value by a scalar means multiplying each of $x$, $y$, $z$, and $w$ by the scalar.)

Polygon clipping may create a clipped vertex along an edge of the clip volume’s boundary. This situation is handled by noting that polygon clipping proceeds by clipping against one half-space at a time. Output value clipping is done in the same way, so that clipped points always occur at the intersection of polygon edges (possibly already clipped) with the clip volume’s boundary.

For vertex shader outputs specified to be interpolated without perspective correction (using the `noperspective` qualifier), the value of $t$ used to obtain the output value associated with $P$ will be adjusted to produce results that vary linearly in screen space.

Outputs of integer or unsigned integer type must always be declared with the `flat` qualifier. Since such outputs are constant over the primitive being rasterized

---

5 Since this computation is performed in clip space before division by $w_c$, clipped output values are perspective-correct.
(see sections 3.5.1 and 3.6.1), no interpolation is performed.
Chapter 3

Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional image. Each point of this image contains such information as color and depth. Thus, rasterizing a primitive consists of two parts. The first is to determine which squares of an integer grid in window coordinates are occupied by the primitive. The second is assigning a depth value and one or more color values to each such square. The results of this process are passed on to the next stage of the GL (per-fragment operations), which uses the information to update the appropriate locations in the framebuffer. Figure 3.1 diagrams the rasterization process. The color values assigned to a fragment are determined by a fragment shader as defined in section 3.10. The final depth value is initially determined by the rasterization operations and may be modified or replaced by a fragment shader. The results from rasterizing a point, line, or polygon are routed through a fragment shader.

A grid square along with its \(z\) (depth) and shader output parameters is called a fragment; the parameters are collectively dubbed the fragment’s associated data. A fragment is located by its lower left corner, which lies on integer grid coordinates. Rasterization operations also refer to a fragment’s center, which is offset by \((1/2, 1/2)\) from its lower left corner (and so lies on half-integer coordinates).

Grid squares need not actually be square in the GL. Rasterization rules are not affected by the actual aspect ratio of the grid squares. Display of non-square grids, however, will cause rasterized points and line segments to appear fatter in one direction than the other. We assume that fragments are square, since it simplifies antialiasing and texturing.

Several factors affect rasterization. Primitives may be discarded before rasterization. Points may be given differing diameters and line segments differing widths. A point, line segment, or polygon may be antialiased.

Rasterization only produces fragments corresponding to pixels in the frame-
buffer. Fragments which would be produced by application of any of the primitive rasterization rules described below but which lie outside the framebuffer are not produced, nor are they processed by any later stage of the GL, including any of the early per-fragment tests described in section 3.8.

3.1 Discarding Primitives Before Rasterization

Primitives sent to vertex stream zero (see section 2.17) are processed further; primitives emitted to any other stream are discarded. When geometry shaders are disabled, all vertices are considered to be emitted to stream zero.

Primitives can be optionally discarded before rasterization by calling `Enable` and `Disable` with `RASTERIZER_DISCARD`. When enabled, primitives are discarded immediately before the rasterization stage, but after the optional transform feedback stage (see section 2.17). When disabled, primitives are passed through to the rasterization stage to be processed normally. When enabled, `RASTERIZER_DISCARD` also causes the `Clear` and `ClearBuffer*` commands to be ignored.

The state required to control primitive discard is a bit indicating whether discard is enabled or disabled. The initial value of primitive discard is `FALSE`.

OpenGL 4.2 (Core Profile) - April 27, 2012
3.2 Invariance

Consider a primitive $p'$ obtained by translating a primitive $p$ through an offset $(x, y)$ in window coordinates, where $x$ and $y$ are integers. As long as neither $p'$ nor $p$ is clipped, it must be the case that each fragment $f'$ produced from $p'$ is identical to a corresponding fragment $f$ from $p$ except that the center of $f'$ is offset by $(x, y)$ from the center of $f$.

3.3 Antialiasing

The R, G, and B values of the rasterized fragment are left unaffected, but the A value is multiplied by a floating-point value in the range $[0, 1]$ that describes a fragment’s screen pixel coverage. The per-fragment stage of the GL can be set up to use the A value to blend the incoming fragment with the corresponding pixel already present in the framebuffer.

The details of how antialiased fragment coverage values are computed are difficult to specify in general. The reason is that high-quality antialiasing may take into account perceptual issues as well as characteristics of the monitor on which the contents of the framebuffer are displayed. Such details cannot be addressed within the scope of this document. Further, the coverage value computed for a fragment of some primitive may depend on the primitive’s relationship to a number of grid squares neighboring the one corresponding to the fragment, and not just on the fragment’s grid square. Another consideration is that accurate calculation of coverage values may be computationally expensive; consequently we allow a given GL implementation to approximate true coverage values by using a fast but not entirely accurate coverage computation.

In light of these considerations, we chose to specify the behavior of exact antialiasing in the prototypical case that each displayed pixel is a perfect square of uniform intensity. The square is called a fragment square and has lower left corner $(x, y)$ and upper right corner $(x + 1, y + 1)$. We recognize that this simple box filter may not produce the most favorable antialiasing results, but it provides a simple, well-defined model.

A GL implementation may use other methods to perform antialiasing, subject to the following conditions:

1. If $f_1$ and $f_2$ are two fragments, and the portion of $f_1$ covered by some primitive is a subset of the corresponding portion of $f_2$ covered by the primitive, then the coverage computed for $f_1$ must be less than or equal to that computed for $f_2$. 
2. The coverage computation for a fragment \( f \) must be local: it may depend only on \( f \)'s relationship to the boundary of the primitive being rasterized. It may not depend on \( f \)'s \( x \) and \( y \) coordinates.

Another property that is desirable, but not required, is:

3. The sum of the coverage values for all fragments produced by rasterizing a particular primitive must be constant, independent of any rigid motions in window coordinates, as long as none of those fragments lies along window edges.

In some implementations, varying degrees of antialiasing quality may be obtained by providing GL hints (section 5.4), allowing a user to make an image quality versus speed tradeoff.

### 3.3.1 Multisampling

Multisampling is a mechanism to antialias all GL primitives: points, lines, and polygons. The technique is to sample all primitives multiple times at each pixel. The color sample values are resolved to a single, displayable color each time a pixel is updated, so the antialiasing appears to be automatic at the application level. Because each sample includes color, depth, and stencil information, the color (including texture operation), depth, and stencil functions perform equivalently to the single-sample mode.

An additional buffer, called the multisample buffer, is added to the framebuffer. Pixel sample values, including color, depth, and stencil values, are stored in this buffer. Samples contain separate color values for each fragment color. When the framebuffer includes a multisample buffer, it does not include depth or stencil buffers, even if the multisample buffer does not store depth or stencil values. Color buffers do coexist with the multisample buffer, however.

Multisample antialiasing is most valuable for rendering polygons, because it requires no sorting for hidden surface elimination, and it correctly handles adjacent polygons, object silhouettes, and even intersecting polygons. If only lines are being rendered, the “smooth” antialiasing mechanism provided by the base GL may result in a higher quality image. This mechanism is designed to allow multisample and smooth antialiasing techniques to be alternated during the rendering of a single scene.

If the value of \texttt{SAMPLE\_BUFFERS} is one, the rasterization of all primitives is changed, and is referred to as multisample rasterization. Otherwise, primitive rasterization is referred to as single-sample rasterization. The value of \texttt{SAMPLE\_BUFFERS} is queried by calling \texttt{GetIntegerv} with \texttt{pname} set to \texttt{SAMPLE\_BUFFERS}.
3.3. ANTIALIASING

During multisample rendering the contents of a pixel fragment are changed in two ways. First, each fragment includes a coverage value with \texttt{SAMPLES} bits. The value of \texttt{SAMPLES} is an implementation-dependent constant, and is queried by calling \texttt{GetIntegerv} with \texttt{pname} set to \texttt{SAMPLES}.

The location of a given sample is queried with the command

\begin{verbatim}
void GetMultisamplefv(enum pname, uint index, float *val);
\end{verbatim}

\textit{pname} must be \texttt{SAMPLE_POSITION}, and \textit{index} corresponds to the sample for which the location should be returned. The sample location is returned as two floating-point values in \texttt{val[0]} and \texttt{val[1]}, each between 0 and 1, corresponding to the \textit{x} and \textit{y} locations respectively in GL pixel space of that sample. \((0.5, 0.5)\) thus corresponds to the pixel center. The error \texttt{INVALID_VALUE} is generated if \textit{index} is greater than or equal to the value of \texttt{SAMPLES}. If the multisample mode does not have fixed sample locations, the returned values may only reflect the locations of samples within some pixels.

Second, each fragment includes \texttt{SAMPLES} depth values and sets of associated data, instead of the single depth value and set of associated data that is maintained in single-sample rendering mode. An implementation may choose to assign the same associated data to more than one sample. The location for evaluating such associated data can be anywhere within the pixel including the fragment center or any of the sample locations. The different associated data values need not all be evaluated at the same location. Each pixel fragment thus consists of integer \textit{x} and \textit{y} grid coordinates, \texttt{SAMPLES} depth values and sets of associated data, and a coverage value with a maximum of \texttt{SAMPLES} bits.

Multisample rasterization is enabled or disabled by calling \texttt{Enable} or \texttt{Disable} with the symbolic constant \texttt{MULTISAMPLE}.

If \texttt{MULTISAMPLE} is disabled, multisample rasterization of all primitives is equivalent to single-sample (fragment-center) rasterization, except that the fragment coverage value is set to full coverage. The color and depth values and the sets of texture coordinates may all be set to the values that would have been assigned by single-sample rasterization, or they may be assigned as described below for multisample rasterization.

If \texttt{MULTISAMPLE} is enabled, multisample rasterization of all primitives differs substantially from single-sample rasterization. It is understood that each pixel in the framebuffer has \texttt{SAMPLES} locations associated with it. These locations are exact positions, rather than regions or areas, and each is referred to as a sample point. The sample points associated with a pixel may be located inside or outside of the unit square that is considered to bound the pixel. Furthermore, the relative
locations of sample points may be identical for each pixel in the framebuffer, or they may differ.

If `MULTISAMPLE` is enabled and the current program object includes a fragment shader with one or more input variables qualified with `sample in`, the data associated with those variables will be assigned independently. The values for each sample must be evaluated at the location of the sample. The data associated with any other variables not qualified with `sample in` need not be evaluated independently for each sample.

If the sample locations differ per pixel, they should be aligned to window, not screen, boundaries. Otherwise rendering results will be window-position specific. The invariance requirement described in section 3.2 is relaxed for all multisample rasterization, because the sample locations may be a function of pixel location.

**Sample Shading**

Sample shading can be used to specify a minimum number of unique samples to process for each fragment. Sample shading is controlled by calling `Enable` or `Disable` with the symbolic constant `SAMPLE_SHADING`.

If `MULTISAMPLE` or `SAMPLE_SHADING` is disabled, sample shading has no effect. Otherwise, an implementation must provide a minimum of

$$\max\left(\lceil mss \times \text{samples} \rceil, 1 \right)$$

unique color values for each fragment, where `mss` is the value of `MIN_SAMPLE_SHADING_VALUE` and `samples` is the number of samples (the value of `SAMPLES`). These are associated with the samples in an implementation-dependent manner. The value of `MIN_SAMPLE_SHADING_VALUE` is specified by calling

```c
void MinSampleShading( float value );
```

with `value` set to the desired minimum sample shading fraction. `value` is clamped to [0, 1] when specified. The sample shading fraction may be queried by calling `GetFloatv` with the symbolic constant `MIN_SAMPLE_SHADING_VALUE`.

When the sample shading fraction is 1.0, a separate set of colors and other associated data are evaluated for each sample, and each set of values is evaluated at the sample location.

### 3.4 Points

A point is drawn by generating a set of fragments in the shape of a square or circle centered around the vertex of the point. Each vertex has an associated point size that controls the size of that square or circle.
If program point size mode is enabled, the derived point size is taken from the (potentially clipped) shader built-in `gl_PointSize` written by:

- the geometry shader, if active;
- the tessellation evaluation shader, if active and no geometry shader is active;
- the tessellation control shader, if active and no geometry or tessellation evaluation shader is active; or
- the vertex shader, otherwise

and clamped to the implementation-dependent point size range. If the value written to `gl_PointSize` is less than or equal to zero, or if no value was written to `gl_PointSize`, results are undefined. If program point size mode is disabled, the derived point size is specified with the command

```c
void PointSize(float size);
```

`size` specifies the requested size of a point. The default value is 1.0. A value less than or equal to zero results in the error `INVALID_VALUE`. Program point size mode is enabled and disabled by calling `Enable` or `Disable` with the symbolic value `PROGRAM_POINT_SIZE`.

If multisampling is enabled, an implementation may optionally fade the point alpha (see section 3.12) instead of allowing the point width to go below a given threshold. In this case, the width of the rasterized point is

\[
width = \begin{cases} 
\text{derived size} & \text{if derived size} \geq \text{threshold} \\
\text{threshold} & \text{otherwise}
\end{cases}
\]

and the fade factor is computed as follows:

\[
fade = \begin{cases} 
1 & \text{if derived size} \geq \text{threshold} \\
\left(\frac{\text{derived size}}{\text{threshold}}\right)^2 & \text{otherwise}
\end{cases}
\]

The point fade `threshold` is specified with

```c
void PointParameter\{if\}(enum pname, T param);
void PointParameter\{if\}v(enum pname, const T params);
```

If `pname` is `POINT_FADE_THRESHOLD_SIZE`, then `param` specifies, or `params` points to the point fade `threshold`. Values of `POINT_FADE_THRESHOLD_SIZE` less than zero result in the error `INVALID_VALUE`.

OpenGL 4.2 (Core Profile) - April 27, 2012
Data conversions are performed as specified in section 2.3.1.

The point sprite texture coordinate origin is set with the **PointParameter** commands where `pname` is `POINT_SPRITECOORD_ORIGIN` and `param` is `LOWER_LEFT` or `UPPER_LEFT`. The default value is `UPPER_LEFT`.

### 3.4.1 Basic Point Rasterization

Point rasterization produces a fragment for each framebuffer pixel whose center lies inside a square centered at the point’s \((x_w, y_w)\), with side length equal to the current point size.

All fragments produced in rasterizing a point sprite are assigned the same associated data, which are those of the vertex corresponding to the point. However, the fragment shader built-in `gl_PointCoord` contains point sprite texture coordinates. The \(s\) point sprite texture coordinate varies from 0 to 1 across the point horizontally left-to-right. If `POINT_SPRITECOORD_ORIGIN` is `LOWER_LEFT`, the \(t\) coordinate varies from 0 to 1 vertically bottom-to-top. Otherwise if the point sprite texture coordinate origin is `UPPER_LEFT`, the \(t\) coordinate varies from 0 to 1 vertically top-to-bottom. The following formula is used to evaluate the \(s\) and \(t\) point sprite texture coordinates:

\[
s = \frac{1}{2} + \left( \frac{x_f + \frac{1}{2} - x_w}{size} \right)
\]

\[
t = \begin{cases} \frac{1}{2} + \left( \frac{y_f + \frac{1}{2} - y_w}{size} \right), & \text{POINT_SPRITECOORD_ORIGIN = LOWER_LEFT} \\ \frac{1}{2} - \left( \frac{y_f + \frac{1}{2} - y_w}{size} \right), & \text{POINT_SPRITECOORD_ORIGIN = UPPER_LEFT} \end{cases}
\]

where \(size\) is the point’s size, \(x_f\) and \(y_f\) are the (integral) window coordinates of the fragment, and \(x_w\) and \(y_w\) are the exact, unrounded window coordinates of the vertex for the point.

Not all point widths need be supported, but the width 1.0 must be provided. The range of supported widths and the width of evenly-spaced gradations within that range are implementation-dependent. The range and gradations may be obtained using the query mechanism described in chapter 6. If, for instance, the width range is from 0.1 to 2.0 and the gradation width is 0.1, then the widths 0.1, 0.2, ..., 1.9, 2.0 are supported. Additional point widths may also be supported. There is no requirement that these widths must be equally spaced. If an unsupported width is requested, the nearest supported width is used instead.
3.4.2 Point Rasterization State

The state required to control point rasterization consists of the floating-point point width, a bit indicating whether or not vertex program point size mode is enabled, a bit for the point sprite texture coordinate origin, and a floating-point value specifying the point fade threshold size.

3.4.3 Point Multisample Rasterization

If MULTISAMPLE is enabled, and the value of SAMPLE_BUFFERS is one, then points are rasterized using the following algorithm. Point rasterization produces a fragment for each framebuffer pixel with one or more sample points that intersect a region centered at the point’s \((x_w, y_w)\). This region is a square with side equal to the current point width. Coverage bits that correspond to sample points that intersect the region are 1, other coverage bits are 0. All data associated with each sample for the fragment are the data associated with the point being rasterized.

The set of point sizes supported is equivalent to those for point sprites without multisample.

3.5 Line Segments

A line segment results from a line strip, a line loop, or a series of separate line segments. Line segment rasterization is controlled by several variables. Line width, which may be set by calling

```c
void LineWidth( float width );
```

with an appropriate positive floating-point width, controls the width of rasterized line segments. The default width is 1.0. Values less than or equal to 0.0 generate the error INVALID_VALUE. Antialiasing is controlled with Enable and Disable using the symbolic constant LINE_SMOOTH.

3.5.1 Basic Line Segment Rasterization

Line segment rasterization begins by characterizing the segment as either x-major or y-major. x-major line segments have slope in the closed interval \([-1, 1]\); all other line segments are y-major (slope is determined by the segment’s endpoints). We shall specify rasterization only for x-major segments except in cases where the modifications for y-major segments are not self-evident.
Ideally, the GL uses a “diamond-exit” rule to determine those fragments that are produced by rasterizing a line segment. For each fragment $f$ with center at window coordinates $x_f$ and $y_f$, define a diamond-shaped region that is the intersection of four half planes:

$$R_f = \{ (x, y) \mid |x - x_f| + |y - y_f| < 1/2 \}$$

Essentially, a line segment starting at $p_a$ and ending at $p_b$ produces those fragments $f$ for which the segment intersects $R_f$, except if $p_b$ is contained in $R_f$. See figure 3.2.

To avoid difficulties when an endpoint lies on a boundary of $R_f$ we (in principle) perturb the supplied endpoints by a tiny amount. Let $p_a$ and $p_b$ have window coordinates $(x_a, y_a)$ and $(x_b, y_b)$, respectively. Obtain the perturbed endpoints $p'_a$ given by $(x_a, y_a) - (\epsilon, \epsilon^2)$ and $p'_b$ given by $(x_b, y_b) - (\epsilon, \epsilon^2)$. Rasterizing the line segment starting at $p_a$ and ending at $p_b$ produces those fragments $f$ for which the segment starting at $p'_a$ and ending on $p'_b$ intersects $R_f$, except if $p'_b$ is contained in $R_f$. $\epsilon$ is chosen to be so small that rasterizing the line segment produces the same fragments when $\delta$ is substituted for $\epsilon$ for any $0 < \delta \leq \epsilon$.

When $p_a$ and $p_b$ lie on fragment centers, this characterization of fragments reduces to Bresenham’s algorithm with one modification: lines produced in this description are “half-open,” meaning that the final fragment (corresponding to $p_b$) is not drawn. This means that when rasterizing a series of connected line segments, shared endpoints will be produced only once rather than twice (as would occur with Bresenham’s algorithm).

Because the initial and final conditions of the diamond-exit rule may be difficult to implement, other line segment rasterization algorithms are allowed, subject to the following rules:

1. The coordinates of a fragment produced by the algorithm may not deviate by more than one unit in either $x$ or $y$ window coordinates from a corresponding fragment produced by the diamond-exit rule.

2. The total number of fragments produced by the algorithm may differ from that produced by the diamond-exit rule by no more than one.

3. For an $x$-major line, no two fragments may be produced that lie in the same window-coordinate column (for a $y$-major line, no two fragments may appear in the same row).

4. If two line segments share a common endpoint, and both segments are either $x$-major (both left-to-right or both right-to-left) or $y$-major (both bottom-to-top or both top-to-bottom), then rasterizing both segments may not produce...
3.5. LINE SEGMENTS

duplicate fragments, nor may any fragments be omitted so as to interrupt continuity of the connected segments.

Next we must specify how the data associated with each rasterized fragment are obtained. Let the window coordinates of a produced fragment center be given by \( p_r = (x_d, y_d) \) and let \( p_a = (x_a, y_a) \) and \( p_b = (x_b, y_b) \). Set

\[
t = \frac{(p_r - p_a) \cdot (p_b - p_a)}{\|p_b - p_a\|^2}.
\]

(3.5)

(Note that \( t = 0 \) at \( p_a \) and \( t = 1 \) at \( p_b \).) The value of an associated datum \( f \) for the fragment, whether it be a shader output or the clip \( w \) coordinate, is found as

\[
f = \frac{(1-t)f_a/w_a + tf_b/w_b}{(1-t)/w_a + t/w_b}.
\]

(3.6)

where \( f_a \) and \( f_b \) are the data associated with the starting and ending endpoints of the segment, respectively; \( w_a \) and \( w_b \) are the clip \( w \) coordinates of the starting and ending endpoints of the segments, respectively. However, depth values for lines must be interpolated by

\[
z = (1-t)z_a + tz_b
\]

(3.7)
where \( z_a \) and \( z_b \) are the depth values of the starting and ending endpoints of the segment, respectively.

The \texttt{noperspective} and \texttt{flat} keywords used to declare shader outputs affect how they are interpolated. When neither keyword is specified, interpolation is performed as described in equation 3.6. When the \texttt{noperspective} keyword is specified, interpolation is performed in the same fashion as for depth values, as described in equation 3.7. When the \texttt{flat} keyword is specified, no interpolation is performed, and outputs are taken from the corresponding input value of the provoking vertex corresponding to that primitive (see section 2.19).

### 3.5.2 Other Line Segment Features

We have just described the rasterization of non-antialiased line segments of width one. We now describe the rasterization of line segments for general values of the line segment rasterization parameters.

#### Wide Lines

The actual width of non-antialiased lines is determined by rounding the supplied width to the nearest integer, then clamping it to the implementation-dependent maximum non-antialiased line width. This implementation-dependent value must be no less than the implementation-dependent maximum antialiased line width, rounded to the nearest integer value, and in any event no less than 1. If rounding the specified width results in the value 0, then it is as if the value were 1.

Non-antialiased line segments of width other than one are rasterized by offsetting them in the minor direction (for an \( x \)-major line, the minor direction is \( y \), and for a \( y \)-major line, the minor direction is \( x \)) and replicating fragments in the minor direction (see figure 3.3). Let \( w \) be the width rounded to the nearest integer (if \( w = 0 \), then it is as if \( w = 1 \)). If the line segment has endpoints given by \((x_0, y_0)\) and \((x_1, y_1)\) in window coordinates, the segment with endpoints \((x_0, y_0 - (w - 1)/2)\) and \((x_1, y_1 - (w - 1)/2)\) is rasterized, but instead of a single fragment, a column of fragments of height \( w \) (a row of fragments of length \( w \) for a \( y \)-major segment) is produced at each \( x \) (\( y \) for \( y \)-major) location. The lowest fragment of this column is the fragment that would be produced by rasterizing the segment of width 1 with the modified coordinates. The whole column is not produced if the stipple bit for the column’s \( x \) location is zero; otherwise, the whole column is produced.
3.5. LINE SEGMENTS

Antialiasing

Rasterized antialiased line segments produce fragments whose fragment squares intersect a rectangle centered on the line segment. Two of the edges are parallel to the specified line segment; each is at a distance of one-half the current width from that segment: one above the segment and one below it. The other two edges pass through the line endpoints and are perpendicular to the direction of the specified line segment. Coverage values are computed for each fragment by computing the area of the intersection of the rectangle with the fragment square (see figure 3.4; see also section 3.3). Equation 3.6 is used to compute associated data values just as with non-antialiased lines; equation 3.5 is used to find the value of \( t \) for each fragment whose square is intersected by the line segment’s rectangle. Not all widths need be supported for line segment antialiasing, but width 1.0 antialiased segments must be provided. As with the point width, a GL implementation may be queried for the range and number of gradations of available antialiased line widths.
3.5.3 Line Rasterization State

The state required for line rasterization consists of the floating-point line width and a bit indicating whether line antialiasing is on or off. The initial value of the line width is 1.0. The initial state of line segment antialiasing is disabled.

3.5.4 Line Multisample Rasterization

If MULTISAMPLE is enabled, and the value of SAMPLE_BUFFERS is one, then lines are rasterized using the following algorithm, regardless of whether line antialiasing (LINE_SMOOTH) is enabled or disabled. Line rasterization produces a fragment for each framebuffer pixel with one or more sample points that intersect the rectangular region that is described in the Antialiasing portion of section 3.5.2 (Other Line Segment Features).

Coverage bits that correspond to sample points that intersect a retained rectangle are 1, other coverage bits are 0. Each depth value and set of associated data is produced by substituting the corresponding sample location into equation 3.5, then using the result to evaluate equation 3.7. An implementation may choose to assign the associated data to more than one sample by evaluating equation 3.5 at any location within the pixel including the fragment center or any one of the sample locations, then substituting into equation 3.6. The different associated data values need not be evaluated at the same location.
3.6 Polygons

A polygon results from a triangle arising from a triangle strip, triangle fan, or series of separate triangles. Like points and line segments, polygon rasterization is controlled by several variables. Polygon antialiasing is controlled with Enable and Disable with the symbolic constant POLYGON_SMOOTH.

3.6.1 Basic Polygon Rasterization

The first step of polygon rasterization is to determine if the polygon is back-facing or front-facing. This determination is made based on the sign of the (clipped or unclipped) polygon’s area computed in window coordinates. One way to compute this area is

\[ a = \frac{1}{2} \sum_{i=0}^{n-1} x_i^w y_{i+1}^w - x_{i+1}^w y_i^w \]  

(3.8)

where \( x_i^w \) and \( y_i^w \) are the \( x \) and \( y \) window coordinates of the \( i \)th vertex of the \( n \)-vertex polygon (vertices are numbered starting at zero for purposes of this computation) and \( i \oplus 1 \) is \( (i + 1) \mod n \). The interpretation of the sign of this value is controlled with

```c
void FrontFace( enum dir );
```

Setting \( dir \) to CCW (corresponding to counter-clockwise orientation of the projected polygon in window coordinates) uses \( a \) as computed above. Setting \( dir \) to CW (corresponding to clockwise orientation) indicates that the sign of \( a \) should be reversed prior to use. Front face determination requires one bit of state, and is initially set to CCW.

If the sign of \( a \) (including the possible reversal of this sign as determined by FrontFace) is positive, the polygon is front-facing; otherwise, it is back-facing. This determination is used in conjunction with the CullFace enable bit and mode value to decide whether or not a particular polygon is rasterized. The CullFace mode is set by calling

```c
void CullFace( enum mode );
```
3.6. **POLYGONS**

*mode* is a symbolic constant: one of FRONT, BACK or FRONT_AND_BACK. Culling is enabled or disabled with **Enable** or **Disable** using the symbolic constant CULL_FACE. Front-facing polygons are rasterized if either culling is disabled or the CullFace mode is BACK while back-facing polygons are rasterized only if either culling is disabled or the CullFace mode is FRONT. The initial setting of the CullFace mode is BACK. Initially, culling is disabled.

The rule for determining which fragments are produced by polygon rasterization is called point sampling. The two-dimensional projection obtained by taking the $x$ and $y$ window coordinates of the polygon’s vertices is formed. Fragment centers that lie inside of this polygon are produced by rasterization. Special treatment is given to a fragment whose center lies on a polygon edge. In such a case we require that if two polygons lie on either side of a common edge (with identical endpoints) on which a fragment center lies, then exactly one of the polygons results in the production of the fragment during rasterization.

As for the data associated with each fragment produced by rasterizing a polygon, we begin by specifying how these values are produced for fragments in a triangle. Define *barycentric coordinates* for a triangle. Barycentric coordinates are a set of three numbers, $a$, $b$, and $c$, each in the range $[0, 1]$, with $a + b + c = 1$. These coordinates uniquely specify any point $p$ within the triangle or on the triangle’s boundary as

$$p = ap_a + bp_b + cp_c,$$

where $p_a$, $p_b$, and $p_c$ are the vertices of the triangle. $a$, $b$, and $c$ can be found as

$$a = \frac{A(pp_bp_c)}{A(p_ap_bp_c)}, \quad b = \frac{A(pp_ap_c)}{A(p_ap_bp_c)}, \quad c = \frac{A(pp_ap_b)}{A(p_ap_bp_c)},$$

where $A(lmn)$ denotes the area in window coordinates of the triangle with vertices $l$, $m$, and $n$.

Denote an associated datum at $p_a$, $p_b$, or $p_c$ as $f_a$, $f_b$, or $f_c$, respectively. Then the value $f$ of a datum at a fragment produced by rasterizing a triangle is given by

$$f = \frac{af_a/w_a + bf_b/w_b + cf_c/w_c}{a/w_a + b/w_b + c/w_c} \quad (3.9)$$

where $w_a$, $w_b$, and $w_c$ are the clip $w$ coordinates of $p_a$, $p_b$, and $p_c$, respectively. $a$, $b$, and $c$ are the barycentric coordinates of the fragment for which the data are produced. $a$, $b$, and $c$ must correspond precisely to the exact coordinates of the center of the fragment. Another way of saying this is that the data associated with a fragment must be sampled at the fragment’s center. However, depth values for polygons must be interpolated by

$$z = az_a + bz_b + cz_c \quad (3.10)$$
3.6. POLYGONS

where \( z_a, z_b, \) and \( z_c \) are the depth values of \( p_a, p_b, \) and \( p_c, \) respectively.

The \texttt{noperspective} and \texttt{flat} keywords used to declare shader outputs affect how they are interpolated. When neither keyword is specified, interpolation is performed as described in equation 3.9. When the \texttt{noperspective} keyword is specified, interpolation is performed in the same fashion as for depth values, as described in equation 3.10. When the \texttt{flat} keyword is specified, no interpolation is performed, and outputs are taken from the corresponding input value of the provoking vertex corresponding to that primitive (see section 2.19).

For a polygon with more than three edges, such as may be produced by clipping a triangle, we require only that a convex combination of the values of the datum at the polygon’s vertices can be used to obtain the value assigned to each fragment produced by the rasterization algorithm. That is, it must be the case that at every fragment

\[
f = \sum_{i=1}^{n} a_i f_i
\]

where \( n \) is the number of vertices in the polygon, \( f_i \) is the value of the \( f \) at vertex \( i \); for each \( i \) \( 0 \leq a_i \leq 1 \) and \( \sum_{i=1}^{n} a_i = 1 \). The values of the \( a_i \) may differ from fragment to fragment, but at vertex \( i, a_j = 0, j \neq i \) and \( a_i = 1 \).

One algorithm that achieves the required behavior is to triangulate a polygon (without adding any vertices) and then treat each triangle individually as already discussed. A scan-line rasterizer that linearly interpolates data along each edge and then linearly interpolates data across each horizontal span from edge to edge also satisfies the restrictions (in this case, the numerator and denominator of equation 3.9 should be iterated independently and a division performed for each fragment).

3.6.2 Antialiasing

Polygon antialiasing rasterizes a polygon by producing a fragment wherever the interior of the polygon intersects that fragment’s square. A coverage value is computed at each such fragment, and this value is saved to be applied as described in section 3.11. An associated datum is assigned to a fragment by integrating the datum’s value over the region of the intersection of the fragment square with the polygon’s interior and dividing this integrated value by the area of the intersection. For a fragment square lying entirely within the polygon, the value of a datum at the fragment’s center may be used instead of integrating the value across the fragment.

3.6.3 Options Controlling Polygon Rasterization

The interpretation of polygons for rasterization is controlled using
3.6. POLYGONS

```c
void PolygonMode( enum face, enum mode );
```

`face` must be `FRONT_AND_BACK`, indicating that the rasterizing method described by `mode` replaces the rasterizing method for both front- and back-facing polygons. `mode` is one of the symbolic constants `POINT`, `LINE`, or `FILL`. Calling `PolygonMode` with `POINT` causes the vertices of a polygon to be treated, for rasterization purposes, as if they had been drawn with `mode` `POINTS`. `LINE` causes edges to be rasterized as line segments. `FILL` is the default mode of polygon rasterization, corresponding to the description in sections 3.6.1 and 3.6.2. Note that these modes affect only the final rasterization of polygons: in particular, a polygon’s vertices are lit, and the polygon is clipped and possibly culled before these modes are applied.

Polygon antialiasing applies only to the `FILL` state of `PolygonMode`. For `POINT` or `LINE`, point antialiasing or line segment antialiasing, respectively, apply.

### 3.6.4 Depth Offset

The depth values of all fragments generated by the rasterization of a polygon may be offset by a single value that is computed for that polygon. The function that determines this value is specified by calling

```c
void PolygonOffset( float factor, float units );
```

`factor` scales the maximum depth slope of the polygon, and `units` scales an implementation-dependent constant that relates to the usable resolution of the depth buffer. The resulting values are summed to produce the polygon offset value. Both `factor` and `units` may be either positive or negative.

The maximum depth slope $m$ of a triangle is

$$m = \sqrt{\left(\frac{\partial z_w}{\partial x_w}\right)^2 + \left(\frac{\partial z_w}{\partial y_w}\right)^2}$$

(3.11)

where $(x_w, y_w, z_w)$ is a point on the triangle. $m$ may be approximated as

$$m = \max \left\{ \left| \frac{\partial z_w}{\partial x_w} \right|, \left| \frac{\partial z_w}{\partial y_w} \right| \right\}.$$  

(3.12)

The minimum resolvable difference $r$ is an implementation-dependent parameter that depends on the depth buffer representation. It is the smallest difference in window coordinate $z$ values that is guaranteed to remain distinct throughout polygon rasterization and in the depth buffer. All pairs of fragments generated by the

OpenGL 4.2 (Core Profile) - April 27, 2012
rasterization of two polygons with otherwise identical vertices, but \( z_w \) values that differ by \( r \), will have distinct depth values.

For fixed-point depth buffer representations, \( r \) is constant throughout the range of the entire depth buffer. For floating-point depth buffers, there is no single minimum resolvable difference. In this case, the minimum resolvable difference for a given polygon is dependent on the maximum exponent, \( e \), in the range of \( z \) values spanned by the primitive. If \( n \) is the number of bits in the floating-point mantissa, the minimum resolvable difference, \( r \), for the given primitive is defined as

\[
r = 2^{e-n}.
\]

If no depth buffer is present, \( r \) is undefined.

The offset value \( o \) for a polygon is

\[
o = m \times \text{factor} + r \times \text{units}.
\]

\( m \) is computed as described above. If the depth buffer uses a fixed-point representation, \( m \) is a function of depth values in the range \([0, 1]\), and \( o \) is applied to depth values in the same range.

Boolean state values \texttt{POLYGON_OFFSET_POINT}, \texttt{POLYGON_OFFSET_LINE}, and \texttt{POLYGON_OFFSET_FILL} determine whether \( o \) is applied during the rasterization of polygons in \texttt{POINT}, \texttt{LINE}, and \texttt{FILL} modes. These boolean state values are enabled and disabled as argument values to the commands \texttt{Enable} and \texttt{Disable}. If \texttt{POLYGON_OFFSET_POINT} is enabled, \( o \) is added to the depth value of each fragment produced by the rasterization of a polygon in \texttt{POINT} mode. Likewise, if \texttt{POLYGON_OFFSET_LINE} or \texttt{POLYGON_OFFSET_FILL} is enabled, \( o \) is added to the depth value of each fragment produced by the rasterization of a polygon in \texttt{LINE} or \texttt{FILL} modes, respectively.

For fixed-point depth buffers, fragment depth values are always limited to the range \([0, 1]\) by clamping after offset addition is performed. Fragment depth values are clamped even when the depth buffer uses a floating-point representation.

### 3.6.5 Polygon Multisample Rasterization

If \texttt{MULTISAMPLE} is enabled and the value of \texttt{SAMPLE_BUFFER} is one, then polygons are rasterized using the following algorithm, regardless of whether polygon antialiasing (\texttt{POLYGONMOOTH}) is enabled or disabled. Polygon rasterization produces a fragment for each framebuffer pixel with one or more sample points that satisfy the point sampling criteria described in section 3.6.1. If a polygon is culled, based on its orientation and the \texttt{CullFace} mode, then no fragments are produced during rasterization.
3.7. PIXEL RECTANGLES

Coverage bits that correspond to sample points that satisfy the point sampling criteria are 1, other coverage bits are 0. Each associated datum is produced as described in section 3.6.1, but using the corresponding sample location instead of the fragment center. An implementation may choose to assign the same associated data values to more than one sample by barycentric evaluation using any location within the pixel including the fragment center or one of the sample locations.

When using a vertex shader, the noperspective and flat qualifiers affect how shader outputs are interpolated in the same fashion as described for for basic polygon rasterization in section 3.6.1.

The rasterization described above applies only to the FILL state of Polygon-Mode. For POINT and LINE, the rasterizations described in sections 3.4.3 (Point Multisample Rasterization) and 3.5.4 (Line Multisample Rasterization) apply.

3.6.6 Polygon Rasterization State

The state required for polygon rasterization consists of the current state of polygon antialiasing (enabled or disabled), the current values of the PolygonMode setting, whether point, line, and fill mode polygon offsets are enabled or disabled, and the factor and bias values of the polygon offset equation. The initial setting of polygon antialiasing is disabled. The initial state for PolygonMode is FILL. The initial polygon offset factor and bias values are both 0; initially polygon offset is disabled for all modes.

3.7 Pixel Rectangles

Rectangles of color, depth, and certain other values may be specified to the GL using TexImage*D (see section 3.9.3). Some of the parameters and operations governing the operation of these commands are shared by ReadPixels (used to obtain pixel values from the framebuffer); the discussion of ReadPixels, however, is deferred until chapter 4 after the framebuffer has been discussed in detail. Nevertheless, we note in this section when parameters and state pertaining to these commands also pertain to ReadPixels.

A number of parameters control the encoding of pixels in buffer object or client memory (for reading and writing) and how pixels are processed before being placed in or after being read from the framebuffer (for reading, writing, and copying). These parameters are set with PixelStore.
Table 3.1: PixelStore parameters pertaining to one or more of TexImage*D, TexSubImage*D, CompressedTexImage*D and CompressedTexSubImage*D.

### 3.7.1 Pixel Storage Modes and Pixel Buffer Objects

Pixel storage modes affect the operation of TexImage*D, TexSubImage*D, CompressedTexImage*D, CompressedTexSubImage*D, and ReadPixels when one of these commands is issued. Pixel storage modes are set with

```c
void PixelStore{if}((enum pname, T param);
```

`pname` is a symbolic constant indicating a parameter to be set, and `param` is the value to set it to. Table 3.1 summarizes the pixel storage parameters, their types, their initial values, and their allowable ranges. Setting a parameter to a value outside the given range results in the error `INVALID_VALUE`.

Data conversions are performed as specified in section 2.3.1.

In addition to storing pixel data in client memory, pixel data may also be stored in buffer objects (described in section 2.9). The current pixel unpack and pack buffer objects are designated by the `PIXEL_UNPACK_BUFFER` and `PIXEL_PACK_BUFFER` targets respectively.

Initially, zero is bound for the `PIXEL_UNPACK_BUFFER`, indicating that image specification commands such as TexImage*D source their pixels from client memory pointer parameters. However, if a non-zero buffer object is bound as the current pixel unpack buffer, then the pointer parameter is treated as an offset into the designated buffer object.
3.7. PIXEL RECTANGLES

3.7.2 Transfer of Pixel Rectangles

The process of transferring pixels encoded in buffer object or client memory is diagrammed in figure 3.5. We describe the stages of this process in the order in which they occur.

Commands accepting or returning pixel rectangles take the following arguments (as well as additional arguments specific to their function):

- **format** is a symbolic constant indicating what the values in memory represent.
- **width** and **height** are the width and height, respectively, of the pixel rectangle to be transferred.
- **data** refers to the data to be drawn. These data are represented with one of several GL data types, specified by **type**. The correspondence between the **type** token values and the GL data types they indicate is given in table 3.2.

Not all combinations of **format** and **type** are valid. If **format** is **DEPTH_STENCIL** and **type** is not **UNSIGNED_INT_24_8** or **FLOAT_32_UNSIGNED_INT_24_8_REV**, then the error **INVALID_ENUM** occurs. If **format** is one of the **INTEGER** component formats as defined in table 3.3 and **type** is one of the floating-point types as defined in table 3.2, an **INVALID_ENUM** error is generated. Some

OpenGL 4.2 (Core Profile) - April 27, 2012
additional constraints on the combinations of format and type values that are accepted are discussed below. Additional restrictions may be imposed by specific commands.

**Unpacking**

Data are taken from the currently bound pixel unpack buffer or client memory as a sequence of signed or unsigned bytes (GL data types byte and ubyte), signed or unsigned short integers (GL data types short and ushort), signed or unsigned integers (GL data types int and uint), or floating-point values (GL data types half and float). These elements are grouped into sets of one, two, three, or four values, depending on the format, to form a group. Table 3.3 summarizes the format of groups obtained from memory; it also indicates those formats that yield indices and those that yield floating-point or integer components.

If a pixel unpack buffer is bound (as indicated by a non-zero value of PIXEL_UNPACK_BUFFER_BINDING), data is an offset into the pixel unpack buffer and the pixels are unpacked from the buffer relative to this offset; otherwise, data is a pointer to client memory and the pixels are unpacked from client memory relative to the pointer. If a pixel unpack buffer object is bound and unpacking the pixel data according to the process described below would access memory beyond the size of the pixel unpack buffer’s memory size, an INVALID_OPERATION error results. If a pixel unpack buffer object is bound and data is not evenly divisible by the number of basic machine units needed to store in memory the corresponding GL data type from table 3.2 for the type parameter (or not evenly divisible by 4 for type FLOAT_32_UNSIGNED_INT_24_8_REV, which does not have a corresponding GL data type), an INVALID_OPERATION error results.

By default the values of each GL data type are interpreted as they would be specified in the language of the client’s GL binding. If UNPACK_SWAP_BYTES is enabled, however, then the values are interpreted with the bit orderings modified as per table 3.4. The modified bit orderings are defined only if the GL data type ubyte has eight bits, and then for each specific GL data type only if that type is represented with 8, 16, or 32 bits.

The groups in memory are treated as being arranged in a rectangle. This rectangle consists of a series of rows, with the first element of the first group of the first row pointed to by data. If the value of UNPACK_ROW_LENGTH is not positive, then the number of groups in a row is width; otherwise the number of groups is UNPACK_ROW_LENGTH. If p indicates the location in memory of the first element of the first row, then the first element of the Nth row is indicated by

\[ p + Nk \]  

(3.14)
### Table 3.2: Pixel data type parameter values and the corresponding GL data types.

Refer to table 2.2 for definitions of GL data types. Special interpretations are described near the end of section 3.6. Floating-point types are incompatible with INTEGER formats as described above.

<table>
<thead>
<tr>
<th>type Parameter Token Name</th>
<th>Corresponding GL Data Type</th>
<th>Special Interpretation</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNSIGNED_BYTE</td>
<td>ubyte</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>BYTE</td>
<td>byte</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_SHORT</td>
<td>ushort</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SHORT</td>
<td>short</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_INT</td>
<td>uint</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>INT</td>
<td>int</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>HALF_FLOAT</td>
<td>half</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>FLOAT</td>
<td>float</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_BYTE_3_3_2</td>
<td>ubyte</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_BYTE_2_3_3_REV</td>
<td>ubyte</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_6_5</td>
<td>ushort</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_6_5_REV</td>
<td>ushort</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_4_4_4_4</td>
<td>ushort</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_4_4_4_4_REV</td>
<td>ushort</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_5_5_1</td>
<td>ushort</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_1_5_5_5_REV</td>
<td>ushort</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_INT_8_8_8_8</td>
<td>uint</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_INT_8_8_8_8_REV</td>
<td>uint</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_INT_10_10_10_2</td>
<td>uint</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_INT_2_10_10_10_REV</td>
<td>uint</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_INT_24_8</td>
<td>uint</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_INT_10F_11F_11F_REV</td>
<td>uint</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_INT_5_9_9_9_REV</td>
<td>uint</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>FLOAT_32_UNSIGNED_INT_24_8_REV</td>
<td>n/a</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_INT_5_9_9_9_REV</td>
<td>uint</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**OpenGL 4.2 (Core Profile) - April 27, 2012**
### 3.7. PIXEL RECTANGLES

<table>
<thead>
<tr>
<th>Format Name</th>
<th>Element Meaning and Order</th>
<th>Target Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>STENCIL_INDEX</td>
<td>Stencil Index</td>
<td>Stencil</td>
</tr>
<tr>
<td>DEPTH_COMPONENT</td>
<td>Depth</td>
<td>Depth</td>
</tr>
<tr>
<td>DEPTH_STENCIL</td>
<td>Depth and Stencil Index</td>
<td>Depth and Stencil</td>
</tr>
<tr>
<td>RED</td>
<td>R</td>
<td>Color</td>
</tr>
<tr>
<td>GREEN</td>
<td>G</td>
<td>Color</td>
</tr>
<tr>
<td>BLUE</td>
<td>B</td>
<td>Color</td>
</tr>
<tr>
<td>RG</td>
<td>R, G</td>
<td>Color</td>
</tr>
<tr>
<td>RGB</td>
<td>R, G, B</td>
<td>Color</td>
</tr>
<tr>
<td>RGBA</td>
<td>R, G, B, A</td>
<td>Color</td>
</tr>
<tr>
<td>BGR</td>
<td>B, G, R</td>
<td>Color</td>
</tr>
<tr>
<td>BGRA</td>
<td>B, G, R, A</td>
<td>Color</td>
</tr>
<tr>
<td>RED_INTEGER</td>
<td>iR</td>
<td>Color</td>
</tr>
<tr>
<td>GREEN_INTEGER</td>
<td>iG</td>
<td>Color</td>
</tr>
<tr>
<td>BLUE_INTEGER</td>
<td>iB</td>
<td>Color</td>
</tr>
<tr>
<td>RG_INTEGER</td>
<td>iR, iG</td>
<td>Color</td>
</tr>
<tr>
<td>RGB_INTEGER</td>
<td>iR, iG, iB</td>
<td>Color</td>
</tr>
<tr>
<td>RGBA_INTEGER</td>
<td>iR, iG, iB, iA</td>
<td>Color</td>
</tr>
<tr>
<td>BGR_INTEGER</td>
<td>iB, iG, iR</td>
<td>Color</td>
</tr>
<tr>
<td>BGRA_INTEGER</td>
<td>iB, iG, iR, iA</td>
<td>Color</td>
</tr>
</tbody>
</table>

Table 3.3: Pixel data formats. The second column gives a description of and the number and order of elements in a group. Unless specified as an index, formats yield components. Components are floating-point unless prefixed with the letter ‘i’, which indicates they are integer.

<table>
<thead>
<tr>
<th>Element Size</th>
<th>Default Bit Ordering</th>
<th>Modified Bit Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 bit</td>
<td>[7..0]</td>
<td>[7..0]</td>
</tr>
<tr>
<td>16 bit</td>
<td>[15..0]</td>
<td>[7..0][15..8]</td>
</tr>
<tr>
<td>32 bit</td>
<td>[31..0]</td>
<td>[7..0][15..8][23..16][31..24]</td>
</tr>
</tbody>
</table>

Table 3.4: Bit ordering modification of elements when UNPACK_SWAP_BYTES is enabled. These reorderings are defined only when GL data type ubyte has 8 bits, and then only for GL data types with 8, 16, or 32 bits. Bit 0 is the least significant.
where $N$ is the row number (counting from zero) and $k$ is defined as

$$
k = \begin{cases} 
nl & s \geq a, \\
a \left\lfloor \frac{snl}{a} \right\rfloor & s < a \end{cases} \quad (3.15)
$$

where $n$ is the number of elements in a group, $l$ is the number of groups in the row, $a$ is the value of UNPACK_ALIGNMENT, and $s$ is the size, in units of GL ubyte, of an element. If the number of bits per element is not 1, 2, 4, or 8 times the number of bits in a GL ubyte, then $k = nl$ for all values of $a$.

There is a mechanism for selecting a sub-rectangle of groups from a larger containing rectangle. This mechanism relies on three integer parameters: UNPACK_ROW_LENGTH, UNPACK_SKIP_ROWS, and UNPACK_SKIP_PIXELS. Before obtaining the first group from memory, the data pointer is advanced by $(\text{UNPACK_SKIP_PIXELS})n + (\text{UNPACK_SKIP_ROWS})k$ elements. Then width groups are obtained from contiguous elements in memory (without advancing the pointer), after which the pointer is advanced by $k$ elements. height sets of width groups of values are obtained this way. See figure 3.6.

**Special Interpretations**

A type matching one of the types in table 3.5 is a special case in which all the components of each group are packed into a single unsigned byte, unsigned short, or unsigned int, depending on the type. If type is FLOAT_32 UNSIGNED_-INT_24_6_REV, the components of each group are contained within two 32-bit...
words; the first word contains the float component, and the second word contains a packed 24-bit unused field, followed by an 8-bit component. The number of components per packed pixel is fixed by the type, and must match the number of components per group indicated by the format parameter, as listed in table 3.5. The error INVALID_OPERATION is generated by any command processing pixel rectangles if a mismatch occurs.

Bitfield locations of the first, second, third, and fourth components of each packed pixel type are illustrated in tables 3.6-3.9. Each bitfield is interpreted as an unsigned integer value.

Components are normally packed with the first component in the most significant bits of the bitfield, and successive component occupying progressively less significant locations. Types whose token names end with _REV reverse the component packing order from least to most significant locations. In all cases, the most significant bit of each component is packed in the most significant bit location of its location in the bitfield.
### Table 3.5: Packed pixel formats.

<table>
<thead>
<tr>
<th>Type Parameter</th>
<th>Token Name</th>
<th>GL Data Type</th>
<th>Number of Components</th>
<th>Matching Pixel Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNSIGNED_BYTE_3_3_2</td>
<td>unsigned byte</td>
<td>3</td>
<td></td>
<td>RGB, RGBA, RGBA_, INTEGER, BGRA_, INTEGER</td>
</tr>
<tr>
<td>UNSIGNED_BYTE_2_3_3_REV</td>
<td>signed byte</td>
<td>3</td>
<td></td>
<td>RGB, RGBA, RGBA_, INTEGER, BGRA_, INTEGER</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_6_5</td>
<td>short byte</td>
<td>3</td>
<td></td>
<td>RGB, RGBA, RGBA_, INTEGER, BGRA_, INTEGER</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_6_5_REV</td>
<td>short byte</td>
<td>3</td>
<td></td>
<td>RGB, RGBA, RGBA_, INTEGER, BGRA_, INTEGER</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_4_4_4</td>
<td>short word</td>
<td>4</td>
<td></td>
<td>RGBA, RGBA, RGBA_, INTEGER, BGRA_, INTEGER</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_4_4_4_REV</td>
<td>short word</td>
<td>4</td>
<td></td>
<td>RGBA, RGBA, RGBA_, INTEGER, BGRA_, INTEGER</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_5_5_1</td>
<td>short word</td>
<td>4</td>
<td></td>
<td>RGBA, RGBA, RGBA_, INTEGER, BGRA_, INTEGER</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_1_5_5_5_REV</td>
<td>short word</td>
<td>4</td>
<td></td>
<td>RGBA, RGBA, RGBA_, INTEGER, BGRA_, INTEGER</td>
</tr>
<tr>
<td>UNSIGNED_INT_8_8_8_8</td>
<td>unsigned int</td>
<td>4</td>
<td></td>
<td>RGBA, RGBA, RGBA_, INTEGER, BGRA_, INTEGER</td>
</tr>
<tr>
<td>UNSIGNED_INT_8_8_8_8_REV</td>
<td>unsigned int</td>
<td>4</td>
<td></td>
<td>RGBA, RGBA, RGBA_, INTEGER, BGRA_, INTEGER</td>
</tr>
<tr>
<td>UNSIGNED_INT_10_10_10_2</td>
<td>unsigned int</td>
<td>4</td>
<td></td>
<td>RGBA, RGBA, RGBA_, INTEGER, BGRA_, INTEGER</td>
</tr>
<tr>
<td>UNSIGNED_INT_2_10_10_10_REV</td>
<td>unsigned int</td>
<td>4</td>
<td></td>
<td>RGBA, RGBA, RGBA_, INTEGER, BGRA_, INTEGER</td>
</tr>
<tr>
<td>UNSIGNED_INT_24_8</td>
<td>unsigned int</td>
<td>2</td>
<td></td>
<td>DEPTH_STENCIL</td>
</tr>
<tr>
<td>UNSIGNED_INT_10F_11F_11F_REV</td>
<td>float</td>
<td>3</td>
<td></td>
<td>RGB</td>
</tr>
<tr>
<td>UNSIGNED_INT_5_9_9_9_REV</td>
<td>unsigned int</td>
<td>4</td>
<td></td>
<td>RGB</td>
</tr>
<tr>
<td>FLOAT_32_UNSIGNED_INT_24_8_REV</td>
<td>n/a</td>
<td>2</td>
<td></td>
<td>DEPTH_STENCIL</td>
</tr>
</tbody>
</table>

OpenGL 4.2 (Core Profile) - April 27, 2012
3.7. PIXEL RECTANGLES

UNSIGNED_BYTE_3_3_2:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Component</td>
<td>2nd</td>
<td>3rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UNSIGNED_BYTE_2_3_3_REV:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3rd</td>
<td>2nd</td>
<td>1st Component</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.6: UNSIGNED_BYTE formats. Bit numbers are indicated for each component.
### 3.7. PIXEL RECTANGLES

<table>
<thead>
<tr>
<th>Format</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UNSIGNED_SHORT_5_6_5</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>1st</td>
<td>2nd</td>
<td>3rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>UNSIGNED_SHORT_5_6_5_REV</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>3rd</td>
<td>2nd</td>
<td>1st</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>UNSIGNED_SHORT_4_4_4_4</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>1st</td>
<td>2nd</td>
<td>3rd</td>
<td>4th</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>UNSIGNED_SHORT_4_4_4_4_REV</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>4th</td>
<td>3rd</td>
<td>2nd</td>
<td>1st</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>UNSIGNED_SHORT_5_5_5_1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>1st</td>
<td>2nd</td>
<td>3rd</td>
<td>4th</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>UNSIGNED_SHORT_1_5_5_5_REV</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>4th</td>
<td>3rd</td>
<td>2nd</td>
<td>1st</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.7: **UNSIGNED_SHORT formats**

OpenGL 4.2 (Core Profile) - April 27, 2012
### UNSIGNED_INT_8_8_8_8:

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1st Component | 2nd | 3rd | 4th |

### UNSIGNED_INT_8_8_8_8_REV:

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 4th | 3rd | 2nd | 1st Component |

### UNSIGNED_INT_10_10_10_2:

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1st Component | 2nd | 3rd | 4th |

### UNSIGNED_INT_2_10_10_10_REV:

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 4th | 3rd | 2nd | 1st Component |

### UNSIGNED_INT_24_8:

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1st Component | 2nd |

### UNSIGNED_INT_10F_11F_11F_REV:

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 3rd | 2nd | 1st Component |

### UNSIGNED_INT_5_9_9_9_REV:

|   | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 4th | 3rd | 2nd | 1st Component |

Table 3.8: UNSIGNED_INT formats
3.7. PIXEL RECTANGLES

FLOAT_32_UNSIGNED_INT_24_8_REV:

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Component</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unused 2nd</td>
</tr>
</tbody>
</table>

Table 3.9: FLOAT UNSIGNED INT formats
### 3.7. PIXEL RECTANGLES

<table>
<thead>
<tr>
<th>Format</th>
<th>First Component</th>
<th>Second Component</th>
<th>Third Component</th>
<th>Fourth Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB</td>
<td>red</td>
<td>green</td>
<td>blue</td>
<td></td>
</tr>
<tr>
<td>RGBA</td>
<td>red</td>
<td>green</td>
<td>blue</td>
<td>alpha</td>
</tr>
<tr>
<td>BGRA</td>
<td>blue</td>
<td>green</td>
<td>red</td>
<td>alpha</td>
</tr>
<tr>
<td>DEPTH_STENCIL</td>
<td>depth</td>
<td>stencil</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.10: Packed pixel field assignments.

The assignment of component to fields in the packed pixel is as described in table 3.10.

Byte swapping, if enabled, is performed before the components are extracted from each pixel. The above discussions of row length and image extraction are valid for packed pixels, if “group” is substituted for “component” and the number of components per group is understood to be one.

A type of `UNSIGNED_INT_10F_11F_11F_REV` and format of `RGB` is a special case in which the data are a series of GL `uint` values. Each `uint` value specifies 3 packed components as shown in table 3.8. The 1st, 2nd, and 3rd components are called $f_{\text{red}}$ (11 bits), $f_{\text{green}}$ (11 bits), and $f_{\text{blue}}$ (10 bits) respectively.

$f_{\text{red}}$ and $f_{\text{green}}$ are treated as unsigned 11-bit floating-point values and converted to floating-point red and green components respectively as described in section 2.1.1. $f_{\text{blue}}$ is treated as an unsigned 10-bit floating-point value and converted to a floating-point blue component as described in section 2.1.1.

A type of `UNSIGNED_INT_5_9_9_9_REV` and format of `RGB` is a special case in which the data are a series of GL `uint` values. Each `uint` value specifies 4 packed components as shown in table 3.8. The 1st, 2nd, 3rd, and 4th components are called $p_{\text{red}}$, $p_{\text{green}}$, $p_{\text{blue}}$, and $p_{\text{exp}}$ respectively and are treated as unsigned integers. These are then used to compute floating-point RGB components (ignoring the “Conversion to floating-point” section below in this case) as follows:

\[
\begin{align*}
red &= p_{\text{red}}2^{p_{\text{exp}}-B-N} \\
\text{green} &= p_{\text{green}}2^{p_{\text{exp}}-B-N} \\
\text{blue} &= p_{\text{blue}}2^{p_{\text{exp}}-B-N}
\end{align*}
\]

where $B = 15$ (the exponent bias) and $N = 9$ (the number of mantissa bits).
3.8 EARLY PER-FRAGMENT TESTS

Conversion to floating-point

This step applies only to groups of floating-point components. It is not performed on indices or integer components. For groups containing both components and indices, such as DEPTH_STENCIL, the indices are not converted.

Each element in a group is converted to a floating-point value. For unsigned integer elements, equation 2.1 is used. For signed integer elements, equation 2.2 is used.

Final Expansion to RGBA

This step is performed only for non-depth component groups. Each group is converted to a group of 4 elements as follows: if a group does not contain an A element, then A is added and set to 1 for integer components or 1.0 for floating-point components. If any of R, G, or B is missing from the group, each missing element is added and assigned a value of 0 for integer components or 0.0 for floating-point components.

3.8 Early Per-Fragment Tests

Once fragments are produced by rasterization, a number of per-fragment operations may be performed prior to fragment shader execution. If a fragment is discarded during any of these operations, it will not be processed by any subsequent stage, including fragment shader execution.

Up to five operations are performed on each fragment, in the following order:

- the pixel ownership test (see section 4.1.1);
- the scissor test (see section 4.1.2);
- the stencil test (see section 4.1.4);
- the depth buffer test (see section 4.1.5); and
- occlusion query sample counting (see section 4.1.6).

The pixel ownership and scissor tests are always performed.
The other operations are performed if and only if early fragment tests are enabled in the active fragment shader (see section 3.10.2). When early per-fragment operations are enabled, the stencil test, depth buffer test, and occlusion query sample counting operations are performed prior to fragment shader execution, and the
stencil buffer, depth buffer, and occlusion query sample counts will be updated accordingly. When early per-fragment operations are enabled, these operations will not be performed again after fragment shader execution. When the active program has no fragment shader, or the active program was linked with early fragment tests disabled, these operations are performed only after fragment program execution, in the order described in chapter 4.

If early fragment tests are enabled, any depth value computed by the fragment shader has no effect. Additionally, the depth buffer, stencil buffer, and occlusion query sample counts may be updated even for fragments or samples that would be discarded after fragment shader execution due to per-fragment operations such as alpha-to-coverage tests.

3.9 Texturing

Texturing maps a portion of one or more specified images onto a fragment or vertex. This mapping is accomplished in shaders by sampling the color of an image at the location indicated by specified $(s, t, r)$ texture coordinates. Texture lookups are typically used to modify a fragment’s RGBA color but may be used for any purpose in a shader.

The internal data type of a texture may be signed or unsigned normalized fixed-point, signed or unsigned integer, or floating-point, depending on the internal format of the texture. The correspondence between the internal format and the internal data type is given in tables 3.12-3.13. Fixed-point and floating-point textures return a floating-point value and integer textures return signed or unsigned integer values. The fragment shader is responsible for interpreting the result of a texture lookup as the correct data type, otherwise the result is undefined.

Each of the supported types of texture is a collection of images built from one-, two-, or three-dimensional arrays of image elements referred to as texels. One-, two-, and three-dimensional textures consist respectively of one-, two-, or three-dimensional texel arrays. One- and two-dimensional array textures are arrays of one- or two-dimensional images, consisting of one or more layers. Two-dimensional multisample and two-dimensional multisample array textures are special two-dimensional and two-dimensional array textures, respectively, containing multiple samples in each texel. Cube maps are special two-dimensional array textures with six layers that represent the faces of a cube. When accessing a cube map, the texture coordinates are projected onto one of the six faces of the cube. A cube map array is a collection of cube map layers stored as a two-dimensional array texture. When accessing a cube map array, the texture coordinate $s$, $t$, and $r$ are applied similarly as cube maps while the last texture coordinate $q$ is used as the index.
of one of the cube map slices. Rectangular textures are special two-dimensional textures consisting of only a single image and accessed using unnormalized coordinates. Buffer textures are special one-dimensional textures whose texel arrays are stored in separate buffer objects.

Implementations must support texturing using multiple images. The following subsections (up to and including section 3.9.11) specify the GL operation with a single texture. The process by which multiple texture images may be sampled and combined by the application-supplied vertex and fragment shaders is described in sections 2.11 and 3.10.

The coordinates used for texturing in a fragment shader are defined by the OpenGL Shading Language Specification.

The command

```c
void ActiveTexture( enum texture );
```

specifies the active texture unit selector, ACTIVE_TEXTURE. Each texture image unit consists of all the texture state defined in section 3.9.

The active texture unit selector selects the texture image unit accessed by commands involving texture image processing. Such commands include TexParameter, TexImage, BindTexture, and queries of all such state.

ActiveTexture generates the error INVALID_ENUM if an invalid texture is specified. texture is a symbolic constant of the form TEXTUREi, indicating that texture unit i is to be modified. The constants obey TEXTUREi = TEXTURE0 + i (i is in the range 0 to k − 1, where k is the value of MAX_COMBINED_TEXTURE_IMAGE_UNITS).

The state required for the active texture image unit selector is a single integer. The initial value is TEXTURE0.

### 3.9.1 Texture Objects

Textures in GL are represented by named objects. The name space for texture objects is the unsigned integers, with zero reserved by the GL to represent the default texture object. The default texture object is bound to each of the TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_RECTANGLE, TEXTURE_BUFFER, TEXTURE_CUBE_MAP, TEXTURE_CUBE_MAP_ARRAY, TEXTURE_2D_MULTISAMPLE, and TEXTURE_2D_MULTISAMPLE_ARRAY targets during context initialization.

A new texture object is created by binding an unused name to one of these texture targets. The command

```c
void GenTextures( sizei n, uint *textures );
```

OpenGL 4.2 (Core Profile) - April 27, 2012
returns \( n \) previously unused texture names in \textit{textures}. These names are marked
as used, for the purposes of \texttt{GenTextures} only, but they acquire texture state and
a dimensionality only when they are first bound, just as if they were unused. The
binding is effected by calling

\begin{verbatim}
void BindTexture( enum target, uint texture );
\end{verbatim}

with \texttt{target} set to the desired texture target and \texttt{texture} set to the unused name. The
resulting texture object is a new state vector, comprising all the state and with the
same initial values listed in section 3.9.15 The new texture object bound to \texttt{target}
is, and remains a texture of the dimensionality and type specified by \texttt{target} until it
is deleted.

\texttt{BindTexture} may also be used to bind an existing texture object to any of
these targets. The error \texttt{INVALID\_OPERATION} is generated if an attempt is made
to bind a texture object of different dimensionality than the specified \texttt{target}. If the
bind is successful no change is made to the state of the bound texture object, and
any previous binding to \texttt{target} is broken.

\texttt{BindTexture} fails and an \texttt{INVALID\_OPERATION} error is generated if \texttt{texture}
is not zero or a name returned from a previous call to \texttt{GenTextures}, or if such a
name has since been deleted.

While a texture object is bound, GL operations on the target to which it is
bound affect the bound object, and queries of the target to which it is bound return
state from the bound object. If texture mapping of the dimensionality of the target
to which a texture object is bound is enabled, the state of the bound texture object
directs the texturing operation.

Texture objects are deleted by calling

\begin{verbatim}
void DeleteTextures( sizei \( n \), const uint *\textit{textures} );
\end{verbatim}

textures contains \( n \) names of texture objects to be deleted. After a texture object
is deleted, it has no contents or dimensionality, and its name is again unused. If
a texture that is currently bound to any of the target bindings of \texttt{BindTexture} is
deleted, it is as though \texttt{BindTexture} had been executed with the same target and
texture zero. Additionally, special care must be taken when deleting a texture if any
of the images of the texture are attached to a framebuffer object. See section 4.4.2
for details.

Unused names in \textit{textures} that have been marked as used for the purposes of
\texttt{GenTextures} are marked as unused again. Unused names in \textit{textures} are silently
ignored, as is the name zero.

The texture object name space, including the initial one-, two-, and three-
dimensional, one- and two-dimensional array, rectangular, buffer, cube map, cube
3.9. TEXTURING

map array, two-dimensional multisample, and two-dimensional multisample array texture objects, is shared among all texture units. A texture object may be bound to more than one texture unit simultaneously. After a texture object is bound, any GL operations on that target object affect any other texture units to which the same texture object is bound.

Texture binding is affected by the setting of the state ACTIVE_TEXTURE. If a texture object is deleted, it as if all texture units which are bound to that texture object are rebound to texture object zero.

3.9.2 Sampler Objects

The state necessary for texturing can be divided into two categories as described in section 3.9.15. A GL texture object includes both categories. The first category represents dimensionality and other image parameters, and the second category represents sampling state. Additionally, a sampler object may be created to encapsulate only the second category - the sampling state - of a texture object.

A new sampler object is created by binding an unused name to a texture unit. The command

```c
void GenSamplers(sizei count, uint *samplers);
```

returns `count` previously unused sampler object names in `samplers`. The name zero is reserved by the GL to represent no sampler being bound to a sampler unit. The names are marked as used, for the purposes of `GenSamplers` only, but they acquire state only when they are first used as a parameter to `BindSampler`, `SamplerParameter*`, `GetSamplerParameter*`, or `IsSampler`. When a sampler object is first used in one of these functions, the resulting sampler object is initialized with a new state vector, comprising all the state and with the same initial values listed in table 6.18.

When a sampler object is bound to a texture unit, its state supersedes that of the texture object bound to that texture unit. If the sampler name zero is bound to a texture unit, the currently bound texture’s sampler state becomes active. A single sampler object may be bound to multiple texture units simultaneously.

A sampler object binding is effected with the command

```c
void BindSampler(uint unit, uint sampler);
```

with `unit` set to the zero-based index of the texture unit to which to bind the sampler and `sampler` set to the name of a sampler object returned from a previous call to `GenSamplers`. 
3.9. TEXTURING

If the bind is successful no change is made to the state of the bound sampler object, and any previous binding to unit is broken.

**BindSampler** fails and an INVALID_OPERATION error is generated if sampler is not zero or a name returned from a previous call to *GenSamplers*, or if such a name has since been deleted with **DeleteSamplers**. An INVALID_VALUE error is generated if unit is greater than or equal to the value of MAX_COMBINED_TEXTURE_IMAGE_UNITS.

If state is present in a sampler object bound to a texture unit that would have been rejected by a call to **TexParameter** for the texture bound to that unit, the behavior of the implementation is as if the texture were incomplete. For example, if TEXTURE_WRAP_S or TEXTURE_WRAP_T or TEXTURE_WRAP_R is set to REPEAT or MIRRORED_REPEAT on the sampler object bound to a texture unit and the texture bound to that unit is a rectangular texture, the texture will be considered incomplete.

Sampler object state which does not affect sampling for the type of texture bound to a texture unit, such as TEXTURE_WRAP_R for a rectangular texture, does not affect completeness.

The currently bound sampler may be queried by calling **GetIntegerv** with pname set to **SAMPLER_BINDING**. When a sampler object is unbound from the texture unit (by binding another sampler object, or the sampler object named zero, to that texture unit) the modified state is again replaced with the sampler state associated with the texture object bound to that texture unit.

The parameters represented by a sampler object are a subset of those described in section 3.9.8. Each parameter of a sampler object is set by calling

```c
void SamplerParameter{if}v(uint sampler, enum pname, T param);
void SamplerParameter{if}v(uint sampler, enum pname, const T param);
void SamplerParameterI{i ui}v(uint sampler, enum pname, const T *params);
```

*sampler* is the name of a sampler object previously reserved by a call to **GenSamplers**. *pname* is the name of a parameter to modify and *param* is the new value of that parameter. An INVALID_OPERATION error is generated if sampler is not the name of a sampler object previously returned from a call to **GenSamplers**. The values accepted in the *pname* parameter are TEXTURE_WRAP_S, TEXTURE_WRAP_T, TEXTURE_WRAP_R, TEXTURE_MIN_FILTER, TEXTURE_MAG_FILTER, TEXTURE_BORDER_COLOR, TEXTURE_MIN_LOD, TEXTURE_MAX_LOD, TEXTURE_LOD_BIAS, TEXTURE_COMPARE_MODE, and...
3.9. **TEXTURING**

TEXTURE_COMPARE_FUNC. Texture state listed in table 6.17 but not listed here and in the sampler state in table 6.18 is not part of the sampler state, and remains in the texture object.

Data conversions are performed as specified in section 2.3.1, except that if the values for TEXTURE_BORDER_COLOR are specified with a call to SamplerParameterIiv or SamplerParameterIuiv, the values are unmodified and stored with an internal data type of integer. If specified with SamplerParameteriv, they are converted to floating-point using equation 2.1. Otherwise, border color values are unmodified and stored as floating-point.

An INVALID_ENUM error is generated if `pname` is not the name of a parameter accepted by SamplerParameter*. If the value of `param` is not an acceptable value for the parameter specified in `pname`, an error is generated as specified in the description of TexParameter*.

Modifying a parameter of a sampler object affects all texture units to which that sampler object is bound. Calling TexParameter has no effect on the sampler object bound to the active texture unit. It will modify the parameters of the texture object bound to that unit.

Sampler objects are deleted by calling

```c
void DeleteSamplers(sizei count, const uint *samplers);
```

`samplers` contains `count` names of sampler objects to be deleted. After a sampler object is deleted, its name is again unused. If a sampler object that is currently bound to a sampler unit is deleted, it is as though BindSampler is called with `unit` set to the unit the sampler is bound to and `sampler` zero. Unused names in `samplers` that have been marked as used for the purposes of GenSamplers are marked as unused again. Unused names in `samplers` are silently ignored, as is the reserved name zero.

### 3.9.3 Texture Image Specification

The command

```c
void TexImage3D(enum target, int level, int internalformat, sizei width, sizei height, sizei depth, int border, enum format, enum type, const void *data);
```

is used to specify a three-dimensional texture image. `target` must be one of TEXTURE_3D for a three-dimensional texture, TEXTURE_2D_ARRAY for an two-dimensional array texture, or TEXTURE_CUBE_MAP_ARRAY for a cube map array texture. Additionally, `target` may be either PROXY_TEXTURE_3D for a three-dimensional proxy texture, PROXY_TEXTURE_2D_ARRAY for a two-dimensional
proxy array texture, or PROXY_TEXTURE_CUBE_MAP_ARRAY for a cube map array texture, as discussed in section 3.9.15. format, type, and data specify the format of the image data, the type of those data, and a reference to the image data in the currently bound pixel unpack buffer or client memory, as described in section 3.7.2. The format STENCIL_INDEX is not allowed.

The groups in memory are treated as being arranged in a sequence of adjacent rectangles. Each rectangle is a two-dimensional image, whose size and organization are specified by the width and height parameters to TexImage3D. The values of UNPACK_ROW_LENGTH and UNPACK_ALIGNMENT control the row-to-row spacing in these images as described in section 3.7.2. If the value of the integer parameter UNPACK_IMAGE_HEIGHT is not positive, then the number of rows in each two-dimensional image is height; otherwise the number of rows is UNPACK_IMAGE_HEIGHT. Each two-dimensional image comprises an integral number of rows, and is exactly adjacent to its neighbor images.

The mechanism for selecting a sub-volume of a three-dimensional image relies on the integer parameter UNPACK_SKIP_IMAGES. If UNPACK_SKIP_IMAGES is positive, the pointer is advanced by UNPACK_SKIP_IMAGES times the number of elements in one two-dimensional image before obtaining the first group from memory. Then depth two-dimensional images are processed, each having a subimage extracted as described in section 3.7.2.

The selected groups are transferred to the GL as described in section 3.7.2 and then clamped to the representable range of the internal format. If the internalformat of the texture is signed or unsigned integer, components are clamped to \([-2^{n-1}, 2^{n-1} - 1]\) or \([0, 2^n - 1]\), respectively, where \(n\) is the number of bits per component. For color component groups, if the internalformat of the texture is signed or unsigned normalized fixed-point, components are clamped to \([-1, 1]\) or \([0, 1]\), respectively. For depth component groups, the depth value is clamped to \([0, 1]\). Otherwise, values are not modified. Stencil index values are masked by \(2^n - 1\), where \(n\) is the number of stencil bits in the internal format resolution (see below). If the base internal format is DEPTH_STENCIL and format is not DEPTH_STENCIL, then the values of the stencil index texture components are undefined.

Components are then selected from the resulting R, G, B, A, depth, or stencil values to obtain a texture with the base internal format specified by (or derived from) internalformat. Table 3.11 summarizes the mapping of R, G, B, A, depth, or stencil values to texture components, as a function of the base internal format of the texture image. internalformat may be specified as one of the internal format symbolic constants listed in table 3.11, as one of the sized internal format symbolic constants listed in tables 3.12- 3.13, as one of the generic compressed internal format symbolic constants listed in table 3.14, or as one of the specific compressed internal format symbolic constants (if listed in table 3.14). Specifying a value for
3.9. TEXTURING

<table>
<thead>
<tr>
<th>Base Internal Format</th>
<th>RGBA, Depth, and Stencil Values</th>
<th>Internal Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPTH_COMPONENT</td>
<td>Depth</td>
<td>D</td>
</tr>
<tr>
<td>DEPTH_STENCIL</td>
<td>Depth,Stencil</td>
<td>D, S</td>
</tr>
<tr>
<td>RED</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>RG</td>
<td>R, G</td>
<td>R, G</td>
</tr>
<tr>
<td>RGB</td>
<td>R, G, B</td>
<td>R, G, B</td>
</tr>
<tr>
<td>RGBA</td>
<td>R, G, B, A</td>
<td>R, G, B, A</td>
</tr>
</tbody>
</table>

Table 3.11: Conversion from RGBA, depth, and stencil pixel components to internal texture components. Texture components R, G, B, and A are converted back to RGBA colors during filtering as shown in table 3.23.

*internalformat* that is not one of the above values generates the error `INVALID_VALUE`.

Textures with a base internal format of `DEPTH_COMPONENT` or `DEPTH_STENCIL` are supported by texture image specification commands only if `target` is `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`, `TEXTURE_RECTANGLE`, `TEXTURE_CUBE_MAP`, `TEXTURE_CUBE_MAP_ARRAY`, `PROXY_TEXTURE_1D`, `PROXY_TEXTURE_2D`, `PROXY_TEXTURE_1D_ARRAY`, `PROXY_TEXTURE_2D_ARRAY`, `PROXY_TEXTURE_RECTANGLE`, `PROXY_TEXTURE_CUBE_MAP`, or `PROXY_TEXTURE_CUBE_MAP_ARRAY`. Using these formats in conjunction with any other `target` will result in an `INVALID_OPERATION` error.

Textures with a base internal format of `DEPTH_COMPONENT` or `DEPTH_STENCIL` require either depth component data or depth/stencil component data. Textures with other base internal formats require RGBA component data. The error `INVALID_OPERATION` is generated if one of the base internal format and `format` is `DEPTH_COMPONENT` or `DEPTH_STENCIL`, and the other is neither of these values.

Textures with integer internal formats (see table 3.12) require integer data. If the internal format is integer and `format` is not one of the integer formats listed in table 3.3, or if the internal format is not integer and `format` is an integer format, or if `format` is an integer format and `type` is `FLOAT`, `HALF_FLOAT`, `UNSIGNED_INT_10F_11F_11F_REV`, or `UNSIGNED_INT_5_9_9_9_REV`, an `INVALID_OPERATION` error is generated.

In addition to the specific compressed internal formats listed in table 3.14, the GL provides a mechanism to query token values for specific compressed internal
3.9. TEXTURING

formats, suitable for general-purpose\(^1\) usage. Formats with restrictions that need to be specifically understood prior to use will not be returned by this query. The number of specific compressed internal formats is obtained by querying the value of \texttt{NUM\_COMPRESSED\_TEXTURE\_FORMATS}. The set of specific compressed internal formats is obtained by querying \texttt{COMPRESSED\_TEXTURE\_FORMATS} with \texttt{GetInteger}, returning an array containing that number of values.

Generic compressed internal formats are never used directly as the internal formats of texture images. If \texttt{internalformat} is one of the six generic compressed internal formats, its value is replaced by the symbolic constant for a specific compressed internal format of the GL’s choosing with the same base internal format. If no specific compressed format is available, \texttt{internalformat} is instead replaced by the corresponding base internal format. If \texttt{internalformat} is given as or mapped to a specific compressed internal format, but the GL can not support images compressed in the chosen internal format for any reason (e.g., the compression format might not support 3D textures), \texttt{internalformat} is replaced by the corresponding base internal format and the texture image will not be compressed by the GL.

The \textit{internal component resolution} is the number of bits allocated to each value in a texture image. If \texttt{internalformat} is specified as a base internal format, the GL stores the resulting texture with internal component resolutions of its own choosing. If a sized internal format is specified, the mapping of the R, G, B, A, depth, and stencil values to texture components is equivalent to the mapping of the corresponding base internal format’s components, as specified in table 3.11; the type (unsigned int, float, etc.) is assigned the same type specified by \texttt{internalformat}; and the memory allocation per texture component is assigned by the GL to match the allocations listed in tables 3.12- 3.13 as closely as possible. (The definition of closely is left up to the implementation. However, a non-zero number of bits must be allocated for each component whose \textit{desired} allocation in tables 3.12- 3.13 is non-zero, and zero bits must be allocated for all other components).

\textbf{Required Texture Formats}

Implementations are required to support at least one allocation of internal component resolution for each type (unsigned int, float, etc.) for each base internal format.

In addition, implementations are required to support the following sized and compressed internal formats. Requesting one of these sized internal formats for any texture type will allocate at least the internal component sizes, and exactly the component types shown for that format in tables 3.12- 3.13:

\footnote{These queries have been deprecated in OpenGL 4.2, because the vagueness of the term “general-purpose” makes it possible for implementations to choose to return no formats from the query.}

OpenGL 4.2 (Core Profile) - April 27, 2012
3.9. TEXTURING

• Texture and renderbuffer color formats (see section 4.4.2).
  – RGBA32F, RGBA32I, RGBA32UI, RGBA16, RGBA16F, RGBA16I, 
    RGBA16UI, RGBA8, RGBA8I, RGBA8UI, SRGB8_ALPHA8, RGB10_A2, 
    RGB10_A2UI, RGB5_A1, and RGBA4.
  – RG32F, RG32I, RG32UI, RG16, RG16F, RG16I, RG16UI, RG8, RG8I, 
    and RG8UI.
  – R32F, R32I, R32UI, R16F, R16I, R16UI, R16, R8, R8I, and R8UI.

• Texture-only color formats:
  – RGBA16_SNORM and RGBA8_SNORM.
  – RGB32F, RGB32I, and RGB32UI.
  – RGB16_SNORM, RGB16F, RGB16I, RGB16UI, and RGB16.
  – RGB8_SNORM, RGB8, RGB8I, RGB8UI, and SRGB8.
  – RGB9_E5.
  – RG16_SNORM, RG8_SNORM, COMPRESSED_RG_RGTC2 and 
    COMPRESSED_SIGNED_RG_RGTC2.
  – R16_SNORM, R8_SNORM, COMPRESSED_RED_RGTC1 and 
    COMPRESSED_SIGNED_RED_RGTC1.

• Depth formats: DEPTH_COMPONENT32F, DEPTH_COMPONENT24, and 
  DEPTH_COMPONENT16.

• Combined depth+stencil formats: DEPTH32F_STENCIL8 and DEPTH24_-
 _STENCIL8.

Encoding of Special Internal Formats

If internalformat is R11F_G11F_B10F, the red, green, and blue bits are converted 
to unsigned 11-bit, unsigned 11-bit, and unsigned 10-bit floating-point values as 
described in sections 2.1.1 and 2.1.1.

If internalformat is RGB9_E5, the red, green, and blue bits are converted to a 
shared exponent format according to the following procedure:

Components red, green, and blue are first clamped (in the process, mapping 
NaN to zero) as follows:

OpenGL 4.2 (Core Profile) - April 27, 2012
3.9. TEXTURING

\[ red_c = \max(0, \min(\text{sharedexp}_{max}, \text{red})) \]
\[ green_c = \max(0, \min(\text{sharedexp}_{max}, \text{green})) \]
\[ blue_c = \max(0, \min(\text{sharedexp}_{max}, \text{blue})) \]

where
\[ \text{sharedexp}_{max} = \frac{(2^N - 1)2^{E_{max} - B}}{2^N} \]

\( N \) is the number of mantissa bits per component (9), \( B \) is the exponent bias (15), and \( E_{max} \) is the maximum allowed biased exponent value (31).

The largest clamped component, \( \max_c \), is determined:
\[ \max_c = \max(\text{red}_c, \text{green}_c, \text{blue}_c) \]

A preliminary shared exponent \( \text{exp}_p \) is computed:
\[ \text{exp}_p = \max(-B - 1, \lfloor \log_2(\max_c) \rfloor) + 1 + B \]

A refined shared exponent \( \text{exp}_s \) is computed:
\[ \max_s = \left\lfloor \frac{\max_c}{2^{\text{exp}_p - B - N}} + 0.5 \right\rfloor \]
\[ \text{exp}_s = \begin{cases} \text{exp}_p, & 0 \leq \max_s < 2^N \\ \text{exp}_p + 1, & \max_s = 2^N \end{cases} \]

Finally, three integer values in the range 0 to \( 2^N - 1 \) are computed:

\[ \text{red}_s = \left\lfloor \frac{\text{red}_c}{2^{\text{exp}_s - B - N}} + 0.5 \right\rfloor \]
\[ \text{green}_s = \left\lfloor \frac{\text{green}_c}{2^{\text{exp}_s - B - N}} + 0.5 \right\rfloor \]
\[ \text{blue}_s = \left\lfloor \frac{\text{blue}_c}{2^{\text{exp}_s - B - N}} + 0.5 \right\rfloor \]

The resulting \( \text{red}_s, \text{green}_s, \text{blue}_s \), and \( \text{exp}_s \) are stored in the red, green, blue, and shared bits respectively of the texture image.

An implementation accepting pixel data of type \texttt{UNSIGNED\_INT\_5\_9\_9\_9\_REV} with format \texttt{RGB} is allowed to store the components “as is”.

OpenGL 4.2 (Core Profile) - April 27, 2012
### Sized Internal Color Formats

<table>
<thead>
<tr>
<th>Sized Internal Format</th>
<th>Base Internal Format</th>
<th>R bits</th>
<th>G bits</th>
<th>B bits</th>
<th>A bits</th>
<th>Shared bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>R8</td>
<td>RED</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8_SNORM</td>
<td>RED</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16</td>
<td>RED</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16_SNORM</td>
<td>RED</td>
<td>s16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG8</td>
<td>RG</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG8_SNORM</td>
<td>RG</td>
<td>s8</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG16</td>
<td>RG</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG16_SNORM</td>
<td>RG</td>
<td>s16</td>
<td>s16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3_G3_B2</td>
<td>RGB</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB4</td>
<td>RGB</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB5</td>
<td>RGB</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB565</td>
<td>RGB</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB8</td>
<td>RGB</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB8_SNORM</td>
<td>RGB</td>
<td>s8</td>
<td>s8</td>
<td>s8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB10</td>
<td>RGB</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB12</td>
<td>RGB</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB16</td>
<td>RGB</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB16_SNORM</td>
<td>RGB</td>
<td>s16</td>
<td>s16</td>
<td>s16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGBA2</td>
<td>RGBA</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RGBA4</td>
<td>RGBA</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>RGB5_A1</td>
<td>RGBA</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>RGBA8</td>
<td>RGBA</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>RGBA8_SNORM</td>
<td>RGBA</td>
<td>s8</td>
<td>s8</td>
<td>s8</td>
<td>s8</td>
<td></td>
</tr>
<tr>
<td>RGB10_A2</td>
<td>RGBA</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RGB10_A2UI</td>
<td>RGBA</td>
<td>ui10</td>
<td>ui10</td>
<td>ui10</td>
<td>ui2</td>
<td></td>
</tr>
<tr>
<td>RGBA12</td>
<td>RGBA</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>RGBA16</td>
<td>RGBA</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>RGBA16_SNORM</td>
<td>RGBA</td>
<td>s16</td>
<td>s16</td>
<td>s16</td>
<td>s16</td>
<td></td>
</tr>
<tr>
<td>SRGB8</td>
<td>RGB</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRGB8_ALPHA8</td>
<td>RGBA</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16F</td>
<td>RED</td>
<td>f16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG16F</td>
<td>RG</td>
<td>f16</td>
<td>f16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB16F</td>
<td>RGB</td>
<td>f16</td>
<td>f16</td>
<td>f16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGBA16F</td>
<td>RGBA</td>
<td>f16</td>
<td>f16</td>
<td>f16</td>
<td>f16</td>
<td></td>
</tr>
<tr>
<td>R32F</td>
<td>RED</td>
<td>f32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sized internal color formats continued on next page
### Sized internal color formats continued from previous page

<table>
<thead>
<tr>
<th>Sized Internal Format</th>
<th>Base Internal Format</th>
<th>R bits</th>
<th>G bits</th>
<th>B bits</th>
<th>A bits</th>
<th>Shared bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>RG32F</td>
<td>RG</td>
<td>f32</td>
<td>f32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB32F</td>
<td>RGB</td>
<td>f32</td>
<td>f32</td>
<td>f32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGBA32F</td>
<td>RGBA</td>
<td>f32</td>
<td>f32</td>
<td>f32</td>
<td>f32</td>
<td></td>
</tr>
<tr>
<td>R11F_G11F_B10F</td>
<td>RGB</td>
<td>f11</td>
<td>f11</td>
<td>f10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB9_E5</td>
<td>RGB</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>R8I</td>
<td>RED</td>
<td>i8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8UI</td>
<td>RED</td>
<td>ui8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16I</td>
<td>RED</td>
<td>i16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16UI</td>
<td>RED</td>
<td>ui16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R32I</td>
<td>RED</td>
<td>i32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R32UI</td>
<td>RED</td>
<td>ui32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG8I</td>
<td>RG</td>
<td>i8</td>
<td>i8</td>
<td>i8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB8UI</td>
<td>RGB</td>
<td>ui8</td>
<td>ui8</td>
<td>ui8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB16I</td>
<td>RGB</td>
<td>i16</td>
<td>i16</td>
<td>i16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB16UI</td>
<td>RGB</td>
<td>ui16</td>
<td>ui16</td>
<td>ui16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB32I</td>
<td>RGB</td>
<td>i32</td>
<td>i32</td>
<td>i32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB32UI</td>
<td>RGB</td>
<td>ui32</td>
<td>ui32</td>
<td>ui32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGBA8I</td>
<td>RGBA</td>
<td>i8</td>
<td>i8</td>
<td>i8</td>
<td>i8</td>
<td></td>
</tr>
<tr>
<td>RGBA8UI</td>
<td>RGBA</td>
<td>ui8</td>
<td>ui8</td>
<td>ui8</td>
<td>ui8</td>
<td></td>
</tr>
<tr>
<td>RGBA16I</td>
<td>RGBA</td>
<td>i16</td>
<td>i16</td>
<td>i16</td>
<td>i16</td>
<td></td>
</tr>
<tr>
<td>RGBA16UI</td>
<td>RGBA</td>
<td>ui16</td>
<td>ui16</td>
<td>ui16</td>
<td>ui16</td>
<td></td>
</tr>
<tr>
<td>RGBA32I</td>
<td>RGBA</td>
<td>i32</td>
<td>i32</td>
<td>i32</td>
<td>i32</td>
<td></td>
</tr>
<tr>
<td>RGBA32UI</td>
<td>RGBA</td>
<td>ui32</td>
<td>ui32</td>
<td>ui32</td>
<td>ui32</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.12: Correspondence of sized internal color formats to base internal formats, internal data type, and desired component resolutions for each sized internal format. The component resolution prefix indicates the internal data type: f is floating-point, i is signed integer, ui is unsigned integer, s is signed normalized fixed-point, and no prefix is unsigned normalized fixed-point.
Table 3.13: Correspondence of sized internal depth and stencil formats to base internal formats, internal data type, and desired component resolutions for each sized internal format. The component resolution prefix indicates the internal data type: $f$ is floating-point, $i$ is signed integer, $ui$ is unsigned integer, and no prefix is fixed-point.

<table>
<thead>
<tr>
<th>Sized Internal Format</th>
<th>Base Internal Format</th>
<th>$D$ bits</th>
<th>$S$ bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPTH_COMPONENT16</td>
<td>DEPTH_COMPONENT</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>DEPTH_COMPONENT24</td>
<td>DEPTH_COMPONENT</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>DEPTH_COMPONENT32</td>
<td>DEPTH_COMPONENT</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>DEPTH_COMPONENT32F</td>
<td>DEPTH_COMPONENT</td>
<td>f32</td>
<td></td>
</tr>
<tr>
<td>DEPTH24_STENCIL8</td>
<td>DEPTH_STENCIL</td>
<td>24</td>
<td>8</td>
</tr>
<tr>
<td>DEPTH32F_STENCIL8</td>
<td>DEPTH_STENCIL</td>
<td>f32</td>
<td>8</td>
</tr>
</tbody>
</table>

If a compressed internal format is specified, the mapping of the R, G, B, and A values to texture components is equivalent to the mapping of the corresponding base internal format’s components, as specified in table 3.11. The specified image is compressed using a (possibly lossy) compression algorithm chosen by the GL.

A GL implementation may vary its allocation of internal component resolution or compressed internal format based on any `TexImage3D`, `TexImage2D` (see below), or `TexImage1D` (see below) parameter (except `target`), but the allocation and chosen compressed image format must not be a function of any other state and cannot be changed once they are established. In addition, the choice of a compressed image format may not be affected by the `data` parameter. Allocations must be invariant; the same allocation and compressed image format must be chosen each time a texture image is specified with the same parameter values. These allocation rules also apply to proxy textures, which are described in section 3.9.15.

The image itself (referred to by `data`) is a sequence of groups of values. The first group is the lower left back corner of the texture image. Subsequent groups fill out rows of width `width` from left to right; height rows are stacked from bottom to top forming a single two-dimensional image slice; and depth slices are stacked from back to front. When the final R, G, B, and A components have been computed for a group, they are assigned to components of a texel as described by table 3.11. Counting from zero, each resulting $N$th texel is assigned internal integer coordinates $(i, j, k)$, where
### 3.9. TEXTURING

<table>
<thead>
<tr>
<th>Compressed Internal Format</th>
<th>Base Internal Format</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPRESSED_RED</td>
<td>RED</td>
<td>Generic</td>
</tr>
<tr>
<td>COMPRESSED_RG</td>
<td>RG</td>
<td>Generic</td>
</tr>
<tr>
<td>COMPRESSED_RGB</td>
<td>RGB</td>
<td>Generic</td>
</tr>
<tr>
<td>COMPRESSED_RGBA</td>
<td>RGBA</td>
<td>Generic</td>
</tr>
<tr>
<td>COMPRESSED_SRGB</td>
<td>RGB</td>
<td>Generic</td>
</tr>
<tr>
<td>COMPRESSED_SRGB_ALPHA</td>
<td>RGBA</td>
<td>Generic</td>
</tr>
<tr>
<td>COMPRESSED_RED_RGTC1</td>
<td>RED</td>
<td>Specific</td>
</tr>
<tr>
<td>COMPRESSED_SIGNED_RED_RGTC1</td>
<td>RED</td>
<td>Specific</td>
</tr>
<tr>
<td>COMPRESSED_RG_RGTC2</td>
<td>RG</td>
<td>Specific</td>
</tr>
<tr>
<td>COMPRESSED_SIGNED_RG_RGTC2</td>
<td>RG</td>
<td>Specific</td>
</tr>
<tr>
<td>COMPRESSED_RGBA_BPTC_UNORM</td>
<td>RGBA</td>
<td>Specific</td>
</tr>
<tr>
<td>COMPRESSED_SRGB_ALPHA_BPTC_UNORM</td>
<td>RGBA</td>
<td>Specific</td>
</tr>
<tr>
<td>COMPRESSED_RGB_BPTC_SIGNED_FLOAT</td>
<td>RGB</td>
<td>Specific</td>
</tr>
<tr>
<td>COMPRESSED_RGB_BPTC_UNSIGNED_FLOAT</td>
<td>RGB</td>
<td>Specific</td>
</tr>
</tbody>
</table>

Table 3.14: Generic and specific compressed internal formats. The specific RGTC and BPTC formats are described in appendix C.

\[
i = (N \mod width) - w_b
\]

\[
j = \left\lfloor \frac{N}{width} \mod height \right\rfloor - h_b
\]

\[
k = \left\lfloor \frac{N}{width \times height} \mod depth \right\rfloor - d_b
\]

and \(w_b, h_b,\) and \(d_b\) are the specified border width, height, and depth. \(w_b\) and \(h_b\) are the specified border value; \(d_b\) is the specified border value if \(target\) is TEXTURE_3D, or zero if \(target\) is TEXTURE_2D_ARRAY or TEXTURE_CUBE_MAP_ARRAY. Thus the last two-dimensional image slice of the three-dimensional image is indexed with the highest value of \(k\).

When \(target\) is TEXTURE_CUBE_MAP_ARRAY, specifying a cube map array texture, \(k\) refers to a layer-face. The layer is given by

\[
layer = \left\lfloor \frac{k}{6} \right\rfloor
\]

and the face is given by
3.9. TEXTURING

\[ \text{face} = k \mod 6. \]

The face number corresponds to the cube map faces as shown in table 4.12.

If the internal data type of the image array is signed or unsigned normalized fixed-point, each color component is converted using equation 2.4 or 2.3, respectively. If the internal type is floating-point or integer, components are clamped to the representable range of the corresponding internal component, but are not converted.

The level argument to TexImage3D is an integer level-of-detail number. Levels of detail are discussed below, under Mipmapping. The main texture image has a level of detail number of 0. If a level-of-detail less than zero is specified, the error INVALID_VALUE is generated.

The border argument to TexImage3D is a border width. The significance of borders is described below. The border width affects the dimensions of the texture image: let

\[
egin{align*}
    w_s &= w_t + 2w_b \\
    h_s &= h_t + 2h_b \\
    d_s &= d_t + 2d_b
\end{align*}
\]

(3.16)

where \(w_s, h_s,\) and \(d_s\) are the specified image width, height, and depth, and \(w_t, h_t,\) and \(d_t\) are the dimensions of the texture image internal to the border. If \(w_t, h_t,\) or \(d_t\) are less than zero, then the error INVALID_VALUE is generated.

The maximum border width \(b_t\) is 0. If \(border\) is less than zero, or greater than \(b_t,\) then the error INVALID_VALUE is generated.

The maximum allowable width, height, or depth of a texel array for a three-dimensional texture is an implementation-dependent function of the level-of-detail and internal format of the resulting image array. It must be at least \(2^{k-\text{lod}} + 2b_t\) for image arrays of level-of-detail 0 through \(k,\) where \(k\) is the log base 2 of MAX_3D_TEXTURE_SIZE, \(lod\) is the level-of-detail of the image array, and \(b_t\) is the maximum border width. It may be zero for image arrays of any level-of-detail greater than \(k.\)

If width, height, or depth exceed the corresponding maximum size, an INVALID_VALUE error is generated. As described in section 3.9.14, these implementation-dependent limits may be configured to reject textures at level 1 or greater unless a mipmap complete set of image arrays consistent with the specified sizes can be supported.

When target is TEXTURE_CUBE_MAP_ARRAY or proxy_TEXTURE_CUBE_MAP_ARRAY width and height must be equal, otherwise the error INVALID_VALUE

OpenGL 4.2 (Core Profile) - April 27, 2012
is generated. Also, depth must be a multiple of six indicating $6N$ layer-faces in the cube map array, otherwise the error INVALID_VALUE is generated.

If a pixel unpack buffer object is bound and storing texture data would access memory beyond the end of the pixel unpack buffer, an INVALID_OPERATION error results.

In a similar fashion, the maximum allowable width of a texel array for a one- or two-dimensional, one- or two-dimensional array, two-dimensional multisample, or two-dimensional multisample array texture, and the maximum allowable height of a two-dimensional, two-dimensional array, two-dimensional multisample, or two-dimensional multisample array texture, must be at least $2^{k-\text{lod}} + 2b_t$ for image arrays of level 0 through $k$, where $k$ is the log base 2 of MAX_TEXTURE_SIZE.

The maximum allowable width and height of a cube map or cube map array texture must be the same, and must be at least $2^{k-\text{lod}} + 2b_t$ for image arrays level 0 through $k$, where $k$ is the log base 2 of the value of MAX_CUBE_MAP_TEXTURE_SIZE. The maximum number of layers for one- and two-dimensional array textures (height or depth, respectively), and the maximum number of layer-faces for cube map array textures (depth), must be at least the value of MAX_ARRAY_TEXTURE_LAYERS for all levels.

The maximum allowable width and height of a rectangular texture image must each be at least the value of the implementation-dependent constant MAX_RECTANGLE_TEXTURE_SIZE.

An implementation may allow an image array of level 0 to be created only if that single image array can be supported. Additional constraints on the creation of image arrays of level 1 or greater are described in more detail in section 3.9.14.

The command

```
void TexImage2D( enum target, int level, int internalformat,
    sizei width, sizei height, int border, enum format,
    enum type, const void *data );
```

is used to specify a two-dimensional texture image. target must be one of TEXTURE_2D for a two-dimensional texture, TEXTURE_1D_ARRAY for a one-dimensional array texture, TEXTURE_RECTANGLE for a rectangle texture, or one of TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z for a cube map texture. Additionally, target may be either PROXY_TEXTURE_2D for a two-dimensional proxy texture, PROXY_TEXTURE_1D_ARRAY for a one-dimensional proxy array texture, PROXY_TEXTURE_RECTANGLE for a rectangle proxy texture, or PROXY_TEXTURE_CUBE_MAP for a cube map proxy texture.
in the special case discussed in section 3.9.15. The other parameters match the corresponding parameters of `TexImage3D`.

For the purposes of decoding the texture image, `TexImage2D` is equivalent to calling `TexImage3D` with corresponding arguments and `depth` of 1, except that `UNPACK_SKIP_IMAGES` is ignored.

A two-dimensional or rectangle texture consists of a single two-dimensional texture image. A cube map texture is a set of six two-dimensional texture images. The six cube map texture targets form a single cube map texture though each target names a distinct face of the cube map. The `TEXTURE_CUBE_MAP_*` targets listed above update their appropriate cube map face 2D texture image. Note that the six cube map two-dimensional image tokens such as `TEXTURE_CUBE_MAP_POSITIVE_X` are used when specifying, updating, or querying one of a cube map’s six two-dimensional images, but when binding to a cube map texture object (that is when the cube map is accessed as a whole as opposed to a particular two-dimensional image), the `TEXTURE_CUBE_MAP` target is specified.

When the `target` parameter to `TexImage2D` is one of the six cube map two-dimensional image targets, the error `INVALID_VALUE` is generated if the `width` and `height` parameters are not equal.

When `target` is `TEXTURE_RECTANGLE`, an `INVALID_VALUE` error is generated if `level` is non-zero.

An `INVALID_VALUE` error is generated if `border` is non-zero.

Finally, the command

```c
void TexImage1D( enum target, int level,
                  int internalformat, sizei width, int border,
                  enum format, enum type, const void *data );
```

is used to specify a one-dimensional texture image. `target` must be either `TEXTURE_1D`, or `PROXY_TEXTURE_1D` in the special case discussed in section 3.9.15.

For the purposes of decoding the texture image, `TexImage1D` is equivalent to calling `TexImage2D` with corresponding arguments and `height` of 1.

The image indicated to the GL by the image pointer is decoded and copied into the GL’s internal memory.

We shall refer to the decoded image as the `texel array`. A three-dimensional texel array has width, height, and depth `w_s`, `h_s`, and `d_s` as defined in equation 3.16. A two-dimensional or rectangular texel array has depth `d_s = 1`, with height `h_s` and width `w_s` as above. A one-dimensional texel array has depth `d_s = 1`, height `h_s = 1`, and width `w_s` as above.
3.9. TEXTURING

Figure 3.7. A texture image and the coordinates used to access it. This is a two-dimensional texture with width 8 and height 4. A one-dimensional texture would consist of a single horizontal strip. \( \alpha \) and \( \beta \), values used in blending adjacent texels to obtain a texture value, are also shown.

An element \((i, j, k)\) of the texel array is called a texel (for a two-dimensional texture or one-dimensional array texture, \(k\) is irrelevant; for a one-dimensional texture, \(j\) and \(k\) are both irrelevant). The texture value used in texturing a fragment is determined by sampling the texture in a shader, but may not correspond to any actual texel. See figure 3.7. If target is TEXTURE_CUBE_MAP_ARRAY, the texture value is determined by \((s, t, r, q)\) coordinates where \(s\), \(t\), and \(r\) are defined to be the same as for TEXTURE_CUBE_MAP and \(q\) is defined as the index of a specific cube map in the cube map array.

If the data argument of TexImage1D, TexImage2D, or TexImage3D is NULL, a null pointer (a zero-valued pointer in the C implementation), and the pixel unpack buffer object is zero, a one-, two-, or three-dimensional texel array is created with the specified target, level, internalformat, border, width, height, and depth, but with unspecified image contents. In this case no pixel values are accessed in client memory, and no pixel processing is performed. Errors are generated, however, exactly as though the data pointer were valid. Otherwise if the pixel unpack
buffer object is non-zero, the data argument is treatedly normally to refer to the beginning of the pixel unpack buffer object’s data.

### 3.9.4 Alternate Texture Image Specification Commands

Two-dimensional and one-dimensional texture images may also be specified using image data taken directly from the framebuffer, and rectangular subregions of existing texture images may be respecified.

The command

```c
void CopyTexImage2D( enum target, int level,
                      enum internalformat, int x, int y, sizei width,
                     sizei height, int border );
```

defines a two-dimensional texel array in exactly the manner of TexImage2D, except that the image data are taken from the framebuffer rather than from client memory. Currently, target must be one of TEXTURE_2D, TEXTURE_1D_ARRAY, TEXTURE_RECTANGLE, TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_-MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_-Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z. x, y, width, and height correspond precisely to the corresponding arguments to ReadPixels (refer to section 4.3.1); they specify the image’s width and height, and the lower left (x, y) coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to CopyPixels (see section 4.3.2) with argument type set to COLOR, DEPTH, or DEPTH_STENCIL, depending on internalformat, stopping after conversion of depth values. RGBA data is taken from the current color buffer, while depth component and stencil index data are taken from the depth and stencil buffers, respectively. The error INVALID_OPERATION is generated if depth component data is required and no depth buffer is present; if stencil index data is required and no stencil buffer is present; if integer RGBA data is required and the format of the current color buffer is not integer; or if floating- or fixed-point RGBA data is required and the format of the current color buffer is integer.

Subsequent processing is identical to that described for TexImage2D, beginning with clamping of the R, G, B, A, or depth values, and masking of the stencil index values from the resulting pixel groups. Parameters level, internalformat, and border are specified using the same values, with the same meanings, as the equivalent arguments of TexImage2D. An invalid value specified for internalformat generates the error INVALID_ENUM. The constraints on width, height, and border are exactly those for the equivalent arguments of TexImage2D.
When the `target` parameter to `CopyTexImage2D` is one of the six cube map two-dimensional image targets, the error `INVALID_VALUE` is generated if the `width` and `height` parameters are not equal.

An `INVALID_FRAMEBUFFER_OPERATION` error will be generated if the object bound to `READ_FRAMEBUFFER_BINDING` (see section 4.4) is not framebuffer complete (as defined in section 4.4.4). An `INVALID_OPERATION` error will be generated if the object bound to `READ_FRAMEBUFFER_BINDING` is framebuffer complete and the value of `SAMPLE_BUFFERS` is greater than zero.

The command

```
void CopyTexImage1D( enum target, int level,
                     enum internalformat, int x, int y, sizei width,
                     int border );
```

defines a one-dimensional texel array in exactly the manner of `TexImage1D`, except that the image data are taken from the framebuffer, rather than from client memory. Currently, `target` must be `TEXTURE_1D`. For the purposes of decoding the texture image, `CopyTexImage1D` is equivalent to calling `CopyTexImage2D` with corresponding arguments and `height` of 1, except that the `height` of the image is always 1, regardless of the value of `border`, `level`, `internalformat`, and `border` are specified using the same values, with the same meanings, as the equivalent arguments of `TexImage1D`. The constraints on `width` and `border` are exactly those of the equivalent arguments of `TexImage1D`.

Six additional commands,

```
void TexSubImage3D( enum target, int level, int xoffset,
                    int yoffset, int zoffset, sizei width, sizei height,
                    sizei depth, enum format, enum type, const
                    void *data );
void TexSubImage2D( enum target, int level, int xoffset,
                    int yoffset, sizei width, sizei height, enum format,
                    enum type, const void *data );
void TexSubImage1D( enum target, int level, int xoffset,
                    sizei width, enum format, enum type, const
                    void *data );
void CopyTexSubImage3D( enum target, int level,
                        int xoffset, int yoffset, int zoffset, int x, int y,
                        sizei width, sizei height );
void CopyTexSubImage2D( enum target, int level,
                        int xoffset, int yoffset, int x, int y, sizei width,
                        sizei height );
```
3.9. TEXTURING

```c
void CopyTexSubImage1D( enum target, int level,
    int xoffset, int x, int y, sizei width );
```

respecify only a rectangular subregion of an existing texel array. No change is made to the internalformat, width, height, depth, or border parameters of the specified texel array, nor is any change made to texel values outside the specified subregion.

The target arguments of TexSubImage1D and CopyTexSubImage1D must be TEXTURE_1D, the target arguments of TexSubImage2D and CopyTexSubImage2D must be one of TEXTURE_2D, TEXTURE_1D_ARRAY, TEXTURE_RECTANGLE, TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z, and the target arguments of TexSubImage3D and CopyTexSubImage3D must be TEXTURE_3D, TEXTURE_2D_ARRAY, or TEXTURE_CUBE_MAP_ARRAY.

The level parameter of each command specifies the level of the texel array that is modified. If level is less than zero or greater than the base 2 logarithm of the maximum texture width, height, or depth, the error INVALID_VALUE is generated. If target is TEXTURE_RECTANGLE and level is not zero, the error INVALID_VALUE is generated. TexSubImage3D arguments width, height, depth, format, type, and data match the corresponding arguments to TexImage3D, meaning that they are specified using the same values, and have the same meanings. Likewise, TexSubImage2D arguments width, height, format, type, and data match the corresponding arguments to TexImage2D, and TexSubImage1D arguments width, format, type, and data match the corresponding arguments to TexImage1D.

CopyTexSubImage3D and CopyTexSubImage2D arguments x, y, width, and height match the corresponding arguments to CopyTexImage2D. CopyTexSubImage1D arguments x, y, and width match the corresponding arguments to CopyTexImage1D. Each of the TexSubImage commands interprets and processes pixel groups in exactly the manner of its TexImage counterpart, except that the assignment of R, G, B, A, depth, and stencil index pixel group values to the texture components is controlled by the internalformat of the texel array, not by an argument to the command. The same constraints and errors apply to the TexSubImage commands’ argument format and the internalformat of the texel array being re-specified as apply to the format and internalformat arguments of its TexImage counterparts.

Arguments xoffset, yoffset, and zoffset of TexSubImage3D and CopyTexImage3D specify the lower left texel coordinates of a width-wide by height-

---

2 Because the framebuffer is inherently two-dimensional, there is no CopyTexImage3D command.
high by depth-deep rectangular subregion of the texel array. For cube map array
textures, zoffset is the first layer-face to update, and depth is the number of layer-
faces to update. The depth argument associated with CopyTexSubImage3D is
always 1, because framebuffer memory is two-dimensional - only a portion of a
single \((s,t)\) slice of a three-dimensional texture is replaced by CopyTexSubImage3D.

Negative values of xoffset, yoffset, and zoffset correspond to the coordinates
of border texels, addressed as in figure 3.7. Taking \(w_s, h_s, d_s, w_b, h_b,\) and \(d_b\) to
be the specified width, height, depth, and border width, border height, and border
depth of the texel array, and taking \(x, y, z, w, h,\) and \(d\) to be the xoffset, yoffset,
zoffset, width, height, and depth argument values, any of the following relationships
generates the error INVALID_VALUE:

\[
\begin{align*}
x &< -w_b \\
x + w &> w_s - w_b \\
y &< -h_b \\
y + h &> h_s - h_b \\
z &< -d_b \\
z + d &> d_s - d_b
\end{align*}
\]

Counting from zero, the \(n\)th pixel group is assigned to the texel with internal integer
coordinates \([i, j, k]\), where

\[
\begin{align*}
i &= x + (n \mod w) \\
j &= y + \left(\left\lfloor \frac{n}{w} \right\rfloor \mod h\right) \\
k &= z + \left(\left\lfloor \frac{n \cdot width \cdot height}{d} \right\rfloor \mod d\right)
\end{align*}
\]

Arguments xoffset and yoffset of TexSubImage2D and CopyTexSubImage2D
specify the lower left texel coordinates of a width-wide by height-high rectangular
subregion of the texel array. Negative values of xoffset and yoffset correspond to the
coordinates of border texels, addressed as in figure 3.7. Taking \(w_s, h_s,\) and \(b_s\)
to be the specified width, height, and border width of the texel array, and taking \(x, y, w,\) and \(h\) to be the xoffset, yoffset, width, and height argument values, any of the
following relationships generates the error INVALID_VALUE:

\[
x < -b_s
\]
3.9. TEXTURING

\[
x + w > w_s - b_s \\
y < -b_s \\
y + h > h_s - b_s
\]

Counting from zero, the \( n \)th pixel group is assigned to the texel with internal integer coordinates \([i, j]\), where

\[
i = x + (n \mod w) \\
j = y + (\lfloor \frac{n}{w} \rfloor \mod h)
\]

The \textit{xoffset} argument of \texttt{TexSubImage1D} and \texttt{CopyTexSubImage1D} specifies the left texel coordinate of a \textit{width}-wide subregion of the texel array. Negative values of \textit{xoffset} correspond to the coordinates of border texels. Taking \( w_s \) and \( b_s \) to be the specified width and border width of the texel array, and \( x \) and \( w \) to be the \textit{xoffset} and \textit{width} argument values, either of the following relationships generates the error \texttt{INVALID\_VALUE}:

\[
x < -b_s \\
x + w > w_s - b_s
\]

Counting from zero, the \( n \)th pixel group is assigned to the texel with internal integer coordinates \([i]\), where

\[
i = x + (n \mod w)
\]

Texture images with compressed internal formats may be stored in such a way that it is not possible to modify an image with subimage commands without having to decompress and recompress the texture image. Even if the image were modified in this manner, it may not be possible to preserve the contents of some of the texels outside the region being modified. To avoid these complications, the GL does not support arbitrary modifications to texture images with compressed internal formats. Calling \texttt{TexSubImage3D}, \texttt{CopyTexSubImage3D}, \texttt{TexSubImage2D}, \texttt{CopyTexSubImage2D}, \texttt{TexSubImage1D}, or \texttt{CopyTexSubImage1D} will result in an \texttt{INVALID\_OPERATION} error if \textit{xoffset}, \textit{yoffset}, or \textit{zoffset} is not equal to \(-b_s\) (border width). In addition, the contents of any texel outside the region modified by such a call are undefined. These restrictions may be relaxed for specific compressed internal formats whose images are easily modified.

If the internal format of the texture image being modified is one of the specific \texttt{RGTC} or \texttt{BPTC} formats described in table 3.14, the texture is stored using the corresponding \texttt{RGTC} or \texttt{BPTC} texture image encoding (see appendix C). Since such
3.9. **TEXTURING**

Images are easily edited along $4 \times 4$ texel boundaries, the limitations on subimage location and size are relaxed for `TexSubImage2D`, `TexSubImage3D`, `CopyTexSubImage2D`, and `CopyTexSubImage3D`. These commands will generate an `INVALID_OPERATION` error if one of the following conditions occurs:

- `width` is not a multiple of four, `$width + xoffset$` is not equal to the value of `TEXTURE_WIDTH`, and either `$xoffset$` or `$yoffset$` is non-zero.
- `height` is not a multiple of four, `$height + yoffset$` is not equal to the value of `TEXTURE_HEIGHT`, and either `$xoffset$` or `$yoffset$` is non-zero.
- `$xoffset$` or `$yoffset$` is not a multiple of four.

The contents of any $4 \times 4$ block of texels of an RGTC or BPTC compressed texture image that does not intersect the area being modified are preserved during valid `TexSubImage*` and `CopyTexSubImage*` calls.

Calling `CopyTexSubImage3D`, `CopyTexImage2D`, `CopyTexSubImage2D`, `CopyTexImage1D`, or `CopyTexSubImage1D` will result in an `INVALID_FRAMEBUFFER_OPERATION` error if the object bound to `READ_FRAMEBUFFER_BINDING` is not framebuffer complete (see section 4.4.4).

**Texture Copying Feedback Loops**

Calling `CopyTexSubImage3D`, `CopyTexImage2D`, `CopyTexSubImage2D`, `CopyTexImage1D`, or `CopyTexSubImage1D` will result in undefined behavior if the destination texture image level is also bound to the selected read buffer (see section 4.3.1) of the read framebuffer. This situation is discussed in more detail in the description of feedback loops in section 4.4.3.

### 3.9.5 Compressed Texture Images

Texture images may also be specified or modified using image data already stored in a known compressed image format, such as the RGTC formats defined in appendix C, or additional formats defined by GL extensions.

The commands

```c
void CompressedTexImage1D( enum target, int level,
    enum internalformat, sizei width, int border,
    sizei imageSize, const void *data );

void CompressedTexImage2D( enum target, int level,
    enum internalformat, sizei width, sizei height,
    int border, sizei imageSize, const void *data );
```

OpenGL 4.2 (Core Profile) - April 27, 2012


3.9. TEXTURING

```c
void CompressedTexImage3D( enum target, int level,
    enum internalformat, sizei width, sizei height,
    sizei depth, int border, sizei imageSize, const
    void *data );
```

define one-, two-, and three-dimensional texture images, respectively, with incoming data stored in a specific compressed image format. The `target`, `level`, `internalformat`, `width`, `height`, `depth`, and `border` parameters have the same meaning as in `TexImage1D`, `TexImage2D`, and `TexImage3D`, except that compressed rectangular texture formats are not supported. `data` refers to compressed image data stored in the specific compressed image format corresponding to `internalformat`. If a pixel unpack buffer is bound (as indicated by a non-zero value of `PIXEL_UNPACK_BUFFER_BINDING`), `data` is an offset into the pixel unpack buffer and the compressed data is read from the buffer relative to this offset; otherwise, `data` is a pointer to client memory and the compressed data is read from client memory relative to the pointer.

If the `target` parameter to any of the `CompressedTexImage*nD` commands is `TEXTURE_RECTANGLE` or `PROXY_TEXTURE_RECTANGLE`, the error `INVALID_ENUM` is generated.

`internalformat` must be a supported specific compressed internal format. An `INVALID_ENUM` error will be generated if any other values, including any of the generic compressed internal formats, is specified.

For all other compressed internal formats, the compressed image will be decoded according to the specification defining the `internalformat` token. Compressed texture images are treated as an array of `imageSize` ubytes relative to `data`. If a pixel unpack buffer object is bound and `data + imageSize` is greater than the size of the pixel buffer, an `INVALID_OPERATION` error results. If the `imageSize` parameter is not consistent with the format, dimensions, and contents of the compressed image, an `INVALID_VALUE` error results. If the compressed image is not encoded according to the defined image format, the results of the call are undefined.

If the compressed data are arranged into fixed-size blocks of texels, the pixel storage modes can be used to select a sub-rectangle from a larger containing rectangle. These pixel storage modes operate in the same way as they do for `TexImage*D` and as described in section 3.7.2. In the remainder of this section, denote by `b_s`, `b_w`, `b_h`, and `b_d` the values of pixel storage modes `UNPACK_COMPRESSED_BLOCK_SIZE`, `UNPACK_COMPRESSED_BLOCK_WIDTH`, `UNPACK_COMPRESSED_BLOCK_HEIGHT`, and `UNPACK_COMPRESSED_BLOCK_DEPTH` respectively. `b_s` is the compressed block size in bytes; `b_w`, `b_h`, and `b_d` are the compressed block width, height, and depth in pixels.
3.9. TEXTURING

By default the pixel storage modes UNPACK_ROW_LENGTH, UNPACK_SKIP_ROWS, UNPACK_SKIP_PIXELS, UNPACK_IMAGE_HEIGHT and UNPACK_SKIP_IMAGES are ignored for compressed images. To enable UNPACK_SKIP_PIXELS and UNPACK_ROW_LENGTH, \( b_s \) and \( b_w \) must both be non-zero. To also enable UNPACK_SKIP_ROWS and UNPACK_IMAGE_HEIGHT, \( b_h \) must be non-zero. And to also enable UNPACK_SKIP_IMAGES, \( b_d \) must be non-zero. All parameters must be consistent with the compressed format to produce the desired results.

When selecting a sub-rectangle from a compressed image:

- the value of UNPACK_SKIP_PIXELS must be a multiple of \( b_w \);
- the value of UNPACK_SKIP_ROWS must be a multiple of \( b_h \) for CompressedTexImage2D and CompressedTexImage3D;
- the value of UNPACK_SKIP_IMAGES must be a multiple of \( b_d \) for CompressedTexImage3D.

The error INVALID_OPERATION will be generated if any of the previous conditions are violated.

For CompressedTexImage1D the imageSize parameter must be equal to

\[
\frac{\text{width}}{b_w}
\]

For CompressedTexImage2D the imageSize parameter must be equal to

\[
\frac{\text{width}}{b_w} \times \frac{\text{height}}{b_h}
\]

For CompressedTexImage3D the imageSize parameter must be equal to

\[
\frac{\text{width}}{b_w} \times \frac{\text{height}}{b_h} \times \frac{\text{depth}}{b_d}
\]

The error INVALID_VALUE will be generated if imageSize does not match this requirement when pixel storage modes are active.

Based on the definition of unpacking from section 3.7.2 for uncompressed images, unpacking compressed images can be defined where:

- \( n \), the number of elements in a group, is 1
- \( s \), the size of an element, is \( b_s \)
3.9. TEXTURING

- \( l \), the number of groups in a row, is

\[
    l = \begin{cases} 
    \left\lceil \frac{\text{row length}}{b_w} \right\rceil, & \text{row length} > 0 \\
    \left\lceil \frac{\text{length}}{b_w} \right\rceil, & \text{otherwise}
    \end{cases}
\]

where \( \text{row length} \) is the value of \texttt{UNPACK\_ROW\_LENGTH}.

- \( a \), the value of \texttt{UNPACK\_ALIGNMENT}, is ignored and

- \( k = n \times l \) as is defined for uncompressed images.

Before obtaining the first compressed image block from memory, the \textit{data} pointer is advanced by

\[
    \frac{\text{UNPACK\_SKIP\_PIXELS}}{b_w} \times n + \frac{\text{UNPACK\_SKIP\_ROWS}}{b_h} \times k
\]

elements. Then \( \left\lceil \frac{\text{width}}{b_w} \right\rceil \) blocks are obtained from contiguous blocks in memory (without advancing the pointer), after which the pointer is advanced by \( k \) elements. \( \left\lceil \frac{\text{height}}{b_h} \right\rceil \) sets of \( \left\lceil \frac{\text{width}}{b_w} \right\rceil \) blocks are obtained this way. For three-dimensional compressed images the pointer is advanced by \( \frac{\text{UNPACK\_SKIP\_IMAGES}}{b_d} \) times the number of elements in one two-dimensional image before obtaining the first group from memory. Then after \( \text{height} \) rows are obtained the pointer skips over the remaining \( \left\lceil \frac{\text{UNPACK\_IMAGE\_HEIGHT}}{b_h} \right\rceil \) rows, if \texttt{UNPACK\_IMAGE\_HEIGHT} is positive, before starting the next two-dimensional image.

Specific compressed internal formats may impose format-specific restrictions on the use of the compressed image specification calls or parameters. For example, the compressed image format might be supported only for 2D textures, or might not allow non-zero \textit{border} values. Any such restrictions will be documented in the extension specification defining the compressed internal format; violating these restrictions will result in an \texttt{INVALID\_OPERATION} error.

Any restrictions imposed by specific compressed internal formats will be invariant, meaning that if the GL accepts and stores a texture image in compressed form, providing the same image to \texttt{CompressedTexImage1D}, \texttt{CompressedTexImage2D}, or \texttt{CompressedTexImage3D} will not result in an \texttt{INVALID\_OPERATION} error if the following restrictions are satisfied:

- \textit{data} points to a compressed texture image returned by \texttt{GetCompressedTexImage} (section 6.1.4).
3.9. TEXTURING

- `target`, `level`, and `internalformat` match the `target`, `level` and `format` parameters provided to the `GetCompressedTexImage` call returning `data`.

- `width`, `height`, `depth`, `internalformat`, and `imageSize` match the values of `TEXTURE_WIDTH`, `TEXTURE_HEIGHT`, `TEXTURE_DEPTH`, `TEXTURE_INTERNAL_FORMAT`, and `TEXTURE_COMPRESSED_IMAGE_SIZE` for image level `level` in effect at the time of the `GetCompressedTexImage` call returning `data`.

This guarantee applies not just to images returned by `GetCompressedTexImage`, but also to any other properly encoded compressed texture image of the same size and format.

If `internalformat` is one of the specific RGTC or BPTC formats described in table 3.14, the compressed image data is stored using the corresponding texture image encoding (see appendix C). The RGTC and BPTC texture compression algorithms support only two-dimensional images without borders, though 3D images can be compressed as multiple slices of compressed 2D BPTC images. If `internalformat` is an RGTC format, `CompressedTexImage1D` will generate an `INVALID_ENUM` error; `CompressedTexImage2D` will generate an `INVALID_OPERATION` error if `border` is non-zero or `target` is `TEXTURE_RECTANGLE`; and `CompressedTexImage3D` will generate an `INVALID_OPERATION` error if `border` is non-zero or `target` is not `TEXTURE_2D_ARRAY`. If `internalformat` is a BPTC format, `CompressedTexImage1D` will generate an `INVALID_ENUM` error; `CompressedTexImage2D` and `CompressedTexImage3D` will generate an `INVALID_OPERATION` error if `border` is non-zero.

If the `data` argument of `CompressedTexImage1D`, `CompressedTexImage2D`, or `CompressedTexImage3D` is `NULL`, a null pointer (a zero-valued pointer in the C implementation), and the pixel unpack buffer object is zero, a texel array with unspecified image contents is created, just as when a `NULL` pointer is passed to `TexImage1D`, `TexImage2D`, or `TexImage3D`.

The commands

```c
void CompressedTexImage1D( enum target, int level,
    int xoffset, sizei width, enum format, sizei imageSize,
    const void *data );
void CompressedTexImage2D( enum target, int level,
    int xoffset, sizei width, sizei height,
    enum format, sizei imageSize, const void *data );
void CompressedTexImage3D( enum target, int level,
    int xoffset, int yoffset, int zoffset, sizei width,
```
respecify only a rectangular region of an existing texel array, with incoming data
stored in a known compressed image format. The target, level, xoffset, yoffset, zoff-
set, width, height, and depth parameters have the same meaning as in TexSubImage1D, TexSubImage2D, and TexSubImage3D. data points to compressed image
data stored in the compressed image format corresponding to format. Using any of
the generic compressed internal formats as format will result in an INVALID_ENUM
error.

If the target parameter to any of the CompressedTexSubImage\textsubscript{nD} commands is TEXTURE_RECTANGLE or PROXY_TEXTURE_RECTANGLE, the error
INVALID_ENUM is generated.

The image pointed to by data and the imageSize parameter are interpreted
as though they were provided to CompressedTexImage1D, CompressedTexImage2D, and CompressedTexImage3D. These commands do not provide for im-
age format conversion, so an INVALID_OPERATION error results if format does
not match the internal format of the texture image being modified. If the image-
Size parameter is not consistent with the format, dimensions, and contents of the
compressed image (too little or too much data), an INVALID_VALUE error results.

As with CompressedTexImage calls, compressed internal formats may have
additional restrictions on the use of the compressed image specification calls or
parameters. Any such restrictions will be documented in the specification defining the compressed internal format; violating these restrictions will result in an
INVALID_OPERATION error.

Any restrictions imposed by specific compressed internal formats will be in-
variant, meaning that if the GL accepts and stores a texture image in compressed
form, providing the same image to CompressedTexSubImage1D, Compressed-
TexSubImage2D, CompressedTexSubImage3D will not result in an INVALID_-
OPERATION error if the following restrictions are satisfied:

- data points to a compressed texture image returned by GetCompressedTex-
Image (section 6.1.4).
- target, level, and format match the target, level and format parameters pro-
vided to the GetCompressedTexImage call returning data.
- width, height, depth, format, and imageSize match the values of TEXTURE-
WIDTH, TEXTURE_HEIGHT, TEXTURE_DEPTH, TEXTURE_INTERNAL_FORMAT, and TEXTURE_COMPRESSED_IMAGE_SIZE for image level level in effect at the time of the GetCompressedTexImage call returning data.
• width, height, depth, and format match the values of TEXTURE_WIDTH, TEXTURE_HEIGHT, TEXTURE_DEPTH, and TEXTURE_INTERNAL_FORMAT currently in effect for image level level.

• xoffset, yoffset, and zoffset are all zero.

This guarantee applies not just to images returned by GetCompressedTexImage, but also to any other properly encoded compressed texture image of the same size.

Calling CompressedTexSubImage3D, CompressedTexSubImage2D, or CompressedTexSubImage1D will result in an INVALID_OPERATION error if xoffset, yoffset, or zoffset are not equal to zero, or if width, height, and depth do not match the values of TEXTURE_WIDTH, TEXTURE_HEIGHT, or TEXTURE_DEPTH, respectively. The contents of any texel outside the region modified by the call are undefined. These restrictions may be relaxed for specific compressed internal formats whose images are easily modified.

If the internal format of the texture image being modified is one of the specific RGTC or BPTC formats described in table 3.14, the texture is stored using the corresponding texture image encoding (see appendix C). If the internal format is an RGTC format, CompressedTexSubImage1D will generate an INVALID_ENUM error; CompressedTexSubImage2D will generate an INVALID_OPERATION error if border is non-zero, and CompressedTexSubImage3D will generate an INVALID_OPERATION error if border is non-zero or target is not TEXTURE_2D_ARRAY. If the internal format is a BPTC format, CompressedTexSubImage1D will generate an INVALID_ENUM error. Since RGTC and BPTC images are easily edited along 4 × 4 texel boundaries, the limitations on subimage location and size are relaxed for CompressedTexSubImage2D and CompressedTexSubImage3D. These commands will result in an INVALID_OPERATION error if one of the following conditions occurs:

• width is not a multiple of four, and width + xoffset is not equal to the value of TEXTURE_WIDTH.

• height is not a multiple of four, and height + yoffset is not equal to the value of TEXTURE_HEIGHT.

• xoffset or yoffset is not a multiple of four.

The contents of any 4 × 4 block of texels of an RGTC compressed texture image that does not intersect the area being modified are preserved during valid TexSubImage* and CopyTexSubImage* calls.
3.9.6 Multisample Textures

In addition to the texture types described in previous sections, two additional types of textures are supported. A multisample texture is similar to a two-dimensional or two-dimensional array texture, except it contains multiple samples per texel. Multisample textures do not have multiple image levels.

The commands

```c
void TexImage2DMultisample( enum target, sizei samples,
    int internalformat, sizei width, sizei height,
    boolean fixedsamplelocations );

void TexImage3DMultisample( enum target, sizei samples,
    int internalformat, sizei width, sizei height,
    sizei depth, boolean fixedsamplelocations );
```

establish the data storage, format, dimensions, and number of samples of a multisample texture’s image. For `TexImage2DMultisample`, `target` must be `TEXTURE_2D_MULTISAMPLE` or `PROXY_TEXTURE_2D_MULTISAMPLE` and for `TexImage3DMultisample` `target` must be `TEXTURE_2D_MULTISAMPLE_ARRAY` or `PROXY_TEXTURE_2D_MULTISAMPLE_ARRAY`. `width` and `height` are the dimensions in texels of the texture.

`samples` represents a request for a desired minimum number of samples. Since different implementations may support different sample counts for multisampled textures, the actual number of samples allocated for the texture image is implementation-dependent. However, the resulting value for `TEXTURE_SAMPLES` is guaranteed to be greater than or equal to `samples` and no more than the next larger sample count supported by the implementation.

`internalformat` must be color-renderable, depth-renderable, or stencil-renderable (as defined in section 4.4.4).

If `fixedsamplelocations` is `TRUE`, the image will use identical sample locations and the same number of samples for all texels in the image, and the sample locations will not depend on the `internalformat` or size of the image.

Upon success, `TexImage*Multisample` deletes any existing image for `target` and the contents of texels are undefined. `TEXTURE_WIDTH`, `TEXTURE_HEIGHT`, `TEXTURE_SAMPLES`, `TEXTURE_INTERNAL_FORMAT` and `TEXTURE_FIXED_SAMPLE_LOCATIONS` are set to `width`, `height`, the actual number of samples allocated, `internalformat`, and `fixedsamplelocations` respectively.

If either `width` or `height` is greater than the value of `MAX_TEXTURE_SIZE`, or if `samples` is zero, an `INVALID_VALUE` error is generated. If `samples` is greater than the maximum number of samples supported for this `target` and `internalformat`, an `INVALID_OPERATION` error is generated. The maximum number of samples
supported can be determined by calling \texttt{GetInternalformat} with a \texttt{pname} of \texttt{SAMPLES} (see section 6.1.15). If the GL is unable to create a texture image of the requested size, an \texttt{OUT_OF_MEMORY} error is generated.

When a multisample texture is accessed in a shader, the access takes one vector of integers describing which texel to fetch and an integer corresponding to the sample numbers described in section 3.3.1 describing which sample within the texel to fetch. No standard sampling instructions are allowed on the multisample texture targets. Fetching a sample numbers less than zero, or greater than or equal to the number of samples in the texture, produces undefined results.

3.9.7 Buffer Textures

In addition to one-, two-, and three-dimensional, one- and two-dimensional array, and cube map textures described in previous sections, one additional type of texture is supported. A buffer texture is similar to a one-dimensional texture. However, unlike other texture types, the texel array is not stored as part of the texture. Instead, a buffer object is attached to a buffer texture and the texel array is taken from that buffer object’s data store. When the contents of a buffer object’s data store are modified, those changes are reflected in the contents of any buffer texture to which the buffer object is attached. Buffer textures do not have multiple image levels; only a single data store is available.

The command

\begin{verbatim}
void TexBuffer( enum target, enum internalformat, uint buffer );
\end{verbatim}

attaches the storage for the buffer object named \texttt{buffer} to the active buffer texture, and specifies an internal format for the texel array found in the attached buffer object. If \texttt{buffer} is zero, any buffer object attached to the buffer texture is detached, and no new buffer object is attached. If \texttt{buffer} is non-zero, but is not the name of an existing buffer object, the error \texttt{INVALID_OPERATION} is generated. \texttt{target} must be \texttt{TEXTURE_BUFFER}. \texttt{internalformat} specifies the storage format, and must be one of the sized internal formats found in table 3.15.

When a buffer object is attached to a buffer texture, the buffer object’s data store is taken as the texture’s texel array. The number of texels in the buffer texture’s texel array is given by

\[ \frac{\text{buffer\_size}}{\text{components} \times \text{sizeof(base\_type)}} \]

where \texttt{buffer\_size} is the size of the buffer object, in basic machine units and \texttt{components} and \texttt{base\_type} are the element count and base data type for elements,
as specified in table 3.15. The number of texels in the texel array is then clamped to the implementation-dependent limit MAX_TEXTURE_BUFFER_SIZE. When a buffer texture is accessed in a shader, the results of a texel fetch are undefined if the specified texel coordinate is negative, or greater than or equal to the clamped number of texels in the texel array.

When a buffer texture is accessed in a shader, an integer is provided to indicate the texel coordinate being accessed. If no buffer object is bound to the buffer texture, the results of the texel access are undefined. Otherwise, the attached buffer object’s data store is interpreted as an array of elements of the GL data type corresponding to internalformat. Each texel consists of one to four elements that are mapped to texture components (R, G, B, and A). Element \( m \) of the texel numbered \( n \) is taken from element \( n \times \text{components} + m \) of the attached buffer object’s data store. Elements and texels are both numbered starting with zero. For texture formats with signed or unsigned normalized fixed-point components, the extracted values are converted to floating-point using equations 2.2 or 2.1, respectively. The components of the texture are then converted to a \((R, G, B, A)\) vector according to table 3.15, and returned to the shader as a four-component result vector with components of the appropriate data type for the texture’s internal format. The base data type, component count, normalized component information, and mapping of data store elements to texture components is specified in table 3.15.

<table>
<thead>
<tr>
<th>Sized Internal Format</th>
<th>Base Type</th>
<th>Components</th>
<th>Norm</th>
<th>Component 0</th>
<th>Component 1</th>
<th>Component 2</th>
<th>Component 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>R8</td>
<td>ubyte</td>
<td>1</td>
<td>Yes</td>
<td>R 0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16</td>
<td>ushort</td>
<td>1</td>
<td>Yes</td>
<td>R 0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16F</td>
<td>half</td>
<td>1</td>
<td>No</td>
<td>R 0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R32F</td>
<td>float</td>
<td>1</td>
<td>No</td>
<td>R 0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8I</td>
<td>byte</td>
<td>1</td>
<td>No</td>
<td>R 0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16I</td>
<td>short</td>
<td>1</td>
<td>No</td>
<td>R 0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R32I</td>
<td>int</td>
<td>1</td>
<td>No</td>
<td>R 0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8UI</td>
<td>ubyte</td>
<td>1</td>
<td>No</td>
<td>R 0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R16UI</td>
<td>ushort</td>
<td>1</td>
<td>No</td>
<td>R 0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R32UI</td>
<td>uint</td>
<td>1</td>
<td>No</td>
<td>R 0 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG8</td>
<td>ubyte</td>
<td>2</td>
<td>Yes</td>
<td>R G 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG16</td>
<td>ushort</td>
<td>2</td>
<td>Yes</td>
<td>R G 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG16F</td>
<td>half</td>
<td>2</td>
<td>No</td>
<td>R G 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG32F</td>
<td>float</td>
<td>2</td>
<td>No</td>
<td>R G 0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Continued on next page)
3.9. TEXTURING

<table>
<thead>
<tr>
<th>Sized Internal Format</th>
<th>Base Type</th>
<th>Components</th>
<th>Norm</th>
<th>Component 0</th>
<th>Component 1</th>
<th>Component 2</th>
<th>Component 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>RG8I</td>
<td>byte</td>
<td>2</td>
<td>No</td>
<td>R</td>
<td>G</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RG16I</td>
<td>short</td>
<td>2</td>
<td>No</td>
<td>R</td>
<td>G</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RG32I</td>
<td>int</td>
<td>2</td>
<td>No</td>
<td>R</td>
<td>G</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RG8UI</td>
<td>ubyte</td>
<td>2</td>
<td>No</td>
<td>R</td>
<td>G</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RG16UI</td>
<td>ushort</td>
<td>2</td>
<td>No</td>
<td>R</td>
<td>G</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RG32UI</td>
<td>uint</td>
<td>2</td>
<td>No</td>
<td>R</td>
<td>G</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RGB32F</td>
<td>float</td>
<td>3</td>
<td>No</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>RGB32I</td>
<td>int</td>
<td>3</td>
<td>No</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>RGB32UI</td>
<td>uint</td>
<td>3</td>
<td>No</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>RGBA8</td>
<td>ubyte</td>
<td>4</td>
<td>Yes</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>RGBA16</td>
<td>ushort</td>
<td>4</td>
<td>Yes</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>RGBA16F</td>
<td>half</td>
<td>4</td>
<td>No</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>RGBA32F</td>
<td>float</td>
<td>4</td>
<td>No</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>RGBA8I</td>
<td>byte</td>
<td>4</td>
<td>No</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>RGBA16I</td>
<td>short</td>
<td>4</td>
<td>No</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>RGBA32I</td>
<td>int</td>
<td>4</td>
<td>No</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>RGBA8UI</td>
<td>ubyte</td>
<td>4</td>
<td>No</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>RGBA16UI</td>
<td>ushort</td>
<td>4</td>
<td>No</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>RGBA32UI</td>
<td>uint</td>
<td>4</td>
<td>No</td>
<td>R</td>
<td>G</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

Table 3.15: Internal formats for buffer textures. For each format, the data type of each element is indicated in the “Base Type” column and the element count is in the “Components” column. The “Norm” column indicates whether components should be treated as normalized floating-point values. The “Component 0, 1, 2, and 3” columns indicate the mapping of each element of a texel to texture components.

In addition to attaching buffer objects to textures, buffer objects can be bound to the buffer object target named TEXTURE_BUFFER, in order to specify, modify, or read the buffer object’s data store. The buffer object bound to TEXTURE_BUFFER has no effect on rendering. A buffer object is bound to TEXTURE_BUFFER by calling BindBuffer with target set to TEXTURE_BUFFER, as described in section 2.9.
3.9. TEXTURING

3.9.8 Texture Parameters

Various parameters control how the texel array is treated when specified or changed, and when applied to a fragment. Each parameter is set by calling

```c
void TexParameter{if}(enum target, enum pname, T param);
void TexParameter{if}v( enum target, enum pname, const T params );
void TexParameterI{ui}v( enum target, enum pname, const T* params );
```

`target` is the target, either `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`, `TEXTURE_RECTANGLE`, `TEXTURE_CUBE_MAP`, or `TEXTURE_CUBE_MAP_ARRAY`, `params` is a symbolic constant indicating the parameter to be set; the possible constants and corresponding parameters are summarized in table 3.16. In the first form of the command, `param` is a value to which to set a single-valued parameter; in the remaining forms, `params` is an array of parameters whose type depends on the parameter being set.

Data conversions are performed as specified in section 2.3.1.

In addition, if the values for `TEXTURE_BORDER_COLOR` are specified with `TexParameterIiv` or `TexParameterIuiv`, the values are unmodified and stored with an internal data type of integer. If specified with `TexParameteriv`, they are converted to floating-point using equation 2.2. Otherwise the values are unmodified and stored as floating-point.

If `pname` is `TEXTURE_SWIZZLE_RGBA`, `params` is an array of four enums which respectively set the `TEXTURE_SWIZZLE_R`, `TEXTURE_SWIZZLE_G`, `TEXTURE_SWIZZLE_B`, and `TEXTURE_SWIZZLE_A` parameters simultaneously.

An `INVALID_ENUM` error is generated if the type of the parameter specified by `pname` is `enum`, and the value(s) specified by `param` or `params` are not among the legal values shown in table 3.16.

An `INVALID_ENUM` error is generated if `TexParameter{if}` is called for a non-scalar parameter (`TEXTURE_BORDER_COLOR` or `TEXTURE_SWIZZLE_RGBA`, or if `TexParameter{*}` is called for a scalar parameter (all other parameters in table 3.16).

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Legal Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXTURE_BASE_LEVEL</td>
<td>int</td>
<td>any non-negative integer</td>
</tr>
<tr>
<td>TEXTURE_BORDER_COLOR</td>
<td>4 floats,</td>
<td>any 4 values</td>
</tr>
<tr>
<td></td>
<td>ints, or uints</td>
<td></td>
</tr>
</tbody>
</table>

Texture parameters continued on next page

OpenGL 4.2 (Core Profile) - April 27, 2012
3.9. TEXTURING

Table 3.16: Texture parameters and their values.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Legal Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXTURE_COMPARE_MODE</td>
<td>enum</td>
<td>NONE, COMPARE_REF_TO_TEXTURE</td>
</tr>
<tr>
<td>TEXTURE_COMPARE_FUNC</td>
<td>enum</td>
<td>LEQUAL, GEQUAL, LESS, GREATER, EQUAL, NOTEQUAL, ALWAYS, NEVER</td>
</tr>
<tr>
<td>TEXTURE_LOD_BIAS</td>
<td>float</td>
<td>any value</td>
</tr>
<tr>
<td>TEXTURE_MAG_FILTER</td>
<td>enum</td>
<td>NEAREST, LINEAR</td>
</tr>
<tr>
<td>TEXTURE_MAX_LEVEL</td>
<td>int</td>
<td>any non-negative integer</td>
</tr>
<tr>
<td>TEXTURE_MAX_LOD</td>
<td>float</td>
<td>any value</td>
</tr>
<tr>
<td>TEXTURE_MIN_FILTER</td>
<td>enum</td>
<td>NEAREST, LINEAR, Nearest_Mipmap_Nearest, Nearest_Mipmap_Linear, Linear_Mipmap_Nearest, Linear_Mipmap_Linear,</td>
</tr>
<tr>
<td>TEXTURE_MIN_LOD</td>
<td>float</td>
<td>any value</td>
</tr>
<tr>
<td>TEXTURE_SWIZZLE_R</td>
<td>enum</td>
<td>RED, GREEN, BLUE, ALPHA, ZERO, ONE</td>
</tr>
<tr>
<td>TEXTURE_SWIZZLE_G</td>
<td>enum</td>
<td>RED, GREEN, BLUE, ALPHA, ZERO, ONE</td>
</tr>
<tr>
<td>TEXTURE_SWIZZLE_B</td>
<td>enum</td>
<td>RED, GREEN, BLUE, ALPHA, ZERO, ONE</td>
</tr>
<tr>
<td>TEXTURE_SWIZZLE_A</td>
<td>enum</td>
<td>RED, GREEN, BLUE, ALPHA, ZERO, ONE</td>
</tr>
<tr>
<td>TEXTURE_SWIZZLE_RGBA</td>
<td>4 enums</td>
<td>RED, GREEN, BLUE, ALPHA, ZERO, ONE</td>
</tr>
<tr>
<td>TEXTURE_WRAP_S</td>
<td>enum</td>
<td>CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER, MIRRORED_REPEAT</td>
</tr>
<tr>
<td>TEXTURE_WRAP_T</td>
<td>enum</td>
<td>CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER, MIRRORED_REPEAT</td>
</tr>
<tr>
<td>TEXTURE_WRAP_R</td>
<td>enum</td>
<td>CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER, MIRRORED_REPEAT</td>
</tr>
</tbody>
</table>
3.9. TEXTURING

In the remainder of section 3.9, denote by $\text{lod}_{\text{min}}$, $\text{lod}_{\text{max}}$, $\text{level}_{\text{base}}$, and $\text{level}_{\text{max}}$ the values of the texture parameters TEXTURE_MIN_LOD, TEXTURE_MAX_LOD, TEXTURE_BASE_LEVEL, and TEXTURE_MAX_LEVEL respectively. However, if TEXTURE_IMMUTABLE_FORMAT is TRUE, then $\text{level}_{\text{base}}$ is clamped to the range $[0, \text{levels} - 1]$ and $\text{level}_{\text{max}}$ is then clamped to the range $[\text{level}_{\text{base}}, \text{levels} - 1]$, where levels is the parameter passed to TexStorage* for the texture object (see section 3.9.16).

Texture parameters for a cube map texture apply to the cube map as a whole; the six distinct two-dimensional texture images use the texture parameters of the cube map itself.

When target is TEXTURE_RECTANGLE, certain texture parameter values may not be specified. In this case, the error INVALID_ENUM is generated if the TEXTURE_WRAP_S or TEXTURE_WRAP_T parameter is set to REPEAT or MIRRORED_REPEAT. The error INVALID_ENUM is generated if TEXTURE_MIN_FILTER is set to a value other than NEAREST or LINEAR (no mipmap filtering is permitted). The error INVALID_VALUE is generated if TEXTURE_BASE_LEVEL is set to any value other than zero.

3.9.9 Depth Component Textures

Depth textures and the depth components of depth/stencil textures can be treated as RED textures during texture filtering and application (see section 3.9.17). The initial state for depth and depth/stencil textures treats them as RED textures.

3.9.10 Cube Map Texture Selection

When cube map texturing is enabled, the $(s \ t \ r)$ texture coordinates are treated as a direction vector $(r_x \ r_y \ r_z)$ emanating from the center of a cube. The $q$ coordinate is ignored. If the $q$ coordinate is negative, then the values of the direction vector become undefined. At texture application time, the interpolated per-fragment direction vector selects one of the cube map face’s two-dimensional images based on the largest magnitude coordinate direction (the major axis direction). If two or more coordinates have the identical magnitude, the implementation may define the rule to disambiguate this situation. The rule must be deterministic and depend only on $(r_x \ r_y \ r_z)$. The target column in table 3.17 explains how the major axis direction maps to the two-dimensional image of a particular cube map target.

Using the $s_c$, $t_c$, and $m_a$ determined by the major axis direction as specified in table 3.17, an updated $(s \ t)$ is calculated as follows:
3.9. TEXTURING

<table>
<thead>
<tr>
<th>Major Axis Direction</th>
<th>Target</th>
<th>$s_c$</th>
<th>$t_c$</th>
<th>$m_a$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+r_x$</td>
<td>TEXTURE_CUBE_MAP_POSITIVE_X</td>
<td>$-r_z$</td>
<td>$-r_y$</td>
<td>$r_x$</td>
</tr>
<tr>
<td>$-r_x$</td>
<td>TEXTURE_CUBE_MAP_NEGATIVE_X</td>
<td>$r_z$</td>
<td>$-r_y$</td>
<td>$r_x$</td>
</tr>
<tr>
<td>$+r_y$</td>
<td>TEXTURE_CUBE_MAP_POSITIVE_Y</td>
<td>$r_x$</td>
<td>$r_y$</td>
<td>$r_y$</td>
</tr>
<tr>
<td>$-r_y$</td>
<td>TEXTURE_CUBE_MAP_NEGATIVE_Y</td>
<td>$r_x$</td>
<td>$-r_z$</td>
<td>$r_y$</td>
</tr>
<tr>
<td>$+r_z$</td>
<td>TEXTURE_CUBE_MAP_POSITIVE_Z</td>
<td>$r_x$</td>
<td>$-r_y$</td>
<td>$r_z$</td>
</tr>
<tr>
<td>$-r_z$</td>
<td>TEXTURE_CUBE_MAP_NEGATIVE_Z</td>
<td>$-r_x$</td>
<td>$-r_y$</td>
<td>$r_z$</td>
</tr>
</tbody>
</table>

Table 3.17: Selection of cube map images based on major axis direction of texture coordinates.

$$s = \frac{1}{2} \left( \frac{s_c}{m_a} + 1 \right)$$

$$t = \frac{1}{2} \left( \frac{t_c}{m_a} + 1 \right)$$

Seamless Cube Map Filtering

Seamless cube map filtering is enabled or disabled by calling **Enable** or **Disable**, respectively, with the symbolic constant TEXTURE_CUBE_MAP_SEAMLESS.

When seamless cube map filtering is disabled, the new $(s \ t)$ is used to find a texture value in the determined face’s two-dimensional image using the rules given in sections 3.9.11 through 3.9.12.

When seamless cube map filtering is enabled, the rules for texel selection in sections 3.9.11 through 3.9.12 are modified so that texture wrap modes are ignored. Instead,

- If NEAREST filtering is done within a miplevel, always apply wrap mode CLAMP_TO_EDGE.

- If LINEAR filtering is done within a miplevel, always apply wrap mode CLAMP_TO_BORDER. Then,
  
  - If a texture sample location would lie in the texture border in either $u$ or $v$, instead select the corresponding texel from the appropriate neighboring face.
  
  - If a texture sample location would lie in the texture border in both $u$ and $v$ (in one of the corners of the cube), there is no unique neighboring face from which to extract one texel. The recommended method to
generate this texel is to average the values of the three available samples. However, implementations are free to construct this fourth texel in another way, so long as, when the three available samples have the same value, this texel also has that value.

The required state is one bit indicating whether seamless cube map filtering is enabled or disabled. Initially, it is disabled.

3.9.11 Texture Minification

Applying a texture to a primitive implies a mapping from texture image space to framebuffer image space. In general, this mapping involves a reconstruction of the sampled texture image, followed by a homogeneous warping implied by the mapping to framebuffer space, then a filtering, followed finally by a resampling of the filtered, warped, reconstructed image before applying it to a fragment. In the GL this mapping is approximated by one of two simple filtering schemes. One of these schemes is selected based on whether the mapping from texture space to framebuffer space is deemed to magnify or minify the texture image.

Scale Factor and Level of Detail

The choice is governed by a scale factor \( \rho(x, y) \) and the level-of-detail parameter \( \lambda(x, y) \), defined as

\[
\lambda_{\text{base}}(x, y) = \log_2[\rho(x, y)]
\]

\[
\lambda'(x, y) = \lambda_{\text{base}}(x, y) + \text{clamp}(\text{bias}_{\text{texobj}} + \text{bias}_{\text{shader}})
\]

\[
\lambda = \begin{cases} 
lod_{\text{max}}, & \lambda' > lod_{\text{max}} \\
\lambda', & lod_{\text{min}} \leq \lambda' \leq lod_{\text{max}} \\
lod_{\text{min}}, & \lambda' < lod_{\text{min}} \\
\text{undefined}, & lod_{\text{min}} > lod_{\text{max}} 
\end{cases}
\]

\( \text{bias}_{\text{texobj}} \) is the value of \( \text{TEXTURE\_LOD\_BIAS} \) for the bound texture object (as described in section 3.9.8). \( \text{bias}_{\text{shader}} \) is the value of the optional bias parameter in the texture lookup functions available to fragment shaders. If the texture access is performed in a fragment shader without a provided bias, or outside a fragment shader, then \( \text{bias}_{\text{shader}} \) is zero. The sum of these values is clamped to the range...
3.9. TEXTURING

$[-bias_{\text{max}}, bias_{\text{max}}]$ where $bias_{\text{max}}$ is the value of the implementation defined constant `MAX_TEXTURE_LOD_BIAS`.

If $\lambda(x, y)$ is less than or equal to the constant $c$ (see section 3.9.12) the texture is said to be magnified; if it is greater, the texture is minified. Sampling of minified textures is described in the remainder of this section, while sampling of magnified textures is described in section 3.9.12.

The initial values of $lod_{\text{min}}$ and $lod_{\text{max}}$ are chosen so as to never clamp the normal range of $\lambda$. They may be respecified for a specific texture by calling `TexParameter*` with `pname` set to `TEXTURE_MIN_LOD` or `TEXTURE_MAX_LOD` respectively.

Let $s(x, y)$ be the function that associates an $s$ texture coordinate with each set of window coordinates $(x, y)$ that lie within a primitive; define $t(x, y)$ and $r(x, y)$ analogously. Let

$$
\begin{align*}
  u(x, y) &= \begin{cases} 
  s(x, y) + \delta_u, & \text{rectangular texture} \\
  w_t \times s(x, y) + \delta_u, & \text{otherwise}
  \end{cases} \\
  v(x, y) &= \begin{cases} 
  t(x, y) + \delta_v, & \text{rectangular texture} \\
  h_t \times t(x, y) + \delta_v, & \text{otherwise}
  \end{cases} \\
  w(x, y) &= d_t \times r(x, y) + \delta_w
\end{align*}
$$

(3.20)

where $w_t$, $h_t$, and $d_t$ are as defined by equation 3.16 with $w_s$, $h_s$, and $d_s$ equal to the width, height, and depth of the image array whose level is $level_{\text{base}}$. For a one-dimensional or one-dimensional array texture, define $v(x, y) = 0$ and $w(x, y) = 0$; for a two-dimensional, two-dimensional array, rectangular, cube map, or cube map array texture, define $w(x, y) = 0$.

$(\delta_u, \delta_v, \delta_w)$ are the texel offsets specified in the OpenGL Shading Language texture lookup functions that support offsets. If the texture function used does not support offsets, all three shader offsets are taken to be zero.

If the value of any non-ignored component of the offset vector operand is outside implementation-dependent limits, the results of the texture lookup are undefined. For all instructions except `textureGather`, the limits are the values of `MIN_PROGRAM_TEXEL_OFFSET` and `MAX_PROGRAM_TEXEL_OFFSET`. For the `textureGather` instruction, the limits are the values of `MIN_PROGRAM_TEXTURE_GATHER_OFFSET` and `MAX_PROGRAM_TEXTURE_GATHER_OFFSET`.

For a polygon or point, $\rho$ is given at a fragment with window coordinates $(x, y)$
by

\[ \rho = \max \left\{ \sqrt{\left( \frac{\partial u}{\partial x} \right)^2 + \left( \frac{\partial v}{\partial x} \right)^2 + \left( \frac{\partial w}{\partial x} \right)^2}, \sqrt{\left( \frac{\partial u}{\partial y} \right)^2 + \left( \frac{\partial v}{\partial y} \right)^2 + \left( \frac{\partial w}{\partial y} \right)^2} \right\} \]

(3.21)

where \( \partial u/\partial x \) indicates the derivative of \( u \) with respect to window \( x \), and similarly for the other derivatives.

For a line, the formula is

\[ \rho = \sqrt{\left( \frac{\partial u}{\partial x} \Delta x + \frac{\partial u}{\partial y} \Delta y \right)^2 + \left( \frac{\partial v}{\partial x} \Delta x + \frac{\partial v}{\partial y} \Delta y \right)^2 + \left( \frac{\partial w}{\partial x} \Delta x + \frac{\partial w}{\partial y} \Delta y \right)^2} / l, \]

(3.22)

where \( \Delta x = x_2 - x_1 \) and \( \Delta y = y_2 - y_1 \) with \((x_1, y_1)\) and \((x_2, y_2)\) being the segment’s window coordinate endpoints and \( l = \sqrt{\Delta x^2 + \Delta y^2} \).

While it is generally agreed that equations 3.21 and 3.22 give the best results when texturing, they are often impractical to implement. Therefore, an implementation may approximate the ideal \( \rho \) with a function \( f(x, y) \) subject to these conditions:

1. \( f(x, y) \) is continuous and monotonically increasing in each of \( |\partial u/\partial x|, |\partial u/\partial y|, |\partial v/\partial x|, |\partial v/\partial y|, |\partial w/\partial x|, \) and \( |\partial w/\partial y| \)

2. Let

\[ m_u = \max \left\{ \left| \frac{\partial u}{\partial x} \right|, \left| \frac{\partial u}{\partial y} \right| \right\} \]

\[ m_v = \max \left\{ \left| \frac{\partial v}{\partial x} \right|, \left| \frac{\partial v}{\partial y} \right| \right\} \]

\[ m_w = \max \left\{ \left| \frac{\partial w}{\partial x} \right|, \left| \frac{\partial w}{\partial y} \right| \right\}. \]

Then \( \max\{m_u, m_v, m_w\} \leq f(x, y) \leq m_u + m_v + m_w \).
3.9. TEXTURING

Coordinate Wrapping and Texel Selection

After generating \(u(x, y), v(x, y),\) and \(w(x, y),\) they may be clamped and wrapped before sampling the texture, depending on the corresponding texture wrap modes.

Let \(u'(x, y) = u(x, y), v'(x, y) = v(x, y),\) and \(w'(x, y) = w(x, y).\)

The value assigned to \texttt{TEXTURE_MIN_FILTER} is used to determine how the texture value for a fragment is selected.

When the value of \texttt{TEXTURE_MIN_FILTER} is \texttt{NEAREST}, the texel in the image array of level \texttt{levelbase} that is nearest (in Manhattan distance) to \((u', v', w')\) is obtained. Let \((i, j, k)\) be integers such that

\[
\begin{align*}
i &= \text{wrap}(\lfloor u'(x, y) \rfloor) \\
j &= \text{wrap}(\lfloor v'(x, y) \rfloor) \\
k &= \text{wrap}(\lfloor w'(x, y) \rfloor)
\end{align*}
\]

and the value returned by \texttt{wrap()} is defined in table 3.18. For a three-dimensional texture, the texel at location \((i, j, k)\) becomes the texture value. For two-dimensional, two-dimensional array, rectangular, or cube map textures, \(k\) is irrelevant, and the texel at location \((i, j)\) becomes the texture value. For one-dimensional texture or one-dimensional array textures, \(j\) and \(k\) are irrelevant, and the texel at location \(i\) becomes the texture value.

For one- and two-dimensional array textures, the texel is obtained from image layer \(l,\) where

\[
l = \begin{cases} 
\text{clamp}(\lfloor t + 0.5 \rfloor, 0, h_t - 1), & \text{for one-dimensional array textures} \\
\text{clamp}(\lfloor r + 0.5 \rfloor, 0, d_t - 1), & \text{for two-dimensional array textures}
\end{cases}
\]

If the selected \((i, j, k), (i, j),\) or \(i\) location refers to a border texel that satisfies any of the conditions

\[
\begin{align*}
i &< -b_s & i &\geq w_t + b_s \\
j &< -b_s & j &\geq h_t + b_s \\
k &< -b_s & k &\geq d_t + b_s
\end{align*}
\]

then the border values defined by \texttt{TEXTURE_BORDER_COLOR} are used in place of the non-existent texel. If the texture contains color components, the values of \texttt{TEXTURE_BORDER_COLOR} are interpreted as an RGBA color to match the texture’s internal format in a manner consistent with table 3.11. The internal data type of the...
### 3.9. TEXTURING

<table>
<thead>
<tr>
<th>Wrap mode</th>
<th>Result of ( \text{wrap}(\text{coord}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLAMP_TO_EDGE</td>
<td>( \text{clamp}(\text{coord}, 0, \text{size} - 1) )</td>
</tr>
<tr>
<td>CLAMP_TO_BORDER</td>
<td>( \text{clamp}(\text{coord}, -1, \text{size}) )</td>
</tr>
<tr>
<td>REPEAT</td>
<td>( \text{fmod}(\text{coord}, \text{size}) )</td>
</tr>
<tr>
<td>MIRRORED_REPEAT</td>
<td>( (\text{size} - 1) - \text{mirror}(\text{fmod}(\text{coord}, 2 \times \text{size}) - \text{size}) )</td>
</tr>
</tbody>
</table>

Table 3.18: Texel location wrap mode application. \( \text{fmod}(a, b) \) returns \( a - b \times \lfloor \frac{a}{b} \rfloor \). \( \text{mirror}(a) \) returns \( a \) if \( a \geq 0 \), and \(- (1 + a)\) otherwise. The values of \( \text{mode} \) and \( \text{size} \) are \( \text{TEXTURE_WRAP}_S \) and \( w_t \), \( \text{TEXTURE_WRAP}_T \) and \( h_t \), and \( \text{TEXTURE_WRAP}_R \) and \( d_t \) when wrapping \( i \), \( j \), or \( k \) coordinates, respectively.

Border values must be consistent with the type returned by the texture as described in section 3.9, or the result is undefined. Border values are clamped before they are used, according to the format in which texture components are stored. For signed and unsigned normalized fixed-point formats, border values are clamped to \([-1, 1]\) and \([0, 1]\), respectively. For floating-point and integer formats, border values are clamped to the representable range of the format. If the texture contains depth components, the first component of \( \text{TEXTURE_BORDER_COLOR} \) is interpreted as a depth value.

When the value of \( \text{TEXTURE_MIN_FILTER} \) is \( \text{LINEAR} \), a \( 2 \times 2 \times 2 \) cube of texels in the image array of level \( \text{level}_{\text{base}} \) is selected. Let

\[
\begin{align*}
i_0 &= \text{wrap}([u' - 0.5]) \\
\beta &= \text{frac}(u' - 0.5) \\
j_0 &= \text{wrap}([v' - 0.5]) \\
\gamma &= \text{frac}(v' - 0.5) \\
k_0 &= \text{wrap}([w' - 0.5]) \\
\alpha &= \text{frac}(w' - 0.5) \\
i_1 &= \text{wrap}([u' - 0.5] + 1) \\
j_1 &= \text{wrap}([v' - 0.5] + 1) \\
k_1 &= \text{wrap}([w' - 0.5] + 1)
\end{align*}
\]

where \( \text{frac}(x) \) denotes the fractional part of \( x \).
For a three-dimensional texture, the texture value $\tau$ is found as

$$
\tau = (1 - \alpha)(1 - \beta)(1 - \gamma)\tau_{i0j0k0} + \alpha(1 - \beta)(1 - \gamma)\tau_{i1j0k0} \\
+ (1 - \alpha)\beta(1 - \gamma)\tau_{i0j1k0} + \alpha\beta(1 - \gamma)\tau_{i1j1k0} \\
+ (1 - \alpha)(1 - \beta)\gamma\tau_{i0j0k1} + \alpha(1 - \beta)\gamma\tau_{i1j0k1} \\
+ (1 - \alpha)\beta\gamma\tau_{i0j1k1} + \alpha\beta\gamma\tau_{i1j1k1}
$$

(3.23)

where $\tau_{ijk}$ is the texel at location $(i, j, k)$ in the three-dimensional texture image.

For a two-dimensional, two-dimensional array, rectangular, or cube map texture,

$$
\tau = (1 - \alpha)(1 - \beta)\tau_{i0j0} + \alpha(1 - \beta)\tau_{i1j0} \\
+ (1 - \alpha)\beta\tau_{i0j1} + \alpha\beta\tau_{i1j1}
$$

where $\tau_{ij}$ is the texel at location $(i, j)$ in the two-dimensional texture image. For two-dimensional array textures, all texels are obtained from layer $l$, where

$$
l = \text{clamp}(\lfloor r + 0.5 \rfloor, 0, d_t - 1).
$$

The \text{textureGather} and \text{textureGatherOffset} built-in shader functions return a vector derived from sampling a $2 \times 2$ block of texels in the image array of level $level_{base}$. The rules for the \text{LINEAR} minification filter are applied to identify the four selected texels. Each texel is then converted to a texture source color ($R_s, G_s, B_s, A_s$) according to table 3.23 and then swizzled as described in section 3.10.2. A four-component vector is then assembled by taking the $R_s$ component from the swizzled texture source colors of the four texels, in the order $\tau_{i0j1}$, $\tau_{i1j1}$, $\tau_{i0j0}$, and $\tau_{i1j0}$ (see figure 3.8). Incomplete textures (see section 3.9.14) are considered to return a texture source color of $(0, 0, 0, 1)$ for all four source texels.

And for a one-dimensional or one-dimensional array texture,

$$
\tau = (1 - \alpha)\tau_i + \alpha\tau_{i1}
$$

where $\tau_i$ is the texel at location $i$ in the one-dimensional texture. For one-dimensional array textures, both texels are obtained from layer $l$, where

$$
l = \text{clamp}(\lfloor t + 0.5 \rfloor, 0, h_t - 1).
$$

For any texel in the equation above that refers to a border texel outside the defined range of the image, the texel value is taken from the texture border color as with \text{NEAREST} filtering.
Figure 3.8. An example of an $8 \times 8$ texture image and the components returned for `textureGather`. The vector $(X, Y, Z, W)$ is returned, where each component is taken from the post-swizzle $R$ component of the corresponding texel.
3.9. TEXTURING

Rendering Feedback Loops

If all of the following conditions are satisfied, then the value of the selected \( \tau_{ijk} \), \( \tau_{ij} \), or \( \tau_i \) in the above equations is undefined instead of referring to the value of the texel at location \((i,j,k)\), \((i,j)\), or \((i)\) respectively. This situation is discussed in more detail in the description of feedback loops in section 4.4.3.

- The current DRAW_FRAMEBUFFER_BINDING names a framebuffer object \( F \).
- The texture is attached to one of the attachment points, \( A \), of framebuffer object \( F \).
- The value of TEXTURE_MIN_FILTER is NEAREST or LINEAR, and the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL for attachment point \( A \) is equal to \( level_{base} \)

-or-

The value of TEXTURE_MIN_FILTER is NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, or LINEAR_MIPMAP_LINEAR, and the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL for attachment point \( A \) is within the the inclusive range from \( level_{base} \) to \( q \).

Mipmapping

TEXTURE_MIN_FILTER values NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, and LINEAR_MIPMAP_LINEAR each require the use of a mipmap. Rectangular textures do not support mipmapping (it is an error to specify a minification filter that requires mipmapping). A mipmap is an ordered set of arrays representing the same image; each array has a resolution lower than the previous one. If the image array of level \( level_{base} \) has dimensions \( w_t \times h_t \times d_t \), then there are \( \lfloor \log_2(maxsize) \rfloor + 1 \) levels in the mipmap. where

\[
maxsize = \begin{cases} 
  w_t, & \text{for 1D and 1D array textures} \\
  \max(w_t, h_t), & \text{for 2D, 2D array, cube map, and cube map array textures} \\
  \max(w_t, h_t, d_t), & \text{for 3D textures}
\end{cases}
\]

Numbering the levels such that level \( level_{base} \) is the 0th level, the \( i \)th array has dimensions...
3.9. TEXTURING

\[
\max(1, \lfloor \frac{w_t}{w_d} \rfloor) \times \max(1, \lfloor \frac{h_t}{h_d} \rfloor) \times \max(1, \lfloor \frac{d_t}{d_d} \rfloor)
\]

where

\[
w_d = 2^i
\]

\[
h_d = \begin{cases} 1, & \text{for 1D and 1D array textures} \\ 2^i, & \text{otherwise} \end{cases}
\]

\[
d_d = \begin{cases} 2^i, & \text{for 3D textures} \\ 1, & \text{otherwise} \end{cases}
\]

until the last array is reached with dimension \(1 \times 1 \times 1\).

Each array in a mipmap is defined using \texttt{TexImage3D}, \texttt{TexImage2D}, \texttt{CopyTexImage2D}, \texttt{TexImage1D}, or \texttt{CopyTexImage1D} or by functions that are defined in terms of these functions. The array being set is indicated with the level-of-detail argument \texttt{level}. Level-of-detail numbers proceed from \texttt{levelbase} for the original texel array through the maximum level \(p\), with each unit increase indicating an array of half the dimensions of the previous one (rounded down to the next integer if fractional) as already described. For immutable-format textures, \(p\) is one less than the \texttt{levels} parameter passed to \texttt{TexStorage*}; otherwise \(p = \lfloor \log_2(\text{maxsize}) \rfloor + \texttt{levelbase}\). All arrays from \texttt{levelbase} through \(q = \min\{p, \texttt{levelmax}\}\) must be defined, as discussed in section 3.9.14.

The values of \texttt{levelbase} and \texttt{levelmax} may be respecified for a specific texture by calling \texttt{TexParameter*} with \texttt{pname} set to \texttt{TEXTURE_BASE_LEVEL} or \texttt{TEXTURE_MAX_LEVEL} respectively.

The error \texttt{INVALID_VALUE} is generated if either value is negative.

The mipmap is used in conjunction with the level of detail to approximate the application of an appropriately filtered texture to a fragment. Let \(c\) be the value of \(\lambda\) at which the transition from minification to magnification occurs (since this discussion pertains to minification, we are concerned only with values of \(\lambda\) where \(\lambda > c\)).

For mipmap filters \texttt{NEAREST_MIPMAP_NEAREST} and \texttt{LINEAR_MIPMAP_NEAREST}, the \(d\)th mipmap array is selected, where

\[
d = \begin{cases} \texttt{levelbase}, & \lambda \leq \frac{1}{2} \\ \lfloor \texttt{levelbase} + \lambda + \frac{1}{2} \rfloor - 1, & \lambda > \frac{1}{2}, \texttt{levelbase} + \lambda \leq q + \frac{1}{2} \\ q, & \lambda > \frac{1}{2}, \texttt{levelbase} + \lambda > q + \frac{1}{2} \end{cases}
\] (3.24)

OpenGL 4.2 (Core Profile) - April 27, 2012
3.9. TEXTURING

The rules for NEAREST or LINEAR filtering are then applied to the selected array. Specifically, the coordinate \((u, v, w)\) is computed as in equation 3.20, with \(w_s, h_s,\) and \(d_s\) equal to the width, height, and depth of the image array whose level is \(d\).

For mipmap filters NEAREST_MIPMAP_LINEAR and LINEAR_MIPMAP_LINEAR, the level \(d_1\) and \(d_2\) mipmap arrays are selected, where

\[
d_1 = \begin{cases} 
q, & \text{level}_\text{base} + \lambda \geq q \\
\lfloor \text{level}_\text{base} + \lambda \rfloor, & \text{otherwise}
\end{cases} \quad (3.25)
\]

\[
d_2 = \begin{cases} 
q, & \text{level}_\text{base} + \lambda \geq q \\
\lfloor \text{level}_\text{base} + \lambda \rfloor + 1, & \text{otherwise}
\end{cases} \quad (3.26)
\]

The rules for NEAREST or LINEAR filtering are then applied to each of the selected arrays, yielding two corresponding texture values \(\tau_1\) and \(\tau_2\). Specifically, for level \(d_1\), the coordinate \((u, v, w)\) is computed as in equation 3.20, with \(w_s, h_s,\) and \(d_s\) equal to the width, height, and depth of the image array whose level is \(d_1\).

For level \(d_2\) the coordinate \((u', v', w')\) is computed as in equation 3.20, with \(w_s, h_s,\) and \(d_s\) equal to the width, height, and depth of the image array whose level is \(d_2\).

The final texture value is then found as

\[
\tau = [1 - \frac{\lambda}{\tau}]\tau_1 + \frac{\lambda}{\tau}\tau_2.
\]

Manual Mipmap Generation

Mipmaps can be generated manually with the command

\[
\text{void GenerateMipmap( enum target );}
\]

where \textit{target} is one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP, or TEXTURE_CUBE_MAP_ARRAY. Mipmap generation affects the texture image attached to \textit{target}. For cube map and cube map array textures, an INVALID_OPERATION error is generated if the texture bound to \textit{target} is not cube complete or cube array complete, respectively, as defined in section 3.9.14.

Mipmap generation replaces texel array levels \(\text{level}_\text{base} + 1\) through \(q\) with arrays derived from the \(\text{level}_\text{base}\) array, regardless of their previous contents. All other mipmap arrays, including the \(\text{level}_\text{base}\) array, are left unchanged by this computation.
3.9. TEXTURING

The internal formats of the derived mipmap arrays all match those of the level_base array, and the dimensions of the derived arrays follow the requirements described in section 3.9.14.

The contents of the derived arrays are computed by repeated, filtered reduction of the level_base array. For one- and two-dimensional array and cube map array textures, each layer is filtered independently. No particular filter algorithm is required, though a box filter is recommended as the default filter.

3.9.12 Texture Magnification

When \( \lambda \) indicates magnification, the value assigned to TEXTURE_MAG_FILTER determines how the texture value is obtained. There are two possible values for TEXTURE_MAG_FILTER: NEAREST and LINEAR. NEAREST behaves exactly as NEAREST for TEXTURE_MIN_FILTER and LINEAR behaves exactly as LINEAR for TEXTURE_MIN_FILTER as described in section 3.9.11, including the texture coordinate wrap modes specified in table 3.18. The level-of-detail level_base texel array is always used for magnification.

Implementations may either unconditionally assume \( c = 0 \) for the minification vs. magnification switch-over point, or may choose to make \( c \) depend on the combination of minification and magnification modes as follows: if the magnification filter is given by LINEAR and the minification filter is given by NEAREST_MIPMAP_NEAREST or NEAREST_MIPMAP_LINEAR, then \( c = 0.5 \). This is done to ensure that a minified texture does not appear “sharper” than a magnified texture. Otherwise \( c = 0 \).

3.9.13 Combined Depth/Stencil Textures

If the texture image has a base internal format of DEPTH_STENCIL, then the stencil index texture component is ignored. The texture value \( \tau \) does not include a stencil index component, but includes only the depth component.

3.9.14 Texture Completeness

A texture is said to be complete if all the image arrays and texture parameters required to utilize the texture for texture application are consistently defined. The definition of completeness varies depending on texture dimensionality and type.

For one-, two-, and three-dimensional and one-and two-dimensional array textures, a texture is mipmap complete if all of the following conditions hold true:

- The set of mipmap arrays level_base through \( q \) (where \( q \) is defined in the Mipmapping discussion of section 3.9.11) were each specified with the...
3.9. TEXTURING

same internal format.

- The dimensions of the arrays follow the sequence described in the Mipmap-
  ping discussion of section 3.9.11.

- \( \text{level}_{\text{base}} \leq \text{level}_{\text{max}} \)

Array levels \( k \) where \( k < \text{level}_{\text{base}} \) or \( k > q \) are insignificant to the definition of completeness.

A cube map texture is mipmap complete if each of the six texture images, considered individually, is mipmap complete. Additionally, a cube map texture is cube complete if the following conditions all hold true:

- The \( \text{level}_{\text{base}} \) arrays of each of the six texture images making up the cube map have identical, positive, and square dimensions.

- The \( \text{level}_{\text{base}} \) arrays were each specified with the same internal format.

A cube map array texture is cube array complete if it is complete when treated as a two-dimensional array and cube complete for every cube map slice within the array texture.

Using the preceding definitions, a texture is complete unless any of the following conditions hold true:

- Any dimension of the \( \text{level}_{\text{base}} \) array is not positive. For a rectangular or multisample texture, \( \text{level}_{\text{base}} \) is always zero.

- The texture is a cube map texture, and is not cube complete.

- The texture is a cube map array texture, and is not cube array complete.

- The minification filter requires a mipmap (is neither \text{NEAREST} nor \text{LINEAR}), and the texture is not mipmap complete.

- The internal format of the texture arrays is integer (see table \( 3.12 \)), and either the magnification filter is not \text{NEAREST}, or the minification filter is neither \text{NEAREST} nor \text{NEAREST_MIPMAP_NEAREST}.

Effects of Sampler Objects on Texture Completeness

If a sampler object and a texture object are simultaneously bound to the same texture unit, then the sampling state for that unit is taken from the sampler object (see section 3.9.2). This can have an effect on the effective completeness of the texture.
In particular, if the texture is not mipmap complete and the sampler object specifies a TEXTURE_MIN_FILTER requiring mipmaps, the texture will be considered incomplete for the purposes of that texture unit. However, if the sampler object does not require mipmaps, the texture object will be considered complete. This means that a texture can be considered both complete and incomplete simultaneously if it is bound to two or more texture units along with sampler objects with different states.

Effects of Completeness on Texture Application

Texture lookup and texture fetch operations performed in shaders are affected by completeness of the texture being sampled as described in sections 2.11.12 and 3.10.2.

Effects of Completeness on Texture Image Specification

The implementation-dependent maximum sizes for texture image arrays depend on the texture level. In particular, an implementation may allow a texture image array of level 1 or greater to be created only if a mipmap complete set of image arrays consistent with the requested array can be supported where the values of TEXTURE_BASE_LEVEL and TEXTURE_MAX_LEVEL are 0 and 1000 respectively. As a result, implementations may permit a texture image array at level zero that will never be mipmap complete and can only be used with non-mipmapped minification filters.

3.9.15 Texture State and Proxy State

The state necessary for texture can be divided into two categories. First, there are the multiple sets of texel arrays (a single array for the rectangular texture target; one set of mipmap arrays each for the one-, two-, and three-dimensional and one- and two-dimensional array texture targets; and six sets of mipmap arrays for the cube map texture targets) and their number. Each array has associated with it a width, height (two- and three-dimensional, rectangular, one-dimensional array, cube map, and cube map array only), and depth (three-dimensional, two-dimensional array, and cube map array only), an integer describing the internal format of the image, integer values describing the resolutions of each of the red, green, blue, alpha, depth, and stencil components of the image, integer values describing the type (unsigned normalized, integer, floating-point, etc.) of each of the components, a boolean describing whether the image is compressed or not, an integer size of a compressed image, and an integer containing the name of a buffer object bound as the data store of the image.
3.9. TEXTURING

Each initial texel array is null (zero width, height, and depth, internal format RGBA, component sizes set to zero and component types set to NONE, the compressed flag set to FALSE, a zero compressed size, and the bound buffer object name is zero. Multisample textures contain an integer identifying the number of samples in each texel, and a boolean indicating whether identical sample locations and the same number of samples will be used for all texels in the image. The buffer texture target has associated an integer containing the name of the buffer object that provided the data store for the texture, initially zero.

Next, there are the four sets of texture properties, corresponding to the one-, two-, three-dimensional, and cube map texture targets. Each set consists of the selected minification and magnification filters, the wrap modes for s, t (two- and three-dimensional and cube map only), and r (three-dimensional only), the TEXTURE_BORDER_COLOR, two floating-point numbers describing the minimum and maximum level of detail, two integers describing the base and maximum mipmap array, a boolean flag indicating whether the format and dimensions of the texture are immutable, three integers describing the depth texture mode, compare mode, and compare function, and four integers describing the red, green, blue, and alpha swizzle modes (see section 3.10.2). In the initial state, the value assigned to TEXTURE_MIN_FILTER is NEAREST_MIPMAP_LINEAR (except for rectangular textures, where the initial value is LINEAR), and the value for TEXTURE_MAG_FILTER is LINEAR. s, t, and r wrap modes are all set to REPEAT (except for rectangular textures, where the initial value is CLAMP_TO_EDGE). The values of TEXTURE_MIN_LOD and TEXTURE_MAX_LOD are -1000 and 1000 respectively. The values of TEXTURE_BASE_LEVEL and TEXTURE_MAX_LEVEL are 0 and 1000 respectively. The value of TEXTURE_BORDER_COLOR is (0,0,0,0). The value of TEXTURE_IMMUTABLE_FORMAT is FALSE. The values of TEXTURE_COMPARE_MODE, and TEXTURE_COMPARE_FUNC are NONE, and LEQUAL respectively. The values of TEXTURE_SWIZZLE_R, TEXTURE_SWIZZLE_G, TEXTURE_SWIZZLE_B, and TEXTURE_SWIZZLE_A are RED, GREEN, BLUE, and ALPHA, respectively.

In addition to image arrays for the non-proxy texture targets described above, partially instantiated image arrays are maintained for one-, two-, and three-dimensional, rectangular, one- and two-dimensional array, and cube map array textures. Additionally, a single proxy image array is maintained for the cube map texture. Each proxy image array includes width, height, depth, and internal format state values, as well as state for the red, green, blue, alpha, depth, and stencil component resolutions and types. Proxy arrays do not include image data nor texture parameters. When TexImage3D is executed with target specified as PROXY_TEXTURE_3D, the three-dimensional proxy state values of the specified level-of-detail are recomputed and updated. If the image array would not be supported by TexImage3D called with target set to TEXTURE_3D, no error is generated, but the
proxy width, height, depth, and component resolutions are set to zero, and the component types are set to NONE. If the image array would be supported by such a call to TexImage3D, the proxy state values are set exactly as though the actual image array were being specified. No pixel data are transferred or processed in either case.

Proxy arrays for one-and two-dimensional textures, one-and two-dimensional array textures, and cube map array textures are operated on in the same way when TexImage1D is executed with target specified as PROXY_TEXTURE_1D, TexImage2D is executed with target specified as PROXY_TEXTURE_2D, PROXY_TEXTURE_1D_ARRAY, or PROXY_TEXTURE_RECTANGLE, or TexImage3D is executed with target specified as PROXY_TEXTURE_2D_ARRAY or PROXY_TEXTURE_CUBE_MAP_ARRAY.

Proxy arrays for two-dimensional multisample and two-dimensional multisample array textures are operated on in the same way when TexImage2DMultisample is called with target specified as PROXY_TEXTURE_2D_MULTISAMPLE, or TexImage3DMultisample is called with target specified as PROXY_TEXTURE_2D_MULTISAMPLE_ARRAY.

The cube map proxy arrays are operated on in the same manner when TexImage2D is executed with the target field specified as PROXY_TEXTURE_CUBE_MAP, with the addition that determining that a given cube map texture is supported with PROXY_TEXTURE_CUBE_MAP indicates that all six of the cube map 2D images are supported. Likewise, if the specified PROXY_TEXTURE_CUBE_MAP is not supported, none of the six cube map 2D images are supported.

There is no image or non-level-related state associated with proxy textures. Therefore they may not be used as textures, and calling BindTexture, GetTexImage, GetTexParameteriv, or GetTexParameterfv with a proxy texture target generates an INVALID_ENUM error.

### 3.9.16 Immutable-Format Texture Images

An alternative set of commands is provided for specifying the properties of all levels of a texture at once. Once a texture is specified with such a command, the format and dimensions of all levels becomes immutable, unless it is a proxy texture (since otherwise it would no longer be possible to use the proxy). The contents of the images and the parameters can still be modified. Such a texture is referred to as an immutable-format texture. The immutability status of a texture can be determined by calling GetTextureParameter with pname TEXTURE_IMMUTABLE_FORMAT.

Each of the commands below is described by pseudocode which indicates the effect on the dimensions and format of the texture. For all of the commands, the
3.9. TEXTURING

following apply in addition to the pseudocode:

- If the default texture object is bound to target, an INVALID_OPERATION error is generated.

- If executing the pseudo-code would result in a OUT_OF_MEMORY error, the error is generated and the results of executing the command are undefined.

- If executing the pseudocode would result in any other error, the error is generated and the command will have no effect.

- Any existing levels that are not replaced are reset to their initial state.

- If width, height, depth or levels are less than 1, an INVALID_VALUE error is generated.

- The pixel unpack buffer should be considered to be zero; i.e., the image contents are unspecified.

- Since no pixel data are provided, the format and type values used in the pseudocode are irrelevant; they can be considered to be any values that are legal to use with internalformat.

- If the command is successful, TEXTURE_IMMUTABLE_FORMAT becomes TRUE.

- If internalformat is one of the unsized base internal formats listed in table 3.11, an INVALID_ENUM error is generated.

The command

```c
void TexStorage1D( enum target, sizei levels,
                   enum internalformat, sizei width);
```

specifies all the levels of a one-dimensional texture (or proxy) at the same time. It is described by the pseudocode below:

```c
for (i = 0; i < levels; i++) {
    TexImage1D(target, i, internalformat, width, 0,
               format, type, NULL);
    width = max(1, ⌊width 2⌋);
}
```
If `target` is not `TEXTURE_1D` or `PROXY_TEXTURE_1D` then an `INVALID_ENUM` error is generated. If `levels` is greater than \(\left\lfloor \log_2(\text{width}) \right\rfloor + 1\) then an `INVALID_OPERATION` error is generated.

The command

```c
void TexStorage2D( enum target, sizei levels,
                     enum internalformat, sizei width, sizei height);
```

specifies all the levels of a two-dimensional, cube-map, one-dimension array or rectangle texture (or proxy) at the same time. The pseudocode depends on the `target`:

- `TEXTURE_2D`, `PROXY_TEXTURE_2D`, `TEXTURE_RECTANGLE`, `PROXY_TEXTURE_RECTANGLE`, or `PROXY_TEXTURE_CUBE_MAP`:

  ```c
  for (i = 0; i < levels; i++) {
   TexImage2D(target, i, internalformat, width, height, 0,
                format, type, NULL);
    width = max(1, \left\lfloor \frac{width}{2} \right\rfloor);
    height = max(1, \left\lfloor \frac{height}{2} \right\rfloor);
  }
  ```

- `TEXTURE_CUBE_MAP`:

  ```c
  for (i = 0; i < levels; i++) {
    for face in (+X, -X, +Y, -Y, +Z, -Z) {
     TexImage2D(face, i, internalformat, width, height, 0,
                 format, type, NULL);
    }
    width = max(1, \left\lfloor \frac{width}{2} \right\rfloor);
    height = max(1, \left\lfloor \frac{height}{2} \right\rfloor);
  }
  ```

- `TEXTURE_1D_ARRAY` or `PROXY_TEXTURE_1D_ARRAY`:

  ```c
  for (i = 0; i < levels; i++) {
   TexImage2D(target, i, internalformat, width, height, 0,
               format, type, NULL);
    width = max(1, \left\lfloor \frac{width}{2} \right\rfloor);
  }
  ```
If `target` is not one of those listed above, an `INVALID_ENUM` error is generated. An `INVALID_OPERATION` error is generated if any of the following conditions hold:

- `target` is `TEXTURE_1D_ARRAY` or `PROXY_TEXTURE_1D_ARRAY` and `levels` is greater than \(\lceil \log_2(\text{width}) \rceil + 1\)
- `target` is not `TEXTURE_1D_ARRAY` or `PROXY_TEXTURE_1D_ARRAY` and `levels` is greater than \(\lceil \log_2(\max(\text{width}, \text{height})) \rceil + 1\)

The command

```c
void TexStorage3D( enum target, sizei levels,
                    enum internalformat, sizei width, sizei height,
                    sizei depth);
```

specifies all the levels of a three-dimensional, two-dimensional array texture, or cube-map array texture (or proxy). The pseudocode depends on `target`:

**TEXTURE_3D or PROXY_TEXTURE_3D:**

```c
for (i = 0; i < levels; i++) {
   TexImage3D(target, i, internalformat, width, height, depth, 0,
               format, type, NULL);
    width = max(1, \lfloor \frac{\text{width}}{2} \rfloor);
    height = max(1, \lfloor \frac{\text{height}}{2} \rfloor);
    depth = max(1, \lfloor \frac{\text{depth}}{2} \rfloor);
}
```

**TEXTURE_2D_ARRAY, PROXY_TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP_ARRAY or PROXY_TEXTURE_CUBE_MAP_ARRAY:**

```c
for (i = 0; i < levels; i++) {
   TexImage3D(target, i, internalformat, width, height, depth, 0,
               format, type, NULL);
    width = max(1, \lfloor \frac{\text{width}}{2} \rfloor);
    height = max(1, \lfloor \frac{\text{height}}{2} \rfloor);
}
```

If `target` is not one of those listed above, an `INVALID_ENUM` error is generated. An `INVALID_OPERATION` error is generated if any of the following conditions hold:
3.9. TEXTURING

- **target** is TEXTURE_3D or PROXY_TEXTURE_3D and **levels** is greater than \(\lceil \log_2(\max(width, height, depth)) \rceil + 1 \)

- **target** is TEXTURE_2D_ARRAY, PROXY_TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP_ARRAY or PROXY_TEXTURE_CUBE_MAP_ARRAY and **levels** is greater than \(\lceil \log_2(\max(width, height)) \rceil + 1 \)

After a successful call to any TexStorage* command with a non-proxy target, the value of TEXTURE_IMMUTABLE_FORMAT for this texture object is set to TRUE, and no further changes to the dimensions or format of the texture object may be made. Other commands may only alter the texel values and texture parameters. Using any of the following commands with the same texture will result in an INVALID_OPERATION error being generated, even if it does not affect the dimensions or format:

- TexImage*
- CompressedTexImage*
- CopyTexImage*
- TexStorage*

3.9.17 Texture Comparison Modes

Texture values can also be computed according to a specified comparison function. Texture parameter TEXTURE_COMPARE_MODE specifies the comparison operands, and parameter TEXTURE_COMPARE_FUNC specifies the comparison function.

### Depth Texture Comparison Mode

If the currently bound texture’s base internal format is DEPTH_COMPONENT or DEPTH_STENCIL, then TEXTURE_COMPARE_MODE and TEXTURE_COMPARE_FUNC control the output of the texture unit as described below. Otherwise, the texture unit operates in the normal manner and texture comparison is bypassed.

Let \(D_t\) be the depth texture value and \(D_{ref}\) be the reference value, provided by the shader’s texture lookup function.

If the texture’s internal format indicates a fixed-point depth texture, then \(D_t\) and \(D_{ref}\) are clamped to the range \([0, 1]\); otherwise no clamping is performed.

Then the effective texture value is computed as follows:

If the value of TEXTURE_COMPARE_MODE is NONE, then
If the value of TEXTURE_COMPARE_MODE is COMPARE_REF_TO_TEXTURE, then $r$ depends on the texture comparison function as shown in table 3.19.

<table>
<thead>
<tr>
<th>Texture Comparison Function</th>
<th>Computed result $r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEQUAL</td>
<td>$r = \begin{cases} 1.0, &amp; D_{ref} \leq D_t \ 0.0, &amp; D_{ref} &gt; D_t \end{cases}$</td>
</tr>
<tr>
<td>GEQUAL</td>
<td>$r = \begin{cases} 1.0, &amp; D_{ref} \geq D_t \ 0.0, &amp; D_{ref} &lt; D_t \end{cases}$</td>
</tr>
<tr>
<td>LESS</td>
<td>$r = \begin{cases} 1.0, &amp; D_{ref} &lt; D_t \ 0.0, &amp; D_{ref} \geq D_t \end{cases}$</td>
</tr>
<tr>
<td>GREATER</td>
<td>$r = \begin{cases} 1.0, &amp; D_{ref} &gt; D_t \ 0.0, &amp; D_{ref} \leq D_t \end{cases}$</td>
</tr>
<tr>
<td>EQUAL</td>
<td>$r = \begin{cases} 1.0, &amp; D_{ref} = D_t \ 0.0, &amp; D_{ref} \neq D_t \end{cases}$</td>
</tr>
<tr>
<td>NOTEQUAL</td>
<td>$r = \begin{cases} 1.0, &amp; D_{ref} \neq D_t \ 0.0, &amp; D_{ref} = D_t \end{cases}$</td>
</tr>
<tr>
<td>ALWAYS</td>
<td>$r = 1.0$</td>
</tr>
<tr>
<td>NEVER</td>
<td>$r = 0.0$</td>
</tr>
</tbody>
</table>

Table 3.19: Depth texture comparison functions.

The resulting $r$ is assigned to $R_t$. If the value of TEXTURE_MAG_FILTER is not NEAREST, or the value of TEXTURE_MIN_FILTER is not NEAREST or NEAREST_MIPMAP_NEAREST, then $r$ may be computed by comparing more than one depth texture value to the texture reference value. The details of this are implementation-dependent, but $r$ should be a value in the range $[0, 1]$ which is proportional to the number of comparison passes or failures.

3.9.18 sRGB Texture Color Conversion

If the currently bound texture’s internal format is one of SRGB, SRGB8, SRGB_ALPHA, SRGB8_ALPHA8, COMPRESSED_SRGB, or COMPRESSED_SRGB_ALPHA, the red, green, and blue components are converted from an sRGB color space to a...
linear color space as part of filtering described in sections 3.9.11 and 3.9.12. Any
alpha component is left unchanged. Ideally, implementations should perform this
color conversion on each sample prior to filtering but implementations are allowed
to perform this conversion after filtering (though this post-filtering approach is in-
favor to converting from sRGB prior to filtering).

The conversion from an sRGB encoded component, $c_s$, to a linear component,$c_l$, is as follows.

$$c_l = \begin{cases} 
\frac{c_s}{1.0938}, & c_s \leq 0.04045 \\
(c_s + 0.055) / 1.055, & c_s > 0.04045
\end{cases}$$  \hspace{1cm} (3.27)

Assume $c_s$ is the sRGB component in the range $[0, 1]$.

### 3.9.19 Shared Exponent Texture Color Conversion

If the currently bound texture’s internal format is RGB9_E5, the red, green, blue, and shared bits are converted to color components (prior to filtering) using shared exponent decoding. The component $red_s$, $green_s$, $blue_s$, and $exp_{shared}$ values (see section 3.9.3) are treated as unsigned integers and are converted to $red$, $green$, and $blue$ as follows:

$$red = red_s 2^{exp_{shared} - B}$$
$$green = green_s 2^{exp_{shared} - B}$$
$$blue = blue_s 2^{exp_{shared} - B}$$

### 3.9.20 Texture Image Loads and Stores

The contents of a texture may be made available for shaders to read and write by
binding the texture to one of a collection of image units. The GL implementa-
tion provides an array of image units numbered beginning with zero, with the total
number of image units provided given by the implementation-dependent value of
$\text{MAX\_IMAGE\_UNITS}$. Unlike texture image units, image units do not have a sepa-
rate attachment for each texture target texture; each image unit may have only one
texture bound at a time.

A texture may be bound to an image unit for use by image loads and stores
with the command

```c
void BindImageTexture( uint unit, uint texture, int level,
    boolean layered, int layer, enum access, enum format );
```

OpenGL 4.2 (Core Profile) - April 27, 2012
where *unit* identifies the image unit, *texture* is the name of the texture, and *level* selects a single level of the texture. If *texture* is zero, any texture currently bound to image unit *unit* is unbound. If *unit* is greater than or equal to the value of `MAX_IMAGE_UNITS`, if *level* or *layer* is less than zero, or if *texture* is not the name of an existing texture object, the error `INVALID_VALUE` is generated.

If the texture identified by *texture* is a one-dimensional array, two-dimensional array, three-dimensional, cube map, cube map array, or two-dimensional multisample array texture, it is possible to bind either the entire texture level or a single layer or face of the texture level. If *layered* is `TRUE`, the entire level is bound. If *layered* is `FALSE`, only the single layer identified by *layer* will be bound. When *layered* is `FALSE`, the single bound layer is treated as a different texture target for image accesses:

- one-dimensional array texture layers are treated as one-dimensional textures;
- two-dimensional array, three-dimensional, cube map, cube map array texture layers are treated as two-dimensional textures; and
- two-dimensional multisample array textures are treated as two-dimensional multisample textures.

For cube map textures where *layered* is `FALSE`, the face is taken by mapping the layer number to a face according to table 4.12. For cube map array textures where *layered* is `FALSE`, the selected layer number is mapped to a texture layer and cube face using the following equations and mapping *face* to a face according to table 4.12.

\[
\text{layer} = \left\lfloor \frac{\text{layer}_\text{orig}}{6} \right\rfloor \\
\text{face} = \text{layer}_\text{orig} - (\text{layer} \times 6)
\]

If the texture identified by *texture* does not have multiple layers or faces, the entire texture level is bound, regardless of the values specified for *layered* and *layer*.

*format* specifies the format that the elements of the image will be treated as when doing formatted stores, as described later in this section. This is referred to as the *image unit format*. This must be one of the formats listed in table 3.21; otherwise, the error `INVALID_VALUE` is generated.

*access* specifies whether the texture bound to the image will be treated as `READ_ONLY`, `WRITE_ONLY`, or `READ_WRITE`. If a shader reads from an image unit with a texture bound as `WRITE_ONLY`, or writes to an image unit with a texture
### 3.9. TEXTURING

<table>
<thead>
<tr>
<th>Texture target</th>
<th>i</th>
<th>j</th>
<th>k</th>
<th>layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXTURE_1D</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TEXTURE_2D</td>
<td>x</td>
<td>y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TEXTURE_3D</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>-</td>
</tr>
<tr>
<td>TEXTURE_RECTANGLE</td>
<td>x</td>
<td>y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TEXTURE_CUBE_MAP</td>
<td>x</td>
<td>y</td>
<td>-</td>
<td>z</td>
</tr>
<tr>
<td>TEXTURE_BUFFER</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TEXTURE_1D_ARRAY</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TEXTURE_2D_ARRAY</td>
<td>x</td>
<td>y</td>
<td>-</td>
<td>z</td>
</tr>
<tr>
<td>TEXTURE_CUBE_MAP_ARRAY</td>
<td>x</td>
<td>y</td>
<td>-</td>
<td>z</td>
</tr>
<tr>
<td>TEXTURE_2D_MULTISAMPLE</td>
<td>x</td>
<td>y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TEXTURE_2D_MULTISAMPLE_ARRAY</td>
<td>x</td>
<td>y</td>
<td>-</td>
<td>z</td>
</tr>
</tbody>
</table>

Table 3.20: Mapping of image load, store, and atomic texel coordinate components to texel numbers.

If a texture object bound to one or more image units is deleted by `DeleteTextures`, it is detached from each such image unit, as though `BindImageTexture` were called with `unit` identifying the image unit and `texture` set to zero.

When a shader accesses the texture bound to an image unit using a built-in image load, store, or atomic function, it identifies a single texel by providing a one-, two-, or three-dimensional coordinate. Multisample texture accesses also specify a sample number. A coordinate vector is mapped to an individual texel $\tau_i$, $\tau_{ij}$, or $\tau_{ijk}$ according to the target of the texture bound to the image unit using table 3.20. As noted above, single-layer bindings of array or cube map textures are considered to use a texture target corresponding to the bound layer, rather than that of the full texture.

If the texture target has layers or cube map faces, the layer or face number is taken from the `layer` argument of `BindImageTexture` if the texture is bound with `layered` set to `FALSE`, or from the coordinate identified by table 3.20 otherwise. For cube map and cube map array textures with `layered` set to `TRUE`, the coordinate is mapped to a layer and face in the same manner as the `layer` argument of `BindImageTexture`.

If the individual texel identified for an image load, store, or atomic operation doesn’t exist, the access is treated as invalid. Invalid image loads will return zero.
3.9. TEXTURING

Invalid image stores will have no effect. Invalid image atomics will not update any texture bound to the image unit and will return zero. An access is considered invalid if:

- no texture is bound to the selected image unit;
- the texture bound to the selected image unit is incomplete;
- the texture level bound to the image unit is less than the base level or greater than the maximum level of the texture;
- the internal format of the texture bound to the image unit is not found in table 3.21;
- the internal format of the texture bound to the image unit is incompatible with the specified format according to table 3.22;
- the texture bound to the image unit has layers, and the selected layer or cube map face doesn’t exist;
- the selected texel $\tau_i$, $\tau_{ij}$, or $\tau_{ijk}$ doesn’t exist;
- the image has more samples than the implementation-dependent value of MAX_IMAGE_SAMPLES.

Additionally, there are a number of cases where image load, store, or atomic operations are considered to involve a format mismatch. In such cases, undefined values will be returned by image loads and atomic operations and undefined values will be written by stores and atomic operations. A format mismatch will occur if:

- the type of image variable used to access the image unit does not match the target of a texture bound to the image unit with layered set to TRUE;
- the type of image variable used to access the image unit does not match the target corresponding to a single layer of a multi-layer texture target bound to the image unit with layered set to FALSE;
- the type of image variable used to access the image unit has a component data type (floating-point, signed integer, unsigned integer) incompatible with the format of the image unit;
- the format layout qualifier for an image variable used for an image load or atomic operation does not match the format of the image unit, according to table 3.21; or
• the image variable used for an image store has a format layout qualifier, and that qualifier does not match the format of the image unit, according to table 3.21.

For textures with multiple samples per texel, the sample selected for an image load, store, or atomic is undefined if the sample coordinate is negative or greater than or equal to the number of samples in the texture.

If a shader performs an image load, store, or atomic operation using an image variable declared as an array, and if the index used to select an individual element is negative or greater than or equal to the size of the array, the results of the operation are undefined but may not lead to termination.

Accesses to textures bound to image units do format conversions based on the format argument specified when the image is bound. Loads always return a value as a vec4, ivec4, or uvec4, and stores always take the source data as a vec4, ivec4, or uvec4. Data are converted to/from the specified format according to the process described for a TexImage2D or GetTexImage command with format and type as RGBA and FLOAT for vec4 data, as RGBA_INTEGER and INT for ivec4 data, or as RGBA_INTEGER and UNSIGNED_INT for uvec4 data, respectively. Unused components are filled in with (0, 0, 0, 1) (where 0 and 1 are either floating-point or integer values, depending on the format).

Any image variable used for shader loads or atomic memory operations must be declared with a format layout qualifier matching the format of its associated image unit, as enumerated in table 3.21. Otherwise, the access is considered to involve a format mismatch, as described above. Image variables used exclusively for image stores need not include a format layout qualifier, but any declared qualifier must match the image unit format to avoid a format mismatch.

<table>
<thead>
<tr>
<th>Image Unit Format</th>
<th>Format Qualifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGBA32F</td>
<td>rgba32f</td>
</tr>
<tr>
<td>RGBA16F</td>
<td>rgba16f</td>
</tr>
<tr>
<td>RG32F</td>
<td>rg32f</td>
</tr>
<tr>
<td>RG16F</td>
<td>rg16f</td>
</tr>
<tr>
<td>R11F_G11F_B10F</td>
<td>r11f_g11f_b10f</td>
</tr>
<tr>
<td>R32F</td>
<td>r32f</td>
</tr>
<tr>
<td>R16F</td>
<td>r16f</td>
</tr>
<tr>
<td>RGBA32UI</td>
<td>rgba32ui</td>
</tr>
<tr>
<td>RGBA16UI</td>
<td>rgba16ui</td>
</tr>
<tr>
<td>RGB10_A2UI</td>
<td>rgb10_a2ui</td>
</tr>
</tbody>
</table>

(Continued on next page)
### Supported image unit formats (continued)

<table>
<thead>
<tr>
<th>Image Unit Format</th>
<th>Format Qualifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGBA8UI</td>
<td>rgba8ui</td>
</tr>
<tr>
<td>RG32UI</td>
<td>rg32ui</td>
</tr>
<tr>
<td>RG16UI</td>
<td>rg16ui</td>
</tr>
<tr>
<td>RGB8UI</td>
<td>rg8ui</td>
</tr>
<tr>
<td>R32UI</td>
<td>r32ui</td>
</tr>
<tr>
<td>R16UI</td>
<td>r16ui</td>
</tr>
<tr>
<td>R8UI</td>
<td>r8ui</td>
</tr>
<tr>
<td>RGBA32I</td>
<td>rgba32i</td>
</tr>
<tr>
<td>RGBA16I</td>
<td>rgba16i</td>
</tr>
<tr>
<td>RGBA8I</td>
<td>rgba8i</td>
</tr>
<tr>
<td>RG32I</td>
<td>rg32i</td>
</tr>
<tr>
<td>RG16I</td>
<td>rg16i</td>
</tr>
<tr>
<td>RG8I</td>
<td>rg8i</td>
</tr>
<tr>
<td>R32I</td>
<td>r32i</td>
</tr>
<tr>
<td>R16I</td>
<td>r16i</td>
</tr>
<tr>
<td>R8I</td>
<td>r8i</td>
</tr>
<tr>
<td>RGBA16</td>
<td>rgba16</td>
</tr>
<tr>
<td>RGB10_A2</td>
<td>rgb10_a2</td>
</tr>
<tr>
<td>RGBA8</td>
<td>rgba8</td>
</tr>
<tr>
<td>RG16</td>
<td>rg16</td>
</tr>
<tr>
<td>RG8</td>
<td>rg8</td>
</tr>
<tr>
<td>R16</td>
<td>r16</td>
</tr>
<tr>
<td>R8</td>
<td>r8</td>
</tr>
<tr>
<td>RGBA16_SNORM</td>
<td>rgba16_snorm</td>
</tr>
<tr>
<td>RGBA8_SNORM</td>
<td>rgba8_snorm</td>
</tr>
<tr>
<td>RG16_SNORM</td>
<td>rg16_snorm</td>
</tr>
<tr>
<td>RG8_SNORM</td>
<td>rg8_snorm</td>
</tr>
<tr>
<td>R16_SNORM</td>
<td>r16_snorm</td>
</tr>
<tr>
<td>R8_SNORM</td>
<td>r8_snorm</td>
</tr>
</tbody>
</table>

Table 3.21: Supported image unit formats, with equivalent format layout qualifiers.

When a texture is bound to an image unit, the *format* parameter for the image unit need not exactly match the texture internal format as long as the formats are...
considered compatible. A pair of formats is considered to match in size if the corresponding entries in the Size column of table 3.22 are identical. A pair of formats is considered to match by class if the corresponding entries in the Class column of table 3.22 are identical. For textures allocated by the GL, an image unit format is compatible with a texture internal format if they match by size. For textures allocated outside the GL, format compatibility is determined by matching by size or by class, in an implementation dependent manner. The matching criterion used for a given texture may be determined by calling GetTexParameter with value set to IMAGE_FORMAT_COMPATIBILITY_TYPE, with return values of IMAGE_FORMAT_COMPATIBILITY_BY_SIZE and IMAGE_FORMAT_COMPATIBILITY_BY_CLASS, specifying matches by size and class, respectively.

When the format associated with an image unit does not exactly match the internal format of the texture bound to the image unit, image loads, stores, and atomic operations re-interpret the memory holding the components of an accessed texel according to the format of the image unit. The re-interpretation for image loads and the read portion of image atomics is performed as though data were copied from the texel of the bound texture to a similar texel represented in the format of the image unit. Similarly, the re-interpretation for image stores and the write portion of image atomics is performed as though data were copied from a texel represented in the format of the image unit to the texel in the bound texture. In both cases, this copy operation would be performed by:

- reading the texel from the source format to scratch memory according to the process described for GetTexImage (see section 6.1.4), using default pixel storage modes and format and type parameters corresponding to the source format in table 3.22; and
- writing the texel from scratch memory to the destination format according to the process described for TexSubImage3D (see section 3.9.4), using default pixel storage modes and format and type parameters corresponding to the destination format in table 3.22.

<table>
<thead>
<tr>
<th>Image Format</th>
<th>Size</th>
<th>Class</th>
<th>Pixel format</th>
<th>Pixel type</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGBA32F</td>
<td>128</td>
<td>4x32</td>
<td>RGBA</td>
<td>FLOAT</td>
</tr>
<tr>
<td>RGBA16F</td>
<td>64</td>
<td>4x16</td>
<td>RGBA</td>
<td>HALF_FLOAT</td>
</tr>
<tr>
<td>RG32F</td>
<td>64</td>
<td>2x32</td>
<td>RG</td>
<td>FLOAT</td>
</tr>
<tr>
<td>RG16F</td>
<td>32</td>
<td>2x16</td>
<td>RG</td>
<td>HALF_FLOAT</td>
</tr>
<tr>
<td>R11F_G11F_B10F</td>
<td>32</td>
<td>(a)</td>
<td>RGB</td>
<td>UNSIGNED_INT_10F_11F_11F_REV</td>
</tr>
</tbody>
</table>

(Continued on next page)
### Table 3.22: Texel sizes, compatibility classes, and pixel format/type combinations for each image format.

<table>
<thead>
<tr>
<th>Image Format</th>
<th>Size</th>
<th>Class</th>
<th>Pixel format</th>
<th>Pixel type</th>
</tr>
</thead>
<tbody>
<tr>
<td>R32F</td>
<td>32</td>
<td>1x32</td>
<td>RED</td>
<td>FLOAT</td>
</tr>
<tr>
<td>R16F</td>
<td>16</td>
<td>1x16</td>
<td>RED</td>
<td>HALF_FLOAT</td>
</tr>
<tr>
<td>RGBA32UI</td>
<td>128</td>
<td>4x32</td>
<td>RGBA_INTEGER</td>
<td>UNSIGNED_INT</td>
</tr>
<tr>
<td>RGBA16UI</td>
<td>64</td>
<td>4x16</td>
<td>RGBA_INTEGER</td>
<td>UNSIGNED_SHORT</td>
</tr>
<tr>
<td>RGB10_A2UI</td>
<td>32</td>
<td>(b)</td>
<td>RGBA_INTEGER</td>
<td>UNSIGNED_INT_2_10_10_10_REV</td>
</tr>
<tr>
<td>RGBA8UI</td>
<td>32</td>
<td>4x8</td>
<td>RGBA_INTEGER</td>
<td>UNSIGNED_BYTE</td>
</tr>
<tr>
<td>RG32UI</td>
<td>64</td>
<td>2x32</td>
<td>RG_INTEGER</td>
<td>UNSIGNED_INT</td>
</tr>
<tr>
<td>RG16UI</td>
<td>32</td>
<td>2x16</td>
<td>RG_INTEGER</td>
<td>UNSIGNED_SHORT</td>
</tr>
<tr>
<td>RG8UI</td>
<td>16</td>
<td>2x8</td>
<td>RG_INTEGER</td>
<td>UNSIGNED_BYTE</td>
</tr>
<tr>
<td>R32UI</td>
<td>32</td>
<td>1x32</td>
<td>RED_INTEGER</td>
<td>UNSIGNED_INT</td>
</tr>
<tr>
<td>R16UI</td>
<td>16</td>
<td>1x16</td>
<td>RED_INTEGER</td>
<td>UNSIGNED_SHORT</td>
</tr>
<tr>
<td>R8UI</td>
<td>8</td>
<td>1x8</td>
<td>RED_INTEGER</td>
<td>UNSIGNED_BYTE</td>
</tr>
<tr>
<td>RGBA32I</td>
<td>128</td>
<td>4x32</td>
<td>RGBA_INTEGER</td>
<td>INT</td>
</tr>
<tr>
<td>RGBA16I</td>
<td>64</td>
<td>4x16</td>
<td>RGBA_INTEGER</td>
<td>SHORT</td>
</tr>
<tr>
<td>RGBA8I</td>
<td>32</td>
<td>4x8</td>
<td>RGBA_INTEGER</td>
<td>BYTE</td>
</tr>
<tr>
<td>RG32I</td>
<td>64</td>
<td>2x32</td>
<td>RG_INTEGER</td>
<td>INT</td>
</tr>
<tr>
<td>RG16I</td>
<td>32</td>
<td>2x16</td>
<td>RG_INTEGER</td>
<td>SHORT</td>
</tr>
<tr>
<td>RG8I</td>
<td>16</td>
<td>2x8</td>
<td>RG_INTEGER</td>
<td>BYTE</td>
</tr>
<tr>
<td>R32I</td>
<td>32</td>
<td>1x32</td>
<td>RED_INTEGER</td>
<td>INT</td>
</tr>
<tr>
<td>R16I</td>
<td>16</td>
<td>1x16</td>
<td>RED_INTEGER</td>
<td>SHORT</td>
</tr>
<tr>
<td>R8I</td>
<td>8</td>
<td>1x8</td>
<td>RED_INTEGER</td>
<td>BYTE</td>
</tr>
<tr>
<td>RGBA16</td>
<td>64</td>
<td>4x16</td>
<td>RGBA</td>
<td>UNSIGNED_SHORT</td>
</tr>
<tr>
<td>RGB10_A2</td>
<td>32</td>
<td>(b)</td>
<td>RGBA</td>
<td>UNSIGNED_INT_2_10_10_10_REV</td>
</tr>
<tr>
<td>RGBA8</td>
<td>32</td>
<td>4x8</td>
<td>RGBA</td>
<td>UNSIGNED_BYTE</td>
</tr>
<tr>
<td>RG16</td>
<td>32</td>
<td>2x16</td>
<td>RG</td>
<td>UNSIGNED_SHORT</td>
</tr>
<tr>
<td>RG8</td>
<td>16</td>
<td>2x8</td>
<td>RG</td>
<td>UNSIGNED_BYTE</td>
</tr>
<tr>
<td>R16</td>
<td>16</td>
<td>1x16</td>
<td>RED</td>
<td>UNSIGNED_SHORT</td>
</tr>
<tr>
<td>R8</td>
<td>8</td>
<td>1x8</td>
<td>RED</td>
<td>UNSIGNED_BYTE</td>
</tr>
<tr>
<td>RGBA16_SNORM</td>
<td>64</td>
<td>4x16</td>
<td>RGBA</td>
<td>SHORT</td>
</tr>
<tr>
<td>RGBA8_SNORM</td>
<td>32</td>
<td>4x8</td>
<td>RGBA</td>
<td>BYTE</td>
</tr>
<tr>
<td>RG16_SNORM</td>
<td>32</td>
<td>2x16</td>
<td>RG</td>
<td>SHORT</td>
</tr>
<tr>
<td>RG8_SNORM</td>
<td>16</td>
<td>2x8</td>
<td>RG</td>
<td>BYTE</td>
</tr>
<tr>
<td>R16_SNORM</td>
<td>16</td>
<td>1x16</td>
<td>RED</td>
<td>SHORT</td>
</tr>
<tr>
<td>R8_SNORM</td>
<td>8</td>
<td>1x8</td>
<td>RED</td>
<td>BYTE</td>
</tr>
</tbody>
</table>

Class (a) is for 11/11/10 packed floating-point formats; class (b) is for 10/10/10/2 packed formats.

OpenGL 4.2 (Core Profile) - April 27, 2012
3.10 Fragment Shaders

The sequence of operations that are applied to fragments that result from rasterizing a point, line segment, or polygon are described using a fragment shader.

A fragment shader is an array of strings containing source code for the operations that are meant to occur on each fragment that results from rasterization. The language used for fragment shaders is described in the OpenGL Shading Language Specification.

Fragment shaders are created as described in section 2.11.1 using a type parameter of FRAGMENT_SHADER. They are attached to and used in program objects as described in section 2.11.3.

When the current fragment shader program object includes a fragment shader, its fragment shader is considered active, and is used to process fragments. If the fragment shader program object has no fragment shader, or no fragment shader program object is currently in use, the results of fragment shader execution are undefined.

3.10.1 Shader Variables

Fragment shaders can access uniforms belonging to the current program object. The amount of storage available for uniform variables, except for subroutine uniforms and atomic counters, in the default uniform block accessed by a fragment shader is specified by the value of the implementation-dependent constant MAX_FRAGMENT_UNIFORM_COMPONENTS. The implementation-dependent constant MAX_FRAGMENT_UNIFORM_VECTORS has a value equal to the value of MAX_FRAGMENT_UNIFORM_COMPONENTS divided by four. The total amount of combined storage available for uniform variables in all uniform blocks accessed by a fragment shader (including the default uniform block) is specified by the value of the implementation-dependent constant MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS. These values represent the numbers of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a fragment shader. A uniform matrix in the default uniform block with single-
3.10. FRAGMENT SHADERS

or double-precision components will consume no more than $4 \times \min(r,c)$ or $8 \times \min(r,c)$ uniform components, respectively. A scalar or vector uniform with double-precision components will consume no more than $2n$ components, where $n$ is 1 for scalars, and the component count for vectors. A link error is generated if an attempt is made to utilize more than the space available for fragment shader uniform variables. Uniforms are manipulated as described in section 2.11.7. Fragment shaders also have access to samplers to perform texturing operations, as described in section 2.11.9.

Fragment shaders can read input variables or inputs that correspond to the attributes of the fragments produced by rasterization.

The OpenGL Shading Language Specification defines a set of built-in inputs that can be be accessed by a fragment shader. These built-in inputs include data associated with a fragment such as the fragment’s position.

Additionally, the previous active shader stage may define one or more input variables (see section 2.11.11 and the OpenGL Shading Language Specification). The values of these user-defined inputs are, if not flat shaded, interpolated across the primitive being rendered. The results of these interpolations are available when inputs of the same name are defined in the fragment shader.

When interpolating input variables, the default screen-space location at which these variables are sampled is defined in previous rasterization sections. The default location may be overridden by interpolation qualifiers. When interpolating variables declared using `centroid` in, the variable is sampled at a location within the pixel covered by the primitive generating the fragment. When interpolating variables declared using `sample` in when `MULTISAMPLE` is enabled, the fragment shader will be invoked separately for each covered sample and the variable will be sampled at the corresponding sample point.

Additionally, built-in fragment shader functions provide further fine-grained control over interpolation. The built-in functions `interpolateAtCentroid` and `interpolateAtSample` will sample variables as though they were declared with the `centroid` or `sample` qualifiers, respectively. The built-in function `interpolateAtOffset` will sample variables at a specified $(x,y)$ offset relative to the center of the pixel. The range and granularity of offsets supported by this function is implementation-dependent. If either component of the specified offset is less than the value of `MIN_FRAGMENT_INTERPOLATION_OFFSET` or greater than the value of `MAX_FRAGMENT_INTERPOLATION_OFFSET`, the position used to interpolate the variable is undefined. Not all values of offset may be supported; $x$ and $y$ offsets may be rounded to fixed-point values with the number of fraction bits given by the value of the implementation-dependent constant `FRAGMENT_-INTERPOLATION_OFFSET_BITS`.

A fragment shader can also write to output variables. Values written to these
outputs are used in the subsequent per-fragment operations. Output variables can be used to write floating-point, integer or unsigned integer values destined for buffers attached to a framebuffer object, or destined for color buffers attached to the default framebuffer. The **Shader Outputs** subsection of section 3.10.2 describes how to direct these values to buffers.

### 3.10.2 Shader Execution

The executable version of the fragment shader is used to process incoming fragment values that are the result of rasterization.

**Texture Access**

The **Shader Only Texturing** subsection of section 2.11.12 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to fragment shaders.

When a texture lookup is performed in a fragment shader, the GL computes the filtered texture value $\tau$ in the manner described in sections 3.9.11 and 3.9.12, and converts it to a texture base color $C_b$ as shown in Table 3.23, followed by swizzling the components of $C_b$, controlled by the values of the texture parameters TEXTURE_SWIZZLE_R, TEXTURE_SWIZZLE_G, TEXTURE_SWIZZLE_B, and TEXTURE_SWIZZLE_A. If the value of TEXTURE_SWIZZLE_R is denoted by $swizzle_r$, swizzling computes the first component of $C_s$ according to

```plaintext
if (swizzle_r == RED)
    C_s[0] = C_b[0];
else if (swizzle_r == GREEN)
    C_s[0] = C_b[1];
else if (swizzle_r == BLUE)
    C_s[0] = C_b[2];
else if (swizzle_r == ALPHA)
    C_s[0] = A_b;
else if (swizzle_r == ZERO)
    C_s[0] = 0;
else if (swizzle_r == ONE)
    C_s[0] = 1; // float or int depending on texture component type
```

Swizzling of $C_s[1]$, $C_s[2]$, and $A_s$ are similarly controlled by the values of TEXTURE_SWIZZLE_G, TEXTURE_SWIZZLE_B, and TEXTURE_SWIZZLE_A, respectively.
Table 3.23: Correspondence of filtered texture components to texture base components.

<table>
<thead>
<tr>
<th>Texture Base Internal Format</th>
<th>Texture base color $C_b$</th>
<th>$A_b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED</td>
<td>$(R_t, 0, 0)$</td>
<td>1</td>
</tr>
<tr>
<td>RG</td>
<td>$(R_t, G_t, 0)$</td>
<td>1</td>
</tr>
<tr>
<td>RGB</td>
<td>$(R_t, G_t, B_t)$</td>
<td>1</td>
</tr>
<tr>
<td>RGBA</td>
<td>$(R_t, G_t, B_t)$</td>
<td>$A_t$</td>
</tr>
</tbody>
</table>

The resulting four-component vector $\left( R_s, G_s, B_s, A_s \right)$ is returned to the fragment shader. For the purposes of level-of-detail calculations, the derivatives $\frac{du}{dx}$, $\frac{du}{dy}$, $\frac{dv}{dx}$, $\frac{dv}{dy}$, and $\frac{dw}{dx}$ may be approximated by a differencing algorithm as detailed in section 8.8 of the OpenGL Shading Language Specification.

Texture lookups involving textures with depth component data generate a texture base color $C_b$ either using depth data directly or by performing a comparison with the $D_{ref}$ value used to perform the lookup, as described in section 3.9.17, and expanding the resulting value $R_t$ to a color $C_b = (R_t, 0, 0, 1)$. Swizzling is then performed as described above, but only the first component $C_s[0]$ is returned to the shader. The comparison operation is requested in the shader by using any of the shadow sampler types (sampler*Shadow), and in the texture using the TEXTURE_COMPARE_MODE parameter. These requests must be consistent; the results of a texture lookup are undefined if any of the following conditions are true:

- The sampler used in a texture lookup function is not one of the shadow sampler types, the texture object’s internal format is DEPTH_COMPONENT or DEPTH_STENCIL, and the TEXTURE_COMPARE_MODE is not NONE.
- The sampler used in a texture lookup function is one of the shadow sampler types, the texture object’s internal format is DEPTH_COMPONENT or DEPTH_STENCIL, and the TEXTURE_COMPARE_MODE is NONE.
- The sampler used in a texture lookup function is one of the shadow sampler types, and the texture object’s internal format is not DEPTH_COMPONENT or DEPTH_STENCIL.

The stencil index texture internal component is ignored if the base internal format is DEPTH_STENCIL.

If a sampler is used in a fragment shader and the sampler’s associated texture...
3.10. FRAGMENT SHADERS

is not complete, as defined in section 3.9.14, \((0, 0, 0, 1)\) will be returned for a non-shadow sampler and \(0\) for a shadow sampler.

The number of separate texture units that can be accessed from within a fragment shader during the rendering of a single primitive is specified by the implementation-dependent constant \texttt{MAX\_TEXTURE\_IMAGE\_UNITS}.

\section*{Shader Inputs}

The OpenGL Shading Language Specification describes the values that are available as inputs to the fragment shader.

The built-in variable \texttt{gl\_ FragCoord} holds the fragment coordinate \((x_f, y_f, z_f, w_f)\) for the fragment. Computing the fragment coordinate depends on the fragment processing pixel-center and origin conventions (discussed below) as follows:

\begin{equation}
\begin{aligned}
x_f &= \begin{cases} 
         x_w - \frac{1}{2}, & \text{pixel-center convention is integer} \\
         x_w, & \text{otherwise} 
        \end{cases} \\
y'_f &= \begin{cases} 
         H - y_w, & \text{origin convention is upper-left} \\
         y_w, & \text{otherwise} 
        \end{cases} \\
y_f &= \begin{cases} 
         y'_f - \frac{1}{2}, & \text{pixel-center convention is integer} \\
         y'_f, & \text{otherwise} 
        \end{cases} \\
z_f &= z_w \\
w_f &= \frac{1}{w_c}
\end{aligned}
\end{equation}

where \((x_w, y_w, z_w)\) is the fragment’s window-space position, \(w_c\) is the \(w\) component of the fragment’s clip-space position, and \(H\) is the window’s height in pixels. Note that \(z_w\) already has a polygon offset added in, if enabled (see section 3.6.4). \(z_f\) must be precisely \(0\) or \(1\) in the case where \(z_w\) is either \(0\) or \(1\), respectively. The \(\frac{1}{w}\) value is computed from the \(w_c\) coordinate (see section 2.14).

Unless otherwise specified by layout qualifiers in the fragment shader (see section 4.3.8.1 of the OpenGL Shading Language Specification), the fragment processing pixel-center convention is half-integer and the fragment processing origin convention is lower-left.

The built-in variable \texttt{gl\_FrontFacing} is set to \texttt{TRUE} if the fragment is generated from a front-facing primitive, and \texttt{FALSE} otherwise. For fragments generated from triangle primitives (including ones resulting from primitives rendered as points or lines), the determination is made by examining the sign of the area.
computed by equation 3.8 of section 3.6.1 (including the possible reversal of this sign controlled by \textbf{FrontFace}). If the sign is positive, fragments generated by the primitive are front-facing; otherwise, they are back-facing. All other fragments are considered front-facing.

If a geometry shader is active, the built-in variable \texttt{gl_PrimitiveID} contains the ID value emitted by the geometry shader for the provoking vertex. If no geometry shader is active, \texttt{gl_PrimitiveID} contains the number of primitives processed by the rasterizer since the last drawing command was called. The first primitive generated by a drawing command is numbered zero, and the primitive ID counter is incremented after every individual point, line, or polygon primitive is processed. For polygons drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may be drawn.

Restarting a primitive using the primitive restart index (see section 2.8) has no effect on the primitive ID counter.

\texttt{gl_PrimitiveID} is only defined under the same conditions that \texttt{gl_VertexID} is defined, as described under “Shader Inputs” in section 2.11.12.

Similarly to the limit on geometry shader output components (see section 2.13.4), there is a limit on the number of components of built-in and user-defined input variables that can be read by the fragment shader, given by the value of the implementation-dependent constant \texttt{MAX_FRAGMENT_INPUT_COMPONENTS}.

The built-in variable \texttt{gl_SampleMaskIn} is an integer array holding bitfields indicating the set of fragment samples covered by the primitive corresponding to the fragment shader invocation. The number of elements in the array is

$$\left\lfloor \frac{s}{32} \right\rfloor,$$

where \( s \) is the maximum number of color samples supported by the implementation. Bit \( n \) of element \( w \) in the array is set if and only if the sample numbered \( 32w + n \) is considered covered for this fragment shader invocation. When rendering to a non-multisample buffer, or if multisample rasterization is disabled, all bits are zero except for bit zero of the first array element. That bit will be one if the pixel is covered and zero otherwise. Bits in the sample mask corresponding to covered samples that will be killed due to \texttt{SAMPLE_COVERAGE} or \texttt{SAMPLE_MASK} will not be set (see section 4.1.3). When per-sample shading is active due to the use of a fragment input qualified by \texttt{sample}, only the bit for the current sample is set in \texttt{gl_SampleMaskIn}. When \texttt{GL} state specifies multiple fragment shader invocations for a given fragment, the sample mask for any single fragment shader invocation may specify a subset of the covered samples for the fragment. In this case, the bit corresponding to each covered sample will be set in exactly one
3.10. **FRAGMENT SHADERS**

Fragment shader invocation.

The built-in read-only variable `gl_SampleID` is filled with the sample number of the sample currently being processed. This variable is in the range 0 to `gl_NumSamples` minus one, where `gl_NumSamples` is the total number of samples in the framebuffer, or one if rendering to a non-multisample framebuffer. Using this variable in a fragment shader causes the entire shader to be evaluated per-sample. When rendering to a non-multisample buffer, or if multisample rasterization is disabled, `gl_SampleID` will always be zero. `gl_NumSamples` is the sample count of the framebuffer regardless of whether multisample rasterization is enabled or not.

The built-in read-only variable `gl_SamplePosition` contains the position of the current sample within the multi-sample draw buffer. The x and y components of `gl_SamplePosition` contain the sub-pixel coordinate of the current sample and will have values in the range [0, 1]. The sub-pixel coordinates of the center of the pixel are always (0.5, 0.5). Using this variable in a fragment shader causes the entire shader to be evaluated per-sample. When rendering to a non-multisample buffer, or if multisample rasterization is disabled, `gl_SamplePosition` will always be (0.5, 0.5).

When a program is linked, all components of any input variables read by a fragment shader will count against this limit. A program whose fragment shader exceeds this limit may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Component counting rules for different variable types and variable declarations are the same as for `MAX_VERTEX_OUTPUT_COMPONENTS`. (see section 2.11.11).

**Shader Outputs**

The OpenGL Shading Language Specification describes the values that may be output by a fragment shader. These outputs are split into two categories, user-defined outputs and the built-in outputs `gl_FragDepth` and `gl_SampleMask`.

For fixed-point depth buffers, the final fragment depth written by a fragment shader is first clamped to [0, 1] and then converted to fixed-point as if it were a window z value (see section 2.14.1). For floating-point depth buffers, conversion is not performed but clamping is. Note that the depth range computation is not applied here, only the conversion to fixed-point.

The built-in integer array `gl_SampleMask` can be used to change the sample coverage for a fragment from within the shader. The number of elements in the array is

\[
\left\lceil \frac{s}{32} \right\rceil,
\]

OpenGL 4.2 (Core Profile) - April 27, 2012
where \( s \) is the maximum number of color samples supported by the implementation. If bit \( n \) of element \( w \) in the array is set to zero, sample \( 32w + n \) should be considered uncovered for the purposes of multisample fragment operations (see section 4.1.3). Modifying the sample mask in this way may exclude covered samples from being processed further at a per-fragment granularity. However, setting sample mask bits to one will never enable samples not covered by the original primitive. If the fragment shader is being evaluated at any frequency other than per-fragment, bits of the sample mask not corresponding to the current fragment shader invocation are ignored.

Color values written by a fragment shader may be floating-point, signed integer, or unsigned integer. If the color buffer has a signed or unsigned normalized fixed-point format, color values are assumed to be floating-point and are converted to fixed-point as described in equations 2.4 or 2.3, respectively; otherwise no type conversion is applied. If the values written by the fragment shader do not match the format(s) of the corresponding color buffer(s), the result is undefined.

A fragment shader may not statically assign values to more than one user-defined output variable. In this case, a compile or link error will result. A shader statically assigns a value to a variable if, after pre-processing, it contains a statement that would write to the variable, whether or not run-time flow of control will cause that statement to be executed.

Writing to \texttt{gl_FragDepth} specifies the depth value for the fragment being processed. If the active fragment shader does not statically assign a value to \texttt{gl_FragDepth}, then the depth value generated during rasterization is used by subsequent stages of the pipeline. Otherwise, the value assigned to \texttt{gl_FragDepth} is used, and is undefined for any fragments where statements assigning a value to \texttt{gl_FragDepth} are not executed. Thus, if a shader statically assigns a value to \texttt{gl_FragDepth}, then it is responsible for always writing it.

The binding of a user-defined output variable to a fragment color number can be specified explicitly. The command

```c
void BindFragDataLocationIndexed( uint program,
                                  uint colorNumber, uint index, const char * name );
```

specifies that the output variable \textit{name} in \textit{program} should be bound to fragment color \textit{colorNumber} when the program is next linked. \textit{index} may be zero or one to specify that the color be used as either the first or second color input to the blend equation, respectively, as described in section 4.1.7.

If \textit{name} was bound previously, its assigned binding is replaced with \textit{colorNumber}. \textit{name} must be a null-terminated string. The error \texttt{INVALID_VALUE} is generated if \textit{index} is greater than one, if \textit{colorNumber} is greater than or equal to
the value of \texttt{MAX_DRAW_BUFFERS} and \texttt{index} is zero, or if \texttt{colorNumber} is greater than or equal to the value of \texttt{MAX_DUAL_SOURCE_DRAW_BUFFERS} and \texttt{index} is greater than or equal to one. The command

\begin{verbatim}
void BindFragDataLocation( uint program,
    uint colorNumber, const char * name );
\end{verbatim}

is equivalent to calling

\begin{verbatim}
BindFragDataLocationIndexed( program, colorNumber, 0, name );
\end{verbatim}

\texttt{BindFragDataLocation} has no effect until the program is linked. In particular, it doesn’t modify the bindings of outputs in a program that has already been linked. The error \texttt{INVALID_OPERATION} is generated if \texttt{name} starts with the reserved \texttt{gl} prefix.

When a program is linked, any output variables without a binding specified either through \texttt{BindFragDataLocationIndexed} or \texttt{BindFragDataLocation}, or explicitly set within the shader text will automatically be bound to fragment colors and indices. \texttt{by-the-GL}: All such assignments will use color indices of zero. Such bindings can be queried using the commands \texttt{GetFragDataLocation} and \texttt{GetFragDataIndex}. If an output has a binding explicitly set within the shader text and a different binding assigned by \texttt{BindFragDataLocationIndexed} or \texttt{BindFragDataLocation}, the assignment in the shader text is used. Output binding assignments will cause \texttt{LinkProgram} to fail:

\begin{itemize}
  \item if the number of active outputs is greater than the value of \texttt{MAX_DRAW_BUFFERS};
  \item if the program has an active output assigned to a location greater than or equal to the value of \texttt{MAX_DUAL_SOURCE_DRAW_BUFFERS} and has an active output assigned an index greater than or equal to one;
  \item if more than one output variable is bound to the same number and index; or
  \item if the explicit binding assignments do not leave enough space for the linker to automatically assign a location for an output array, which requires multiple contiguous locations.
\end{itemize}

\texttt{BindFragDataLocationIndexed} may be issued before any shader objects are attached to a program object. Hence it is allowed to bind any name (except a name starting with \texttt{gl}) to a color number and index, including a name that is never used.
as an output in any fragment shader object. Assigned bindings for variables that do
not exist are ignored.

After a program object has been linked successfully, the bindings of output
variable names to color numbers can be queried. The command

\[
\text{int GetFragDataLocation}(\text{uint program, const char } \ast \text{name});
\]

returns the number of the fragment color to which the output variable \text{name}
was bound when the program object \text{program} was last linked. \text{name} must be
a null-terminated string. If \text{program} has not been successfully linked, the error
\text{INVALID_OPERATION} is generated. If \text{name} is not an output variable, or if an
error occurs, -1 will be returned.

The command

\[
\text{int GetFragDataIndex}(\text{uint program, const char } \ast \text{name});
\]

returns the index of the fragment color to which the variable \text{name} was bound when
the program object \text{program} was last linked. If \text{program} has not been successfully
linked, the error \text{INVALID_OPERATION} is generated. If \text{name} is not an output
variable, or if an error occurs, -1 will be returned.

### Early Fragment Tests

An explicit control is provided to allow fragment shaders to enable early frag-
ment tests. If the fragment shader specifies the \text{early_fragment_tests} layout
qualifier, the per-fragment tests described in section 3.8 will be performed prior
to fragment shader execution. Otherwise, they will be performed after fragment
shader execution.

### 3.11 Antialiasing Application

If antialiasing is enabled for the primitive from which a rasterized fragment was
produced, then the computed coverage value is applied to the fragment. The value
is multiplied by the fragment’s alpha (A) value to yield a final alpha value. The
coverage value is applied separately to each fragment color, and only applied if the
corresponding color buffer in the framebuffer has a fixed- or floating-point format.
3.12 Multisample Point Fade

Finally, if multisampling is enabled and the rasterized fragment results from a point primitive, then the computed fade factor from equation 3.2 is applied to the fragment. The fade factor is multiplied by the fragment’s alpha value to yield a final alpha value. The fade factor is applied separately to each fragment color, and only applied if the corresponding color buffer in the framebuffer has a fixed- or floating-point format.
Chapter 4

Per-Fragment Operations and the Framebuffer

The framebuffer, whether it is the default framebuffer or a framebuffer object (see section 2.1), consists of a set of pixels arranged as a two-dimensional array. For purposes of this discussion, each pixel in the framebuffer is simply a set of some number of bits. The number of bits per pixel may vary depending on the GL implementation, the type of framebuffer selected, and parameters specified when the framebuffer was created. Creation and management of the default framebuffer is outside the scope of this specification, while creation and management of framebuffer objects is described in detail in section 4.4.

Corresponding bits from each pixel in the framebuffer are grouped together into a bitplane; each bitplane contains a single bit from each pixel. These bitplanes are grouped into several logical buffers. These are the color, depth, and stencil buffers. The color buffer actually consists of a number of buffers, and these color buffers serve related but slightly different purposes depending on whether the GL is bound to the default framebuffer or a framebuffer object.

For the default framebuffer, the color buffers are the front left buffer, the front right buffer, the back left buffer, and the back right buffer. Typically the contents of the front buffers are displayed on a color monitor while the contents of the back buffers are invisible. (Monoscopic contexts display only the front left buffer; stereoscopic contexts display both the front left and the front right buffers.) All color buffers must have the same number of bitplanes, although an implementation or context may choose not to provide right buffers, or back buffers at all. Further, an implementation or context may choose not to provide depth or stencil buffers. If no default framebuffer is associated with the GL context, the framebuffer is incomplete except when a framebuffer object is bound (see sections 4.4.1 and 4.4.4).
4.1. **PER-FRAGMENT OPERATIONS**

Framebuffer objects are not visible, and do not have any of the color buffers present in the default framebuffer. Instead, the buffers of an framebuffer object are specified by attaching individual textures or renderbuffers (see section 4.4) to a set of attachment points. A framebuffer object has an array of color buffer attachment points, numbered zero through \( n \), a depth buffer attachment point, and a stencil buffer attachment point. In order to be used for rendering, a framebuffer object must be **complete**, as described in section 4.4.4. Not all attachments of a framebuffer object need to be populated.

Each pixel in a color buffer consists of up to four color components. The four color components are named R, G, B, and A, in that order; color buffers are not required to have all four color components. R, G, B, and A components may be represented as signed or unsigned normalized fixed-point, floating-point, or signed or unsigned integer values; all components must have the same representation. Each pixel in a depth buffer consists of a single unsigned integer value in the format described in section 2.14.1 or a floating-point value. Each pixel in a stencil buffer consists of a single unsigned integer value.

The number of bitplanes in the color, depth, and stencil buffers is dependent on the currently bound framebuffer. For the default framebuffer, the number of bitplanes is fixed. For framebuffer objects, the number of bitplanes in a given logical buffer may change if the image attached to the corresponding attachment point changes.

The GL has two active framebuffers; the **draw framebuffer** is the destination for rendering operations, and the **read framebuffer** is the source for readback operations. The same framebuffer may be used for both drawing and reading. Section 4.4.1 describes the mechanism for controlling framebuffer usage.

The default framebuffer is initially used as the draw and read framebuffer, and the initial state of all provided bitplanes is undefined. The format and encoding of buffers in the draw and read framebuffers can be queried as described in section 6.1.3.

### 4.1 Per-Fragment Operations

A fragment produced by rasterization with window coordinates of \((x_w, y_w)\) modifies the pixel in the framebuffer at that location based on a number of parameters and conditions. We describe these modifications and tests, diagrammed in figure 4.1, in the order in which they are performed. Figure 4.1 diagrams these

---

1The window system binding API may allow associating a GL context with two separate “default framebuffers” provided by the window system as the draw and read framebuffers, but if so, both default framebuffers are referred to by the name zero at their respective binding points.
4.1. PER-FRAGMENT OPERATIONS

4.1.1 Pixel Ownership Test

The first test is to determine if the pixel at location \((x_w, y_w)\) in the framebuffer is currently owned by the GL (more precisely, by this GL context). If it is not, the window system decides the fate of the incoming fragment. Possible results are that the fragment is discarded or that some subset of the subsequent per-fragment operations are applied to the fragment. This test allows the window system to control the GL's behavior, for instance, when a GL window is obscured.

If the draw framebuffer is a framebuffer object (see section 4.2.1), the pixel ownership test always passes, since the pixels of framebuffer objects are owned by the GL, not the window system. If the draw framebuffer is the default framebuffer, the window system controls pixel ownership.

4.1.2 Scissor Test

The scissor test determines if \((x_w, y_w)\) lies within the scissor rectangle defined by four values for each viewport. These values are set with

---

Figure 4.1. Per-fragment operations.
void ScissorArrayv(uint first, sizei count, const int *v);
void ScissorIndexed(uint index, int left, int bottom, sizei width, sizei height);
void ScissorIndexedv(uint index, int *v);
void Scissor(int left, int bottom, sizei width, sizei height);

ScissorArrayv defines a set of scissor rectangles that are each applied to the corresponding viewport (see section 2.14.1). first specifies the index of the first scissor rectangle to modify, and count specifies the number of scissor rectangles. If (first + count) is greater than the value of MAX_VIEWPORTS, then an INVALID_VALUE error is generated. v contains the address of an array of integers containing the left, bottom, width and height of the scissor rectangles, in that order.

If left ≤ xw < left + width and bottom ≤ yw < bottom + height for the selected scissor rectangle, then the scissor test passes. Otherwise, the test fails and the fragment is discarded. For points, lines, and polygons, the scissor rectangle for a primitive is selected in the same manner as the viewport (see section 2.14.1).

For buffer clears (see section 4.2.3) and pixel rectangles, the scissor rectangle numbered zero is used for the scissor test.

The scissor test is enabled or disabled for all viewports using Enable or Disable with the symbolic constant SCISSOR_TEST. The test is enabled or disabled for a specific viewport using Enablei or Disablei with the constant SCISSOR_TEST and the index of the selected viewport. When disabled, it is as if the scissor test always passes. The value of the scissor test enable for viewport i can be queried by calling IsEnabledi with target SCISSOR_TEST and index i. The value of the scissor test enable for viewport zero may also be queried by calling IsEnabled with the same symbolic constant, but no index parameter. If either width or height is less than zero for any scissor rectangle, then an INVALID_VALUE error is generated. If the viewport index specified to Enablei, Disablei or IsEnabledi is greater or equal to the value of MAX_VIEWPORTS, then an INVALID_VALUE error is generated.

The state required consists of four integer values per viewport, and a bit indicating whether the test is enabled or disabled for each viewport. In the initial state, left = bottom = 0, and width and height are determined by the size of the window into which the GL is to do its rendering for all viewports. If the default framebuffer is bound but no default framebuffer is associated with the GL context (see chapter 4), then width and height are initially set to zero. Initially, the scissor test is disabled for all viewports.

ScissorIndexed and ScissorIndexedv specify the scissor rectangle for a single viewport and are equivalent (assuming no errors are generated) to:
4.1. PER-FRAGMENT OPERATIONS

```c
int v[] = { left, bottom, width, height };
ScissorArrayv(index, 1, v);
```

and

```c
ScissorArrayv(index, 1, v);
```

respectively.

`Scissor` sets the scissor rectangle for all viewports to the same values and is equivalent (assuming no errors are generated) to:

```c
for (uint i = 0; i < MAX_VIEWPORTS; i++) {
    ScissorIndexed(i, left, bottom, width, height);
}
```

Calling `Enable` or `Disable` with the symbolic constant `SCISSOR_TEST` is equivalent, assuming no errors, to:

```c
for (uint i = 0; i < MAX_VIEWPORTS; i++) {
    Enablei(SCISSOR_TEST, i);
    /* or */
    Disablei(SCISSOR_TEST, i);
}
```

4.1.3 Multisample Fragment Operations

This step modifies fragment alpha and coverage values based on the values of `SAMPLE_ALPHA_TO_COVERAGE`, `SAMPLE_ALPHA_TO_ONE`, `SAMPLE_COVERAGE`, `SAMPLE_COVERAGE_VALUE`, `SAMPLE_COVERAGE_INVERT`, `SAMPLE_MASK`, `SAMPLE_MASK_VALUE`, and an output sample mask optionally written by the fragment shader. No changes to the fragment alpha or coverage values are made at this step if `MULTISAMPLE` is disabled, or if the value of `SAMPLE_BUFFERS` is not one.

All alpha values in this section refer only to the alpha component of the fragment shader output linked to color number zero, index zero (see section 3.10.2). If the fragment shader does not write to this output, the alpha value is undefined.

`SAMPLE_ALPHA_TO_COVERAGE`, `SAMPLE_ALPHA_TO_ONE`, `SAMPLE_COVERAGE`, and `SAMPLE_MASK` are enabled and disabled by calling `Enable` and `Disable` with the desired token value. All four values are queried by calling `IsEnabled` with the desired token value. If drawbuffer zero is not `NONE` and the buffer...
4.1. **PER-FRAGMENT OPERATIONS**

If **SAMPLE_ALPHA_TO_COVERAGE** is enabled, a temporary coverage value is generated where each bit is determined by the alpha value at the corresponding sample location. The temporary coverage value is then ANDed with the fragment coverage value to generate a new fragment coverage value. If the fragment shader outputs an integer to color number zero, index zero when not rendering to an integer format, the coverage value is undefined.

No specific algorithm is required for converting the sample alpha values to a temporary coverage value. It is intended that the number of 1's in the temporary coverage be proportional to the set of alpha values for the fragment, with all 1's corresponding to the maximum of all alpha values, and all 0's corresponding to all alpha values being 0. The alpha values used to generate a coverage value are clamped to the range [0, 1]. It is also intended that the algorithm be pseudo-random in nature, to avoid image artifacts due to regular coverage sample locations. The algorithm can and probably should be different at different pixel locations. If it does differ, it should be defined relative to window, not screen, coordinates, so that rendering results are invariant with respect to window position.

Next, if **SAMPLE_ALPHA_TO_ONE** is enabled, each alpha value is replaced by the maximum representable alpha value for fixed-point color buffers, or by 1.0 for floating-point buffers. Otherwise, the alpha values are not changed.

Next, if a fragment shader is active and statically assigns to the built-in output variable **gl_SampleMask**, the fragment coverage is ANDed with the bits of the sample mask. If such a fragment shader did not assign a value to **gl_SampleMask** due to flow control, the value ANDed with the fragment coverage is undefined. If no fragment shader is active, or if the active fragment shader does not statically assign values to **gl_SampleMask**, the fragment coverage is not modified.

Next, if **SAMPLE_COVERAGE** is enabled, the fragment coverage is ANDed with another temporary coverage. This temporary coverage is generated in the same manner as the one described above, but as a function of the value of **SAMPLE_COVERAGE_VALUE**. The function need not be identical, but it must have the same properties of proportionality and invariance. If **SAMPLE_COVERAGE_INVERT** is **TRUE**, the temporary coverage is inverted (all bit values are inverted) before it is ANDed with the fragment coverage.

The values of **SAMPLE_COVERAGE_VALUE** and **SAMPLE_COVERAGE_INVERT** are specified by calling

```c
void SampleCoverage( float value, boolean invert );
```

with `value` set to the desired coverage value, and `invert` set to **TRUE** or **FALSE**. `value` is clamped to [0, 1] before being stored as **SAMPLE_COVERAGE_VALUE**.
4.1. PER-FRAGMENT OPERATIONS

SAMPLE_COVERAGE_VALUE is queried by calling 
GetFloatv with pname set to
SAMPLE_COVERAGE_VALUE. SAMPLE_COVERAGE_INVERT is queried by calling
GetBooleanv with pname set to SAMPLE_COVERAGE_INVERT.

Finally, if SAMPLE_MASK is enabled, the fragment coverage is ANDed with
the coverage value SAMPLE_MASK_VALUE. The value of SAMPLE_MASK_VALUE is
specified using

```c
void SampleMaski( uint maskNumber, bitfield mask );
```

with mask set to the desired mask for mask word maskNumber. SAMPLE_MASK_-
VALUE is queried by calling GetIntegeriv with pname set to SAMPLE_MASK_-
VALUE and the index set to maskNumber. Bit B of mask word M corresponds to
sample $32 \times M + B$ as described in section 3.3.1. The error INVALID_VALUE is
generated if the mask word indexed is greater than or equal to the value of MAX_-SAMPLE_MASK_WORDS.

4.1.4 Stencil Test

The stencil test conditionally discards a fragment based on the outcome of a com-
parison between the value in the stencil buffer at location $(x_w, y_w)$ and a reference
value. The test is enabled or disabled with the Enable and Disable commands,
using the symbolic constant STENCIL_TEST. When disabled, the stencil test and
associated modifications are not made, and the fragment is always passed.

The stencil test is controlled with

```c
void StencilFunc( enum func, int ref, uint mask );
void StencilFuncSeparate( enum face, enum func, int ref,
uint mask );
void StencilOp( enum sfail, enum dpfail, enum dpass );
void StencilOpSeparate( enum face, enum sfail, enum dpfail,
enum dpass );
```

There are two sets of stencil-related state, the front stencil state set and the
back stencil state set. Stencil tests and writes use the front set of stencil state
when processing fragments rasterized from non-polygon primitives (points and
lines) and front-facing polygon primitives while the back set of stencil state is
used when processing fragments rasterized from back-facing polygon primitives.
For the purposes of stencil testing, a primitive is still considered a polygon even if
the polygon is to be rasterized as points or lines due to the current polygon mode.
Whether a polygon is front- or back-facing is determined in the same manner used
for face culling (see section 3.6.1).

OpenGL 4.2 (Core Profile) - April 27, 2012
StencilFuncSeparate and StencilOpSeparate take a face argument which can be FRONT, BACK, or FRONT_AND_BACK and indicates which set of state is affected. StencilFunc and StencilOp set front and back stencil state to identical values.

StencilFunc and StencilFuncSeparate take three arguments that control whether the stencil test passes or fails. ref is an integer reference value that is used in the unsigned stencil comparison. Stencil comparison operations and queries of ref clamp its value to the range $[0, 2^s - 1]$, where $s$ is the number of bits in the stencil buffer attached to the draw framebuffer. The $s$ least significant bits of mask are bitwise ANDed with both the reference and the stored stencil value, and the resulting masked values are those that participate in the comparison controlled by func. func is a symbolic constant that determines the stencil comparison function; the eight symbolic constants are NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GEQUAL, GREATER, or NOTEQUAL. Accordingly, the stencil test passes never, always, and if the masked reference value is less than, less than or equal to, equal to, greater than or equal to, greater than, or not equal to the masked stored value in the stencil buffer.

StencilOp and StencilOpSeparate take three arguments that indicate what happens to the stored stencil value if this or certain subsequent tests fail or pass. sfail indicates what action is taken if the stencil test fails. The symbolic constants are KEEP, ZERO, REPLACE, INCR, DECR, INVERT, INCR_WRAP, and DECR_WRAP. These correspond to keeping the current value, setting to zero, replacing with the reference value, incrementing with saturation, decrementing with saturation, bitwise inverting it, incrementing without saturation, and decrementing without saturation.

For purposes of increment and decrement, the stencil bits are considered as an unsigned integer. Incrementing or decrementing with saturation clamps the stencil value at 0 and the maximum representable value. Incrementing or decrementing without saturation will wrap such that incrementing the maximum representable value results in 0, and decrementing 0 results in the maximum representable value.

The same symbolic values are given to indicate the stencil action if the depth buffer test (see section 4.1.5) fails (dpfail), or if it passes (dppass).

If the stencil test fails, the incoming fragment is discarded. The state required consists of the most recent values passed to StencilFunc or StencilFuncSeparate and to StencilOp or StencilOpSeparate, and a bit indicating whether stencil testing is enabled or disabled. In the initial state, stenciling is disabled, the front and back stencil reference value are both zero, the front and back stencil comparison functions are both ALWAYS, and the front and back stencil mask are both set to the value $2^s - 1$, where $s$ is greater than or equal to the number of bits in the deepest stencil buffer supported by the GL implementation. Initially, all three front and back stencil operations are KEEP.
4.1. PER-FRAGMENT OPERATIONS

If there is no stencil buffer, no stencil modification can occur, and it is as if the stencil tests always pass, regardless of any calls to StencilFunc.

4.1.5 Depth Buffer Test

The depth buffer test discards the incoming fragment if a depth comparison fails. The comparison is enabled or disabled with the generic Enable and Disable commands using the symbolic constant DEPTH_TEST. When disabled, the depth comparison and subsequent possible updates to the depth buffer value are bypassed and the fragment is passed to the next operation. The stencil value, however, is modified as indicated below as if the depth buffer test passed. If enabled, the comparison takes place and the depth buffer and stencil value may subsequently be modified.

The comparison is specified with

```c
void DepthFunc( enum func );
```

This command takes a single symbolic constant: one of NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GREATER, GEQUAL, NOTEQUAL. Accordingly, the depth buffer test passes never, always, if the incoming fragment’s $z_w$ value is less than, less than or equal to, equal to, greater than, greater than or equal to, or not equal to the depth value stored at the location given by the incoming fragment’s $(x_w, y_w)$ coordinates.

If depth clamping (see section 2.20) is enabled, before the incoming fragment’s $z_w$ is compared $z_w$ is clamped to the range $[\min(n, f), \max(n, f)]$, where $n$ and $f$ are the current near and far depth range values (see section 2.14.1).

If the depth buffer test fails, the incoming fragment is discarded. The stencil value at the fragment’s $(x_w, y_w)$ coordinates is updated according to the function currently in effect for depth buffer test failure. Otherwise, the fragment continues to the next operation and the value of the depth buffer at the fragment’s $(x_w, y_w)$ location is set to the fragment’s $z_w$ value. In this case the stencil value is updated according to the function currently in effect for depth buffer test success.

The necessary state is an eight-valued integer and a single bit indicating whether depth buffering is enabled or disabled. In the initial state the function is LESS and the test is disabled.

If there is no depth buffer, it is as if the depth buffer test always passes.

4.1.6 Occlusion Queries

Occlusion queries use query objects to track the number of fragments or samples that pass the depth test. An occlusion query can be started and finished by calling
BeginQuery and EndQuery, respectively, with a target of SAMPLES_PASSED or ANY_SAMPLES_PASSED.

When an occlusion query is started with target SAMPLES_PASSED, the samples-passed count maintained by the GL is set to zero. When an occlusion query is active, the samples-passed count is incremented for each fragment that passes the depth test. If the value of SAMPLE_BUFFERS is 0, then the samples-passed count is incremented by 1 for each fragment. If the value of SAMPLE_BUFFERS is 1, then the samples-passed count is incremented by the number of samples whose coverage bit is set. However, implementations, at their discretion, may instead increase the samples-passed count by the value of SAMPLES if any sample in the fragment is covered.

When an occlusion query finishes and all fragments generated by commands issued prior to EndQuery have been generated, the samples-passed count is written to the corresponding query object as the query result value, and the query result for that object is marked as available.

When an occlusion query is started with the target ANY_SAMPLES_PASSED, the samples-boolean state maintained by the GL is set to FALSE. While that occlusion query is active, the samples-boolean state is set to TRUE if any fragment or sample passes the depth test. When the occlusion query finishes, the samples-boolean state of FALSE or TRUE is written to the corresponding query object as the query result value, and the query result for that object is marked as available.

4.1.7 Blending

Blending combines the incoming source fragment’s R, G, B, and A values with the destination R, G, B, and A values stored in the framebuffer at the fragment’s \((x_w, y_w)\) location.

Source and destination values are combined according to the blend equation, quadruplets of source and destination weighting factors determined by the blend functions, and a constant blend color to obtain a new set of R, G, B, and A values, as described below.

If the color buffer is fixed-point, the components of the source and destination values and blend factors are each clamped to \([0, 1]\) or \([-1, 1]\) respectively for an unsigned normalized or signed normalized color buffer prior to evaluating the blend equation. If the color buffer is floating-point, no clamping occurs. The resulting four values are sent to the next operation.

Blending applies only if the color buffer has a fixed-point or floating-point format. If the color buffer has an integer format, proceed to the next operation.

Blending is enabled or disabled for an individual draw buffer with the commands

OpenGL 4.2 (Core Profile) - April 27, 2012
void Enablei ( enum target, uint index );
void Disablei ( enum target, uint index );

target is the symbolic constant BLEND and index is an integer \( i \) specifying the draw buffer associated with the symbolic constant DRAW_BUFFER\(_i\). If the color buffer associated with DRAW_BUFFER\(_i\) is one of FRONT, BACK, LEFT, RIGHT, or FRONT_\_AND_\_BACK (specifying multiple color buffers), then the state enabled or disabled is applicable for all of the buffers. Blending can be enabled or disabled for all draw buffers using Enable or Disable with the symbolic constant BLEND. If blending is disabled for a particular draw buffer, or if logical operation on color values is enabled (section 4.1.10), proceed to the next operation.

An INVALID_VALUE error is generated if index is greater than the value of MAX_DRAW_BUFFERS minus one.

If multiple fragment colors are being written to multiple buffers (see section 4.2.1), blending is computed and applied separately for each fragment color and the corresponding buffer.

**Blend Equation**

Blending is controlled by the blend equation. This equation can be simultaneously set to the same value for all draw buffers using the commands

void BlendEquation ( enum mode );
void BlendEquationSeparate ( enum modeRGB, 
                           enum modeAlpha );

or for an individual draw buffer using the indexed commands

void BlendEquationi ( uint buf, enum mode );
void BlendEquationSeparatei( uint buf, enum modeRGB, 
                          enum modeAlpha );

BlendEquationSeparate and BlendEquationSeparatei argument modeRGB determines the RGB blend equation while modeAlpha determines the alpha blend equation. BlendEquation and BlendEquationi argument mode determines both the RGB and alpha blend equations. mode, modeRGB, and modeAlpha must be one of FUNC_ADD, FUNC_SUBTRACT, FUNC_REVERSE_SUBTRACT, MIN, or MAX. BlendEquation and BlendEquationSeparate modify the blend equations for all draw buffers. BlendEquationi and BlendEquationSeparatei modify the blend equations associated with an individual draw buffer. The buf argument is an integer \( i \) that indicates that the blend equations should be modified for DRAW_BUFFER\(_i\).
An `INVALID_VALUE` error is generated if `buf` is not in the range zero to the value of `MAX_DRAW_BUFFERS` minus one.

Signed or unsigned normalized fixed-point destination (framebuffer) components are represented as described in section 2.1.2. Constant color components, floating-point destination components, and source (fragment) components are taken to be floating-point values. If source components are represented internally by the GL as fixed-point values, they are also interpreted according to section 2.1.2.

Prior to blending, signed and unsigned normalized fixed-point color components undergo an implied conversion to floating-point using equations 2.2 and 2.1, respectively. This conversion must leave the values 0 and 1 invariant. Blending computations are treated as if carried out in floating-point, and will be performed with a precision and dynamic range no lower than that used to represent destination components.

If `FRAMEBUFFER_SRGB` is enabled and the value of `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` for the framebuffer attachment corresponding to the destination buffer is SRGB (see section 6.1.3), the R, G, and B destination color values (after conversion from fixed-point to floating-point) are considered to be encoded for the sRGB color space and hence must be linearized prior to their use in blending. Each R, G, and B component is converted in the same fashion described for sRGB texture components in section 3.9.18.

If `FRAMEBUFFER_SRGB` is disabled or the value of `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` is not SRGB, no linearization is performed.

The resulting linearized R, G, and B and unmodified A values are recombined as the destination color used in blending computations.

Table 4.1 provides the corresponding per-component blend equations for each mode, whether acting on RGB components for `modeRGB` or the alpha component for `modeAlpha`.

In the table, the `s` subscript on a color component abbreviation (R, G, B, or A) refers to the source color component for an incoming fragment, the `d` subscript on a color component abbreviation refers to the destination color component at the corresponding framebuffer location, and the `c` subscript on a color component abbreviation refers to the constant blend color component. A color component abbreviation without a subscript refers to the new color component resulting from blending. Additionally, `S_r`, `S_g`, `S_b`, and `S_a` are the red, green, blue, and alpha components of the source weighting factors determined by the source blend function, and `D_r`, `D_g`, `D_b`, and `D_a` are the red, green, blue, and alpha components of the destination weighting factors determined by the destination blend function. Blend functions are described below.
### 4.1. PER-FRAGMENT OPERATIONS

#### Table 4.1: RGB and alpha blend equations.

<table>
<thead>
<tr>
<th>Mode</th>
<th>RGB Components</th>
<th>Alpha Component</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FUNC_ADD</strong></td>
<td>( R = R_s \cdot S_r + R_d \cdot D_r )</td>
<td>( A = A_s \cdot S_a + A_d \cdot D_a )</td>
</tr>
<tr>
<td></td>
<td>( G = G_s \cdot S_g + G_d \cdot D_g )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( B = B_s \cdot S_b + B_d \cdot D_b )</td>
<td></td>
</tr>
<tr>
<td><strong>FUNC_SUBTRACT</strong></td>
<td>( R = R_s \cdot S_r - R_d \cdot D_r )</td>
<td>( A = A_s \cdot S_a - A_d \cdot D_a )</td>
</tr>
<tr>
<td></td>
<td>( G = G_s \cdot S_g - G_d \cdot D_g )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( B = B_s \cdot S_b - B_d \cdot D_b )</td>
<td></td>
</tr>
<tr>
<td><strong>FUNC_REVERSE_SUBTRACT</strong></td>
<td>( R = R_d \cdot D_r - R_s \cdot S_r )</td>
<td>( A = A_d \cdot D_a - A_s \cdot S_a )</td>
</tr>
<tr>
<td></td>
<td>( G = G_d \cdot D_g - G_s \cdot S_g )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( B = B_d \cdot D_b - B_s \cdot S_b )</td>
<td></td>
</tr>
<tr>
<td><strong>MIN</strong></td>
<td>( R = \min(R_s, R_d) )</td>
<td>( A = \min(A_s, A_d) )</td>
</tr>
<tr>
<td></td>
<td>( G = \min(G_s, G_d) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( B = \min(B_s, B_d) )</td>
<td></td>
</tr>
<tr>
<td><strong>MAX</strong></td>
<td>( R = \max(R_s, R_d) )</td>
<td>( A = \max(A_s, A_d) )</td>
</tr>
<tr>
<td></td>
<td>( G = \max(G_s, G_d) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( B = \max(B_s, B_d) )</td>
<td></td>
</tr>
</tbody>
</table>

#### Blend Functions

The weighting factors used by the blend equation are determined by the blend functions. There are four possible sources for weighting factors. These are the constant color \((R_c, G_c, B_c, A_c)\) set with **BlendColor** (see below), the first source color \((R_{s0}, G_{s0}, B_{s0}, A_{s0})\), the second source color \((R_{s1}, G_{s1}, B_{s1}, A_{s1})\), and the destination color (the existing content of the draw buffer). Additionally the special constants **ZERO** and **ONE** are available as weighting factors.

Blend functions are simultaneously specified for all draw buffers using the commands

```c
void BlendFunc( enum src, enum dst );
void BlendFuncSeparate( enum srcRGB, enum dstRGB, 
                        enum srcAlpha, enum dstAlpha );
```

or for an individual draw buffer using the indexed commands

```c
void BlendFunci( uint buf, enum src, enum dst );
void BlendFuncSeparatei( uint buf, enum srcRGB, 
                        enum dstRGB, enum srcAlpha, enum dstAlpha );
```
BlendFuncSeparate and BlendFuncSeparatei arguments srcRGB and dstRGB determine the source and destination RGB blend functions, respectively, while srcAlpha and dstAlpha determine the source and destination alpha blend functions. BlendFunc and BlendFunci argument src determines both RGB and alpha source functions, while dst determines both RGB and alpha destination functions. BlendFuncSeparate and BlendFunc modify the blend functions for all draw buffers. BlendFuncSeparatei and BlendFunci modify the blend functions associated with an individual draw buffer. The buf argument is an integer i that indicates that the blend equations should be modified for DRAW_BUFFERi.

An INVALID_VALUE error is generated1 if buf is not in the range zero to the value of MAX_DRAW_BUFFERS minus one.

The possible source and destination blend functions and their corresponding computed blend factors are summarized in table 4.2.

Dual Source Blending and Multiple Draw Buffers

Blend functions that require the second color input, \((R_{s1}, G_{s1}, B_{s1}, A_{s1}) \) (SRC1_-COLOR, SRC1_ALPHA, ONE_MINUS_SRC1_COLOR, or ONE_MINUS_SRC1_ALPHA) may consume hardware resources that could otherwise be used for rendering to multiple draw buffers. Therefore, the number of draw buffers that can be attached to a frame buffer may be lower when using dual-source blending.

The maximum number of draw buffers that may be attached to a single frame buffer when using dual-source blending functions is implementation dependent and can be queried by calling GetIntegerv with the symbolic constant MAX_-DUAL_SOURCE_DRAW_BUFFERS. When using dual-source blending, MAX_DUAL_-SOURCE_DRAW_BUFFERS should be used in place of MAX_DRAW_BUFFERS to determine the maximum number of draw buffers that may be attached to a single frame buffer. The value of MAX_DUAL_SOURCE_DRAW_BUFFERS must be at least 1. If the value of MAX_DUAL_SOURCE_DRAW_BUFFERS is 1, then dual-source blending and multiple draw buffers cannot be used simultaneously.

If either blend function requires the second color input for any draw buffer, and any draw buffers greater than or equal to the value of MAX_DUAL_SOURCE_DRAW_BUFFERS have values other than NONE, the error INVALID_OPERATION is generated by DrawArrays and the other drawing commands defined in section 2.8.3.

Generation of Second Color Source for Blending

When using a fragment shader with dual-source blending functions, the color outputs are bound to the first and second inputs of a draw buffer using BindFragDataLocationIndexed as described in the Shader Outputs subsection of sec-
### 4.1. PER-FRAGMENT OPERATIONS

#### Table 4.2: RGB and ALPHA source and destination blending functions and the corresponding blend factors. Addition and subtraction of triplets is performed component-wise.

<table>
<thead>
<tr>
<th>Function</th>
<th>RGB Blend Factors</th>
<th>Alpha Blend Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZERO</td>
<td>$(S_r, S_g, S_b)$ or $(D_r, D_g, D_b)$</td>
<td>$S_a$ or $D_a$</td>
</tr>
<tr>
<td>ONE</td>
<td>$(1, 1, 1)$</td>
<td>1</td>
</tr>
<tr>
<td>SRC_COLOR</td>
<td>$(R_{s0}, G_{s0}, B_{s0})$</td>
<td>$A_{s0}$</td>
</tr>
<tr>
<td>ONE_MINUS_SRC_COLOR</td>
<td>$(1, 1, 1) - (R_{s0}, G_{s0}, B_{s0})$</td>
<td>$1 - A_{s0}$</td>
</tr>
<tr>
<td>DST_COLOR</td>
<td>$(R_d, G_d, B_d)$</td>
<td>$A_d$</td>
</tr>
<tr>
<td>ONE_MINUS_DST_COLOR</td>
<td>$(1, 1, 1) - (R_d, G_d, B_d)$</td>
<td>$1 - A_d$</td>
</tr>
<tr>
<td>SRC_ALPHA</td>
<td>$(A_{s0}, A_{s0}, A_{s0})$</td>
<td>$A_{s0}$</td>
</tr>
<tr>
<td>ONE_MINUS_SRC_ALPHA</td>
<td>$(1, 1, 1) - (A_{s0}, A_{s0}, A_{s0})$</td>
<td>$1 - A_{s0}$</td>
</tr>
<tr>
<td>DST_ALPHA</td>
<td>$(A_d, A_d, A_d)$</td>
<td>$A_d$</td>
</tr>
<tr>
<td>ONE_MINUS_DST_ALPHA</td>
<td>$(1, 1, 1) - (A_d, A_d, A_d)$</td>
<td>$1 - A_d$</td>
</tr>
<tr>
<td>CONSTANT_COLOR</td>
<td>$(R_c, G_c, B_c)$</td>
<td>$A_c$</td>
</tr>
<tr>
<td>ONE_MINUS_CONSTANT_COLOR</td>
<td>$(1, 1, 1) - (R_c, G_c, B_c)$</td>
<td>$1 - A_c$</td>
</tr>
<tr>
<td>CONSTANT_ALPHA</td>
<td>$(A_c, A_c, A_c)$</td>
<td>$A_c$</td>
</tr>
<tr>
<td>ONE_MINUS_CONSTANT_ALPHA</td>
<td>$(1, 1, 1) - (A_c, A_c, A_c)$</td>
<td>$1 - A_c$</td>
</tr>
<tr>
<td>SRC_ALPHA_SATURATE</td>
<td>$(f, f, f)^1$</td>
<td>1</td>
</tr>
<tr>
<td>SRC_COLOR_SATURATE</td>
<td>$(R_{s1}, G_{s1}, B_{s1})$</td>
<td>$A_{s1}$</td>
</tr>
<tr>
<td>ONE_MINUS_SRC_COLOR_SATURATE</td>
<td>$(1, 1, 1) - (R_{s1}, G_{s1}, B_{s1})$</td>
<td>$1 - A_{s1}$</td>
</tr>
<tr>
<td>SRC_ALPHA_SATURATE</td>
<td>$(A_{s1}, A_{s1}, A_{s1})$</td>
<td>$A_{s1}$</td>
</tr>
<tr>
<td>ONE_MINUS_SRC_ALPHA_SATURATE</td>
<td>$(1, 1, 1) - (A_{s1}, A_{s1}, A_{s1})$</td>
<td>$1 - A_{s1}$</td>
</tr>
</tbody>
</table>

\(^1 f = \min(A_{s0}, 1 - A_d)\).
tion 3.10.2. Data written to the first of these outputs becomes the first source color input to the blender (corresponding to \texttt{SRC\_COLOR} and \texttt{SRC\_ALPHA}). Data written to the second of these outputs generates the second source color input to the blender (corresponding to \texttt{SRC1\_COLOR} and \texttt{SRC1\_ALPHA}).

If the second color input to the blender is not written in the shader, or if no output is bound to the second input of a blender, the result of the blending operation is not defined.

**Blend Color**

The constant color $C_c$ to be used in blending is specified with the command

```c
void BlendColor( float red, float green, float blue, float alpha );
```

The constant color can be used in both the source and destination blending functions. If destination framebuffer components use an unsigned normalized fixed-point representation, the constant color components are clamped to the range $[0, 1]$ when computing blend factors.

**Blending State**

The state required for blending, for each draw buffer, is two integers for the RGB and alpha blend equations, four integers indicating the source and destination RGB and alpha blending functions, and a bit indicating whether blending is enabled or disabled. Additionally, four floating-point values to store the RGBA constant blend color are required.

For all draw buffers, the initial blend equations for RGB and alpha are both \texttt{FUNC\_ADD}, and the initial blending functions are \texttt{ONE} for the source RGB and alpha functions and \texttt{ZERO} for the destination RGB and alpha functions. Initially, blending is disabled for all draw buffers. The initial constant blend color is $(R, G, B, A) = (0, 0, 0, 0)$.

The value of the blend enable for draw buffer $i$ can be queried by calling \texttt{IsEnabledi} with \texttt{target BLEND} and \texttt{index $i$}, and the values of the blend equations and functions can be queried by calling \texttt{GetIntegeri\_v} with the corresponding \texttt{target} as shown in table 6.21 and \texttt{index $i$}.

The value of the blend enable, or the blend equations and functions for draw buffer zero may also be queried by calling \texttt{IsEnabled}, or \texttt{GetInteger}, respectively, with the same symbolic constants but no \texttt{index} parameter.
4.1. PER-FRAGMENT OPERATIONS

Blending occurs once for each color buffer currently enabled for blending and for writing (section 4.2.1) using each buffer’s color for \( C_d \). If a color buffer has no \( A \) value, then \( A_d \) is taken to be 1.

4.1.8 sRGB Conversion

If \texttt{FRAMEBUFFER\_SRGB} is enabled and the value of \texttt{FRAMEBUFFER\_ATTACHMENT\_COLOR\_ENCODING} for the framebuffer attachment corresponding to the destination buffer is \texttt{SRGB} \(^2\) (see section 6.1.3), the \( R \), \( G \), and \( B \) values after blending are converted into the non-linear sRGB color space by computing

\[
 c_s = \begin{cases} 
 0.0, & c_l \leq 0 \\
 12.92c_l, & 0 < c_l < 0.0031308 \\
 1.055c_l^{0.41666} - 0.055, & 0.0031308 \leq c_l < 1 \\
 1.0, & c_l \geq 1 
\end{cases} \tag{4.1}
\]

where \( c_l \) is the \( R \), \( G \), or \( B \) element and \( c_s \) is the result (effectively converted into an sRGB color space).

If \texttt{FRAMEBUFFER\_SRGB} is disabled or the value of \texttt{FRAMEBUFFER\_ATTACHMENT\_COLOR\_ENCODING} is not \texttt{SRGB}, then

\[
 c_s = c_l. 
\]

The resulting \( c_s \) values for \( R \), \( G \), and \( B \), and the unmodified \( A \) form a new RGBA color value. If the color buffer is fixed-point, each component is clamped to the range \([0, 1]\) and then converted to a fixed-point value using equation 2.3. The resulting four values are sent to the subsequent dithering operation.

4.1.9 Dithering

Dithering selects between two representable color values or indices. A representable value is a value that has an exact representation in the color buffer. Dithering selects, for each color component, either the largest representable color value (for that particular color component) that is less than or equal to the incoming color component value, \( c \), or the smallest representable color value that is greater than or equal to \( c \). The selection may depend on the \( x_w \) and \( y_w \) coordinates of the pixel, as well as on the exact value of \( c \). If one of the two values does not exist, then the selection defaults to the other value.

\(^2\) Note that only unsigned normalized fixed-point color buffers may be SRGB-encoded. Signed normalized fixed-point + SRGB encoding is not defined.
Many dithering selection algorithms are possible, but an individual selection must depend only on the incoming component value and the fragment’s \(x\) and \(y\) window coordinates. If dithering is disabled, then one of the two values above is selected, in an implementation-dependent manner that must not depend on the \(x_w\) and \(y_w\) coordinates of the pixel.

Dithering is enabled with \texttt{Enable} and disabled with \texttt{Disable} using the symbolic constant \texttt{DITHER}. The state required is thus a single bit. Initially, dithering is enabled.

### 4.1.10 Logical Operation

Finally, a logical operation is applied between the incoming fragment’s color values and the color values stored at the corresponding location in the framebuffer. The result replaces the values in the framebuffer at the fragment’s \((x_w, y_w)\) coordinates. If the selected draw buffers refer to the same framebuffer-attachable image more than once, then the values stored in that image are undefined.

The logical operation on color values is enabled or disabled with \texttt{Enable} or \texttt{Disable} using the symbolic constant \texttt{COLOR_LOGIC_OP}. If the logical operation is enabled for color values, it is as if blending were disabled, regardless of the value of \texttt{BLEND}. If multiple fragment colors are being written to multiple buffers (see section 4.2.1), the logical operation is computed and applied separately for each fragment color and the corresponding buffer.

Logical operation has no effect on a floating-point destination color buffer, or when \texttt{FRAMEBUFFER_SRGB} is enabled and the value of \texttt{FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING} for the framebuffer attachment corresponding to the destination buffer is \texttt{SRGB} (see section 6.1.3). However, if logical operation is enabled, blending is still disabled.

The logical operation is selected by

```c
void LogicOp(enum op);
```

\texttt{op} is a symbolic constant; the possible constants and corresponding operations are enumerated in table 4.3. In this table, \(s\) is the value of the incoming fragment and \(d\) is the value stored in the framebuffer. The numeric values assigned to the symbolic constants are the same as those assigned to the corresponding symbolic values in the X window system.

Logical operations are performed independently for each red, green, blue, and alpha value of each color buffer that is selected for writing. The required state is an integer indicating the logical operation, and a bit indicating whether the logical operation is enabled or disabled. The initial state is for the logic operation to be given by \texttt{COPY}, and to be disabled.
4.1. PER-FRAGMENT OPERATIONS

<table>
<thead>
<tr>
<th>Argument value</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLEAR</td>
<td>0</td>
</tr>
<tr>
<td>AND</td>
<td>( s \land d )</td>
</tr>
<tr>
<td>AND.Reverse</td>
<td>( s \land \neg d )</td>
</tr>
<tr>
<td>COPY</td>
<td>( s )</td>
</tr>
<tr>
<td>AND.Inverted</td>
<td>( \neg s \land d )</td>
</tr>
<tr>
<td>NOOP</td>
<td>( d )</td>
</tr>
<tr>
<td>XOR</td>
<td>( s \oplus d )</td>
</tr>
<tr>
<td>OR</td>
<td>( s \lor d )</td>
</tr>
<tr>
<td>NOR</td>
<td>( \neg(s \lor d) )</td>
</tr>
<tr>
<td>EQUIV</td>
<td>( \neg(s \oplus d) )</td>
</tr>
<tr>
<td>INVERT</td>
<td>( \neg d )</td>
</tr>
<tr>
<td>OR.Reverse</td>
<td>( s \lor \neg d )</td>
</tr>
<tr>
<td>COPY.Inverted</td>
<td>( \neg s )</td>
</tr>
<tr>
<td>OR.Inverted</td>
<td>( \neg s \lor d )</td>
</tr>
<tr>
<td>NAND</td>
<td>( \neg(s \land d) )</td>
</tr>
<tr>
<td>SET</td>
<td>all 1’s</td>
</tr>
</tbody>
</table>

Table 4.3: Arguments to LogicOp and their corresponding operations.

4.1.11 Additional Multisample Fragment Operations

If the DrawBuffer mode is NONE, no change is made to any multisample or color buffer. Otherwise, fragment processing is as described below.

If MULTISAMPLE is enabled, and the value of SAMPLE_BUFFERS is one, the stencil test, depth test, blending, dithering, and logical operations are performed for each pixel sample, rather than just once for each fragment. Failure of the stencil or depth test results in termination of the processing of that sample, rather than discarding of the fragment. All operations are performed on the color, depth, and stencil values stored in the multisample renderbuffer attachments if a draw framebuffer object is bound, or otherwise in the multisample buffer of the default framebuffer. The contents of the color buffers are not modified at this point.

Stencil, depth, blending, dithering, and logical operations are performed for a pixel sample only if that sample’s fragment coverage bit is a value of 1. If the corresponding coverage bit is 0, no operations are performed for that sample.

If MULTISAMPLE is disabled, and the value of SAMPLE_BUFFERS is one, the fragment may be treated exactly as described above, with optimization possible because the fragment coverage must be set to full coverage. Further optimization is allowed, however. An implementation may choose to identify a centermost sam-
ple, and to perform stencil and depth tests on only that sample. Regardless of the outcome of the stencil test, all multisample buffer stencil sample values are set to the appropriate new stencil value. If the depth test passes, all multisample buffer depth sample values are set to the depth of the fragment’s centermost sample’s depth value, and all multisample buffer color sample values are set to the color value of the incoming fragment. Otherwise, no change is made to any multisample buffer color or depth value.

If a draw framebuffer object is not bound, after all operations have been completed on the multisample buffer, the sample values for each color in the multisample buffer are combined to produce a single color value, and that value is written into the corresponding color buffers selected by DrawBuffer or DrawBuffers. An implementation may defer the writing of the color buffers until a later time, but the state of the framebuffer must behave as if the color buffers were updated as each fragment was processed. The method of combination is not specified. If the framebuffer contains sRGB values, then it is recommended that the an average of sample values is computed in a linearized space, as for blending (see section 4.1.7). Otherwise, a simple average computed independently for each color component is recommended.

4.2 Whole Framebuffer Operations

The preceding sections described the operations that occur as individual fragments are sent to the framebuffer. This section describes operations that control or affect the whole framebuffer.

4.2.1 Selecting Buffers for Writing

The first such operation is controlling the color buffers into which each of the fragment color values is written. This is accomplished with either DrawBuffer or DrawBuffers.

The command

```c
void DrawBuffer(enum buf);
```

defines the set of color buffers to which fragment color zero is written. buf must be one of the values from tables 4.4 or 4.5. In addition, acceptable values for buf depend on whether the GL is using the default framebuffer (i.e., DRAW_FRAMEBUFFER_BINDING is zero), or a framebuffer object (i.e., DRAW_FRAMEBUFFER_BINDING is non-zero). In the initial state, the GL is bound to
4.2. WHOLE FRAMEBUFFER OPERATIONS

the default framebuffer. For more information about framebuffer objects, see section 4.4.

If the GL is bound to the default framebuffer, then buf must be one of the values listed in table 4.4, which summarizes the constants and the buffers they indicate. In this case, buf is a symbolic constant specifying zero, one, two, or four buffers for writing. These constants refer to the four potentially visible buffers (front left, front right, back left, and back right). Arguments that omit reference to LEFT or RIGHT refer to both left and right buffers. Arguments that omit reference to FRONT or BACK refer to both front and back buffers.

If the GL is bound to the default framebuffer and DrawBuffer is supplied with a constant (other than NONE) that does not indicate any of the color buffers allocated to the GL context, the error INVALID_OPERATION results.

If the GL is bound to a draw framebuffer object, buf must be one of the values listed in table 4.5, which summarizes the constants and the buffers they indicate. In this case, buf is a symbolic constant specifying a single color buffer for writing. Specifying COLOR_ATTACHMENTi enables drawing only to the image attached to the framebuffer at COLOR_ATTACHMENTi. Each COLOR_ATTACHMENTi adheres to COLOR_ATTACHMENTi = COLOR_ATTACHMENT0 + i. The initial value of DRAW_BUFFER for framebuffer objects is COLOR_ATTACHMENT0.

If the GL is bound to a draw framebuffer object and buf is one of the constants from table 4.4 (other than NONE), then the error INVALID_OPERATION results. If buf is COLOR_ATTACHMENTm and m is greater than or equal to the value of MAX_COLOR_ATTACHMENTS, then the error INVALID_VALUE results.

If DrawBuffer is supplied with a constant that is legal for neither the default framebuffer nor a framebuffer object, then the error INVALID_ENUM results.

DrawBuffer will set the draw buffer for fragment colors other than zero to NONE.

The command

\[
\text{void DrawBuffers( sizei } n, \text{ const enum } *\text{bufs } \);
\]
defines the draw buffers to which all fragment colors are written. n specifies the number of buffers in bufs. bufs is a pointer to an array of symbolic constants specifying the buffer to which each fragment color is written.

Each buffer listed in bufs must be one of the values from tables 4.5 or 4.6. Otherwise, an INVALID_ENUM error is generated. Further, acceptable values for the constants in bufs depend on whether the GL is using the default framebuffer (i.e., DRAW_FRAMEBUFFER_BINDING is zero), or a framebuffer object (i.e., DRAW_FRAMEBUFFER_BINDING is non-zero). For more information about framebuffer objects, see section 4.4.
### 4.2. WHOLE FRAMEBUFFER OPERATIONS

#### Table 4.4: Arguments to DrawBuffer and ReadBuffer when the context is bound to a default framebuffer, and the buffers they indicate.

<table>
<thead>
<tr>
<th>Symbolic Constant</th>
<th>Front Left</th>
<th>Front Right</th>
<th>Back Left</th>
<th>Back Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FRONT_LEFT</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FRONT_RIGHT</td>
<td></td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BACK_LEFT</td>
<td></td>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>BACK_RIGHT</td>
<td></td>
<td></td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>FRONT</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BACK</td>
<td></td>
<td></td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>LEFT</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RIGHT</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FRONT_AND_BACK</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

#### Table 4.5: Arguments to DrawBuffer(s) and ReadBuffer when the context is bound to a framebuffer object, and the buffers they indicate. $i$ in COLOR_ATTACHMENT$i$ may range from zero to the value of MAX_COLOR_ATTACHMENTS - 1.

<table>
<thead>
<tr>
<th>Symbolic Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>No buffer</td>
</tr>
<tr>
<td>COLOR_ATTACHMENT$i$ (see caption)</td>
<td>Output fragment color to image attached at color attachment point $i$</td>
</tr>
</tbody>
</table>

#### Table 4.6: Arguments to DrawBuffers when the context is bound to the default framebuffer, and the buffers they indicate.

<table>
<thead>
<tr>
<th>Symbolic Constant</th>
<th>Front Left</th>
<th>Front Right</th>
<th>Back Left</th>
<th>Back Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FRONT_LEFT</td>
<td>•</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FRONT_RIGHT</td>
<td></td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BACK_LEFT</td>
<td></td>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>BACK_RIGHT</td>
<td></td>
<td></td>
<td></td>
<td>•</td>
</tr>
</tbody>
</table>

OpenGL 4.2 (Core Profile) - April 27, 2012
If the GL is bound to the default framebuffer, then each of the constants must be one of the values listed in table 4.6.

If the GL is bound to a draw framebuffer object, then each of the constants must be one of the values listed in table 4.5.

In both cases, the draw buffers being defined correspond in order to the respective fragment colors. The draw buffer for fragment colors beyond $n$ is set to NONE.

The maximum number of draw buffers is implementation-dependent. The number of draw buffers supported can be queried by calling `GetInteger` with the symbolic constant `MAX_DRAW_BUFFERS`. An `INVALID_VALUE` error is generated if $n$ is greater than `MAX_DRAW_BUFFERS`.

Except for NONE, a buffer may not appear more than once in the array pointed to by `bufs`. Specifying a buffer more than once will result in the error `INVALID_OPERATION`.

If a fragment shader writes to a user-defined output variable, DrawBuffers specifies a set of draw buffers into which each of the multiple output colors defined by these variables are separately written. If a fragment shader writes to no user-defined output variables, the values of the fragment colors following shader execution are undefined, and may differ for each fragment color.

For both the default framebuffer and framebuffer objects, the constants FRONT, BACK, LEFT, RIGHT, and FRONT_AND_BACK are not valid in the `bufs` array passed to DrawBuffers, and will result in the error `INVALID_ENUM`. This restriction is because these constants may themselves refer to multiple buffers, as shown in table 4.4.

If the GL is bound to the default framebuffer and DrawBuffers is supplied with a constant (other than NONE) that does not indicate any of the color buffers allocated to the GL context by the window system, the error `INVALID_OPERATION` will be generated.

If the GL is bound to a draw framebuffer object and DrawBuffers is supplied with a constant from table 4.6, or `COLOR_ATTACHMENTm` where $m$ is greater than or equal to the value of `MAX_COLOR_ATTACHMENTS`, then the error `INVALID_OPERATION` results.

Indicating a buffer or buffers using `DrawBuffer` or DrawBuffers causes subsequent pixel color value writes to affect the indicated buffers. If the GL is bound to a draw framebuffer object and a draw buffer selects an attachment that has no image attached, then that fragment color is not written to any buffer.

Specifying NONE as the draw buffer for a fragment color will inhibit that fragment color from being written to any buffer.

Monoscopic contexts include only left buffers, while stereoscopic contexts include both left and right buffers. Likewise, single buffered contexts include only
4.2. WHOLE FRAMEBUFFER OPERATIONS

Front buffers, while double buffered contexts include both front and back buffers. The type of context is selected at GL initialization.

The state required to handle color buffer selection for each framebuffer is an integer for each supported fragment color. For the default framebuffer, in the initial state the draw buffer for fragment color zero is BACK if there is a back buffer; FRONT if there is no back buffer; and NONE if no default framebuffer is associated with the context. For framebuffer objects, in the initial state the draw buffer for fragment color zero is COLOR_ATTACHMENT0. For both the default framebuffer and framebuffer objects, the initial state of draw buffers for fragment colors other than zero is NONE.

The value of the draw buffer selected for fragment color \(i\) can be queried by calling \texttt{GetIntegerv} with the symbolic constant \texttt{DRAW_BUFFER} \(i\). \texttt{DRAW_BUFFER} is equivalent to \texttt{DRAW_BUFFER0}.

4.2.2 Fine Control of Buffer Updates

Writing of bits to each of the logical framebuffers after all per-fragment operations have been performed may be masked. The commands

\begin{verbatim}
void ColorMask( boolean r, boolean g, boolean b, boolean a );
void ColorMaski( uint buf, boolean r, boolean g, boolean b, boolean a );
\end{verbatim}

control writes to the active draw buffers.

\texttt{ColorMask} and \texttt{ColorMaski} are used to mask the writing of R, G, B and A values to the draw buffer or buffers. \texttt{ColorMaski} sets the mask for a particular draw buffer. The mask for \texttt{DRAW_BUFFER} \(i\) is modified by passing \(i\) as the parameter \texttt{buf}. \(r, g, b,\) and \(a\) indicate whether R, G, B, or A values, respectively, are written or not (a value of \texttt{TRUE} means that the corresponding value is written). The mask specified by \(r, g, b,\) and \(a\) is applied to the color buffer associated with \texttt{DRAW_BUFFER} \(i\). If \texttt{DRAW_BUFFER} \(i\) is one of \texttt{FRONT}, \texttt{BACK}, \texttt{LEFT}, \texttt{RIGHT}, or \texttt{FRONT_AND_BACK} (specifying multiple color buffers) then the mask is applied to all of the buffers.

\texttt{ColorMask} sets the mask for all draw buffers to the same values as specified by \(r, g, b,\) and \(a\).

An \texttt{INVALID_VALUE} error is generated if \texttt{index} is greater than the value of \texttt{MAX_DRAW_BUFFERS} minus one.

In the initial state, all color values are enabled for writing for all draw buffers.

The value of the color writemask for draw buffer \(i\) can be queried by calling \texttt{GetBooleani} with \texttt{target} \texttt{COLOR_WRITEMASK} and \texttt{index} \(i\). The value of the color...
writemask for draw buffer zero may also be queried by calling `GetBooleanv` with the symbolic constant `COLOR_WRITEMASK`.

The depth buffer can be enabled or disabled for writing \( z_w \) values using

```c
void DepthMask(boolean mask);
```

If `mask` is non-zero, the depth buffer is enabled for writing; otherwise, it is disabled. In the initial state, the depth buffer is enabled for writing.

The commands

```c
void StencilMask(uint mask);
void StencilMaskSeparate(enum face, uint mask);
```

control the writing of particular bits into the stencil planes.

The least significant \( s \) bits of `mask`, where \( s \) is the number of bits in the stencil buffer, specify an integer mask. Where a 1 appears in this mask, the corresponding bit in the stencil buffer is written; where a 0 appears, the bit is not written. The `face` parameter of `StencilMaskSeparate` can be `FRONT`, `BACK`, or `FRONT_AND_BACK` and indicates whether the front or back stencil mask state is affected. `StencilMask` sets both front and back stencil mask state to identical values.

Fragments generated by front-facing primitives use the front mask and fragments generated by back-facing primitives use the back mask (see section 4.1.4). The clear operation always uses the front stencil write mask when clearing the stencil buffer.

The state required for the various masking operations is two integers for the front and back stencil values, and a bit for depth values. A set of four bits is also required indicating which color components of an RGBA value should be written. In the initial state, the integer masks are all ones, as are the bits controlling depth value and RGBA component writing.

**Fine Control of Multisample Buffer Updates**

When the value of `SAMPLE_BUFFERS` is one, `ColorMask`, `DepthMask`, and `StencilMask` or `StencilMaskSeparate` control the modification of values in the multisample buffer. The color mask has no effect on modifications to the color buffers. If the color mask is entirely disabled, the color sample values must still be combined (as described above) and the result used to replace the color values of the buffers enabled by `DrawBuffer`.
4.2. WHOLE FRAMEBUFFER OPERATIONS

4.2.3 Clearing the Buffers

The GL provides a means for setting portions of every pixel in a particular buffer to the same value. The argument to

```
void Clear(bitfield buf);
```

is zero or the bitwise OR of one or more values indicating which buffers are to be cleared. The values are COLOR_BUFFER_BIT, DEPTH_BUFFER_BIT, and STENCIL_BUFFER_BIT, indicating the buffers currently enabled for color writing, the depth buffer, and the stencil buffer (see below), respectively. The value to which each buffer is cleared depends on the setting of the clear value for that buffer. If `buf` is zero, no buffers are cleared. If `buf` contains any bits other than COLOR BUFFER_BIT, DEPTH_BUFFER_BIT, or STENCIL_BUFFER_BIT, then the error INVALID_VALUE is generated.

```
void ClearColor( float r, float g, float b, float a );
```

sets the clear value for fixed-point and floating-point color buffers. The specified components are stored as floating-point values.

The command

```
void ClearDepth( double d );
void ClearDepthf( float d );
```

sets the depth value used when clearing the depth buffer. When clearing a fixed-point depth buffer, `d` is clamped to the range \([0, 1]\) and converted to fixed-point according to the rules for a window \(z\) value given in section 2.14.1. No conversion is applied when clearing a floating-point depth buffer.

The command

```
void ClearStencil( int s );
```

takes a single integer argument that is the value to which to clear the stencil buffer. `s` is masked to the number of bitplanes in the stencil buffer.

When Clear is called, the only per-fragment operations that are applied (if enabled) are the pixel ownership test, the scissor test, sRGB conversion (see section 4.1.8), and dithering. The masking operations described in section 4.2.2 are also applied. If a buffer is not present, then a Clear directed at that buffer has no effect.

Unsigned normalized fixed-point and signed normalized fixed-point RGBA color buffers are cleared to color values derived by clamping each component of the
clear color to the range \([0, 1]\) or \([-1, 1]\) respectively, then converting the (possibly sRGB converted and/or dithered) color to fixed-point using equations 2.3 or 2.4, respectively. The result of clearing integer color buffers is undefined.

The state required for clearing is a clear value for each of the color buffer, the depth buffer, and the stencil buffer. Initially, the RGBA color clear value is \((0, 0, 0, 0)\), the depth buffer clear value is 1.0, and the stencil buffer clear index is 0.

Individual buffers of the currently bound draw framebuffer may be cleared with the command

```c
void ClearBuffer{if ui}(enum buffer, int drawbuffer, const T *value);
```

where `buffer` and `drawbuffer` identify a buffer to clear, and `value` specifies the value or values to clear it to.

If `buffer` is `COLOR`, a particular draw buffer `DRAW_BUFFER[i]` is specified by passing `i` as the parameter `drawbuffer`, and `value` points to a four-element vector specifying the R, G, B, and A color to clear that draw buffer to. If the draw buffer is one of `FRONT`, `BACK`, `LEFT`, `RIGHT`, or `FRONT_AND_BACK`, identifying multiple buffers, each selected buffer is cleared to the same value. The `ClearBufferfv`, `ClearBufferiv`, and `ClearBufferuiv` commands should be used to clear fixed- and floating-point, signed integer, and unsigned integer color buffers respectively. Clamping and conversion for fixed-point color buffers are performed in the same fashion as `ClearColor`.

If `buffer` is `DEPTH`, `drawbuffer` must be zero, and `value` points to the single depth value to clear the depth buffer to. Clamping and type conversion for fixed-point depth buffers are performed in the same fashion as `ClearDepth`. Only `ClearBufferfv` should be used to clear depth buffers.

If `buffer` is `STENCIL`, `drawbuffer` must be zero, and `value` points to the single stencil value to clear the stencil buffer to. Masking is performed in the same fashion as `ClearStencil`. Only `ClearBufferiv` should be used to clear stencil buffers.

The command

```c
void ClearBufferfi(enum buffer, int drawbuffer, float depth, int stencil);
```

clears both depth and stencil buffers of the currently bound draw framebuffer. `buffer` must be `DEPTH_STENCIL` and `drawbuffer` must be zero. `depth` and `stencil` are the values to clear the depth and stencil buffers to, respectively. Clamping and type conversion of `depth` for fixed-point depth buffers is performed in the same
fashion as **ClearDepth**. Masking of **stencil** for stencil buffers is performed in the
same fashion as **ClearStencil**. **ClearBufferfi** is equivalent to clearing the depth
and stencil buffers separately, but may be faster when a buffer of internal format
**DEPTH_STENCIL** is being cleared.

The result of **ClearBuffer** is undefined if no conversion between the type of
the specified **value** and the type of the buffer being cleared is defined (for example,
if **ClearBufferiv** is called for a fixed- or floating-point buffer, or if **ClearBufferfv**
is called for a signed or unsigned integer buffer). This is not an error.

When **ClearBuffer** is called, the same per-fragment and masking operations
defined for **Clear** are applied.

**Errors**

**ClearBuffer{u}** generates an **INVALID_ENUM** error if **buffer** is not **COLOR**,  
**DEPTH**, or **STENCIL**. **ClearBufferfi** generates an **INVALID_ENUM** error if **buffer** is not **DEPTH_STENCIL**.

**ClearBuffer** generates an **INVALID_VALUE** error if **buffer** is **COLOR** and draw-
**buffer** is less than zero, or greater than the value of **MAX_DRAW_BUFFERS** minus
one; or if **buffer** is **DEPTH**, **STENCIL**, or **DEPTH_STENCIL** and **drawbuffer** is not
zero.

**Clearing the Multisample Buffer**

The color samples of the multisample buffer are cleared when one or more color
buffers are cleared, as specified by the **Clear** mask bit **COLOR_BUFFER_BIT** and
the **DrawBuffer** mode. If the **DrawBuffer** mode is **NONE**, the color samples of
the multisample buffer cannot be cleared using **Clear**.

If the **Clear** mask bits **DEPTH_BUFFER_BIT** or **STENCIL_BUFFER_BIT** are
set, then the corresponding depth or stencil samples, respectively, are cleared.

The **ClearBuffer** commands also clear color, depth, or stencil samples of mul-
tisample buffers corresponding to the specified buffer.

Masking and scissoring affect clearing the multisample buffer in the same way
as they affect clearing the corresponding color, depth, and stencil buffers.

### 4.3 Reading and Copying Pixels

Pixels may be read from the framebuffer using **ReadPixels**. **BlitFramebuffer**
can be used to copy a block of pixels from one portion of the framebuffer to another.
4.3. READING AND COPYING PIXELS

The method for reading pixels from the framebuffer and placing them in pixel pack buffer or client memory is diagrammed in figure 4.2. We describe the stages of the pixel reading process in the order in which they occur.

Initially, zero is bound for the PIXEL_PACK_BUFFER, indicating that image read and query commands such as ReadPixels return pixel results into client memory pointer parameters. However, if a non-zero buffer object is bound as the current pixel pack buffer, then the pointer parameter is treated as an offset into the designated buffer object.

Pixels are read using

```c
void ReadPixels(int x, int y, size_t width, size_t height,
                enum format, enum type, void *data);
```

The arguments after `x` and `y` to `ReadPixels` are described in section 3.7.2. The pixel storage modes that apply to `ReadPixels` and other commands that query images (see section 6.1) are summarized in table 4.7.

If the current read buffer is neither floating-point nor integer, calling `GetIntegerv` with the symbolic constants IMPLEMENTATION_COLOR_READ_FORMAT and

![Figure 4.2. Operation of ReadPixels. Operations in dashed boxes are not performed for all data formats. Depth and stencil pixel paths are not shown.](image)
### 4.3. READING AND COPYING PIXELS

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Initial Value</th>
<th>Valid Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>PACK_SWAP_BYTES</td>
<td>boolean</td>
<td>FALSE</td>
<td>TRUE/FALSE</td>
</tr>
<tr>
<td>PACK_LSB_FIRST</td>
<td>boolean</td>
<td>FALSE</td>
<td>TRUE/FALSE</td>
</tr>
<tr>
<td>PACK_ROW_LENGTH</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>PACK_SKIP_ROWS</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>PACK_SKIP_PIXELS</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>PACK_ALIGNMENT</td>
<td>integer</td>
<td>4</td>
<td>1,2,4,8</td>
</tr>
<tr>
<td>PACK_IMAGE_HEIGHT</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>PACK_SKIP_IMAGES</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>PACK_COMPRESSED_BLOCK_WIDTH</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>PACK_COMPRESSED_BLOCK_HEIGHT</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>PACK_COMPRESSED_BLOCK_DEPTH</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>PACK_COMPRESSED_BLOCK_SIZE</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
</tbody>
</table>

Table 4.7: PixelStore parameters pertaining to ReadPixels, GetCompressedTexImage and GetTexImage.

IMPLEMENTATION_COLOR_READ_TYPE, will return RGBA and UNSIGNED_BYTE, respectively; otherwise it will generate an INVALID_OPERATION error.

ReadPixels generates an INVALID_OPERATION error if READ_FRAMEBUFFER_BINDING (see section 4.4) is non-zero, the read framebuffer is framebuffer complete, and the value of SAMPLE_BUFFERS for the read framebuffer is greater than zero.

**Obtaining Pixels from the Framebuffer**

If the *format* is DEPTH_COMPONENT, then values are obtained from the depth buffer. If there is no depth buffer, the error INVALID_OPERATION occurs.

If there is a multisample buffer (the value of SAMPLE_BUFFERS is one), then values are obtained from the depth samples in this buffer. It is recommended that the depth value of the centermost sample be used, though implementations may choose any function of the depth sample values at each pixel.

If the *format* is DEPTH_STENCIL, then values are taken from both the depth buffer and the stencil buffer. If there is no depth buffer or if there is no stencil buffer, then the error INVALID_OPERATION occurs. If the *type* parameter is not UNSIGNED_INT_24_8 or FLOAT_32 UNSIGNED_INT_24_8 REV, then the error INVALID_ENUM occurs.

If there is a multisample buffer, then values are obtained from the depth and...
stencil samples in this buffer. It is recommended that the depth and stencil values of
the centermost sample be used, though implementations may choose any function
of the depth and stencil sample values at each pixel.

If the format is STENCIL_INDEX, then values are taken from the stencil buffer;
again, if there is no stencil buffer, the error INVALID_OPERATION occurs.

If there is a multisample buffer, then values are obtained from the stencil sam-
ples in this buffer. It is recommended that the stencil value of the centermost sam-
ples be used, though implementations may choose any function of the stencil sample
values at each pixel.

For all other formats, the read buffer from which values are obtained is one of
the color buffers; the selection of color buffer is controlled with ReadBuffer.

The command

    void ReadBuffer( enum src );

takes a symbolic constant as argument. src must be one of the values from ta-
bles 4.4 or 4.5. Otherwise, an INVALID_ENUM error is generated. Further, the
acceptable values for src depend on whether the GL is using the default frame-
buffer (i.e., READ_FRAMEBUFFER_BINDING is zero), or a framebuffer object (i.e.,
READ_FRAMEBUFFER_BINDING is non-zero). For more information about frame-
buffer objects, see section 4.4.

If the object bound to READ_FRAMEBUFFER_BINDING is not framebuffer com-
plete (as defined in section 4.4.4), then ReadPixels generates the error INVALID_-
FRAMEBUFFER_OPERATION. If ReadBuffer is supplied with a constant that is nei-
ther legal for the default framebuffer, nor legal for a framebuffer object, then the
error INVALID_ENUM results.

When READ_FRAMEBUFFER_BINDING is zero, i.e. the default framebuffer, src
must be one of the values listed in table 4.4, including NONE, FRONT_AND_BACK,
FRONT, and LEFT refer to the front left buffer, BACK refers to the back left buffer,
and RIGHT refers to the front right buffer. The other constants correspond directly
to the buffers that they name. If the requested buffer is missing, then the error
INVALID_OPERATION is generated. For the default framebuffer, the initial setting
for ReadBuffer is FRONT if there is no back buffer; BACK if there is a back buffer;
and NONE if no default framebuffer is associated with the context.

When the GL is using a framebuffer object, src must be one of the values listed
in table 4.5, including NONE. In a manner analogous to how the DRAW_BUFFERS
state is handled, specifying COLOR_ATTACHMENTi enables reading from the image
attached to the framebuffer at COLOR_ATTACHMENTi. For framebuffer objects, the
initial setting for ReadBuffer is COLOR_ATTACHMENT0.
4.3. READING AND COPYING PIXELS

ReadPixels generates an INVALID_OPERATION error if it attempts to select a color buffer while READ_BUFFER is NONE, or if the GL is using a framebuffer object (the value of READ_FRAMEBUFFER_BINDING is non-zero) and the read buffer selects an attachment that has no image attached.

ReadPixels obtains values from the selected buffer from each pixel with lower left hand corner at \((x + i, y + j)\) for \(0 \leq i < \text{width}\) and \(0 \leq j < \text{height}\); this pixel is said to be the \(i\)th pixel in the \(j\)th row. If any of these pixels lies outside of the window allocated to the current GL context, or outside of the image attached to the currently bound read framebuffer object, then the values obtained for those pixels are undefined. When READ_FRAMEBUFFER_BINDING is zero, values are also undefined for individual pixels that are not owned by the current context. Otherwise, ReadPixels obtains values from the selected buffer, regardless of how those values were placed there.

If format is one of RED, GREEN, BLUE, RG, RGB, RGBA, BGR, or BGRA, then red, green, blue, and alpha values are obtained from the selected buffer at each pixel location.

If format is an integer format and the color buffer is not an integer format, or if the color buffer is an integer format and format is not an integer format, or if format is an integer format and type is FLOAT, HALF_FLOAT, UNSIGNED_INT_10F_11F_11F_REV, or UNSIGNED_INT_5_9_9_9_REV, an INVALID_OPERATION error is generated.

When READ_FRAMEBUFFER_BINDING is non-zero, the red, green, blue, and alpha values are obtained by first reading the internal component values of the corresponding value in the image attached to the selected logical buffer. Internal components are converted to an RGBA color by taking each R, G, B, and A component present according to the base internal format of the buffer (as shown in table 3.11). If G, B, or A values are not present in the internal format, they are taken to be zero, zero, and one respectively.

Conversion of RGBA values

This step applies only if format is not STENCIL_INDEX, DEPTH_COMPONENT, or DEPTH_STENCIL. The R, G, B, and A values form a group of elements.

For a signed or unsigned normalized fixed-point color buffer, each element is converted to floating-point using equations 2.2 or 2.1, respectively. For an integer or floating-point color buffer, the elements are unmodified.
Conversion of Depth values

This step applies only if format is DEPTH_COMPONENT or DEPTH_STENCIL and the depth buffer uses a fixed-point representation. An element is taken to be a fixed-point value in $[0, 1]$ with $m$ bits, where $m$ is the number of bits in the depth buffer (see section 2.14.1). No conversion is necessary if the depth buffer uses a floating-point representation.

Final Conversion

Read color clamping is controlled by calling

```c
void ClampColor( enum target, enum clamp );
```

with target set to CLAMP_READ_COLOR. If clamp is TRUE, read color clamping is enabled; if clamp is FALSE, read color clamping is disabled. If clamp is FIXED_ONLY, read color clamping is enabled if the selected read color buffer has fixed-point components.

For an integer RGBA color, each component is clamped to the representable range of type.

For a floating-point RGBA color, if type is FLOAT or HALF_FLOAT, each component is clamped to $[0, 1]$ if read color clamping is enabled. Then the appropriate conversion formula from table 4.8 is applied to the component.

If type is UNSIGNED_INT_10F_11F_11F_REV and format is RGB, each component is clamped to $[0, 1]$ if read color clamping is enabled. The returned data are then packed into a series of uint values. The red, green, and blue components are converted to unsigned 11-bit floating-point, unsigned 11-bit floating-point, and unsigned 10-bit floating-point as described in sections 2.1.1 and 2.1.1. The resulting red 11 bits, green 11 bits, and blue 10 bits are then packed as the 1st, 2nd, and 3rd components of the UNSIGNED_INT_10F_11F_11F_REV format as shown in table 3.8.

If type is UNSIGNED_INT_5_9_9_9_REV and format is RGB, each component is clamped to $[0, 1]$ if read color clamping is enabled. The returned data are then packed into a series of uint values. The red, green, and blue components are converted to red_s, green_s, blue_s, and expshared integers as described in section 3.9.3 when internalformat is RGB9_E5. The red_s, green_s, blue_s, and expshared are then packed as the 1st, 2nd, 3rd, and 4th components of the UNSIGNED_INT_5_9_9_9_REV format as shown in table 3.8.

For other types, and for a floating-point or unsigned normalized fixed-point color buffer, each component is clamped to $[0, 1]$ whether or not read color clamping is enabled. For a signed normalized fixed-point color buffer, each component
is clamped to \([0, 1]\) if read color clamping is enabled, or if \(type\) represents unsigned integer components; otherwise \(type\) represents signed integer components, and each component is clamped to \([-1, 1]\). Following clamping, the appropriate conversion formula from table 4.8 is applied to the component\(^3\).

For an index, if the \(type\) is not \(FLOAT\) or \(HALF_FLOAT\), final conversion consists of masking the index with the value given in table 4.9. If the \(type\) is \(FLOAT\) or \(HALF_FLOAT\), then the integer index is converted to a GL \(float\) or \(half\) data value.

**Placement in Pixel Pack Buffer or Client Memory**

If a pixel pack buffer is bound (as indicated by a non-zero value of \(PIXEL_PACK_-BUFFER_BINDING\)), \(data\) is an offset into the pixel pack buffer and the pixels are packed into the buffer relative to this offset; otherwise, \(data\) is a pointer to a block client memory and the pixels are packed into the client memory relative to the pointer. If a pixel pack buffer object is bound and packing the pixel data according to the pixel pack storage state would access memory beyond the size of the pixel pack buffer’s memory size, an \(INVALID\_OPERATION\) error results. If a pixel pack buffer object is bound and \(data\) is not evenly divisible by the number of basic machine units needed to store in memory the corresponding GL data type from table 3.2 for the \(type\) parameter, an \(INVALID\_OPERATION\) error results.

Groups of elements are placed in memory just as they are taken from memory when transferring pixel rectangles to the GL. That is, the \(i\)th group of the \(j\)th row (corresponding to the \(i\)th pixel in the \(j\)th row) is placed in memory just where the \(i\)th group of the \(j\)th row would be taken from when transferring pixels. See **Unpacking** under section 3.7.2. The only difference is that the storage mode parameters whose names begin with \(PACK\_\) are used instead of those whose names begin with \(UNPACK\_\). If the \(format\) is \(RED\), \(GREEN\), or \(BLUE\), only the corresponding single element is written. Likewise if the \(format\) is \(RG\), \(RGB\), or \(BGR\), only the corresponding two or three elements are written. Otherwise all the elements of each group are written.

### 4.3.2 Copying Pixels

The command

```c
void BlitFramebuffer(int srcX0, int srcY0, int srcX1, int srcY1, int dstX0, int dstY0, int dstX1, int dstY1, bitfield mask, enum filter);
```

\(^3\)OpenGL 4.2 changes the behavior of **ReadPixels** to allow readbacks from a signed normalized color buffer to a signed integer type without loss of information.
### 4.3. READING AND COPYING PIXELS

<table>
<thead>
<tr>
<th>type Parameter</th>
<th>GL Data Type</th>
<th>Component Conversion Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNSIGNED_BYTE</td>
<td>ubyte</td>
<td>Equation 2.3, $b = 8$</td>
</tr>
<tr>
<td>BYTE</td>
<td>byte</td>
<td>Equation 2.4, $b = 8$</td>
</tr>
<tr>
<td>UNSIGNED_SHORT</td>
<td>ushort</td>
<td>Equation 2.3, $b = 16$</td>
</tr>
<tr>
<td>SHORT</td>
<td>short</td>
<td>Equation 2.4, $b = 16$</td>
</tr>
<tr>
<td>UNSIGNED_INT</td>
<td>uint</td>
<td>Equation 2.3, $b = 32$</td>
</tr>
<tr>
<td>INT</td>
<td>int</td>
<td>Equation 2.4, $b = 32$</td>
</tr>
<tr>
<td>HALF_FLOAT</td>
<td>half</td>
<td>$c = f$</td>
</tr>
<tr>
<td>FLOAT</td>
<td>float</td>
<td>$c = f$</td>
</tr>
<tr>
<td>UNSIGNED_BYTE_3_3_2</td>
<td>ubyte</td>
<td>Equation 2.3, $b =$ bitfield width</td>
</tr>
<tr>
<td>UNSIGNED_BYTE_2_3_3_REV</td>
<td>ubyte</td>
<td>Equation 2.3, $b =$ bitfield width</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_6_5</td>
<td>ushort</td>
<td>Equation 2.3, $b =$ bitfield width</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_6_5_REV</td>
<td>ushort</td>
<td>Equation 2.3, $b =$ bitfield width</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_4_4_4_4</td>
<td>ushort</td>
<td>Equation 2.3, $b =$ bitfield width</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_4_4_4_4_REV</td>
<td>ushort</td>
<td>Equation 2.3, $b =$ bitfield width</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_5_5_1</td>
<td>ushort</td>
<td>Equation 2.3, $b =$ bitfield width</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_1_5_5_5_REV</td>
<td>ushort</td>
<td>Equation 2.3, $b =$ bitfield width</td>
</tr>
<tr>
<td>UNSIGNED_INT_8_8_8_8</td>
<td>uint</td>
<td>Equation 2.3, $b =$ bitfield width</td>
</tr>
<tr>
<td>UNSIGNED_INT_8_8_8_8_REV</td>
<td>uint</td>
<td>Equation 2.3, $b =$ bitfield width</td>
</tr>
<tr>
<td>UNSIGNED_INT_10_10_10_2</td>
<td>uint</td>
<td>Equation 2.3, $b =$ bitfield width</td>
</tr>
<tr>
<td>UNSIGNED_INT_2_10_10_10_REV</td>
<td>uint</td>
<td>Equation 2.3, $b =$ bitfield width</td>
</tr>
<tr>
<td>UNSIGNED_INT_24_8</td>
<td>uint</td>
<td>Equation 2.3, $b =$ bitfield width</td>
</tr>
<tr>
<td>UNSIGNED_INT_10F_11F_11F_REV</td>
<td>uint</td>
<td>Special</td>
</tr>
<tr>
<td>UNSIGNED_INT_5_9_9_9_REV</td>
<td>uint</td>
<td>Special</td>
</tr>
<tr>
<td>FLOAT_32_UNSIGNED_INT_24_8_REV</td>
<td>float</td>
<td>$c = f$ (depth only)</td>
</tr>
</tbody>
</table>

Table 4.8: Reversed component conversions, used when component data are being returned to client memory. Color and depth components are converted from the internal floating-point representation ($f$) to a datum of the specified GL data type ($c$). All arithmetic is done in the internal floating-point format. These conversions apply to component data returned by GL query commands and to components of pixel data returned to client memory. The equations remain the same even if the implemented ranges of the GL data types are greater than the minimum required ranges (see table 2.2).
### 4.3. READING AND COPYING PIXELS

<table>
<thead>
<tr>
<th>type Parameter</th>
<th>Index Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNSIGNED_BYTE</td>
<td>$2^8 - 1$</td>
</tr>
<tr>
<td>BYTE</td>
<td>$2^7 - 1$</td>
</tr>
<tr>
<td>UNSIGNED_SHORT</td>
<td>$2^{16} - 1$</td>
</tr>
<tr>
<td>SHORT</td>
<td>$2^{15} - 1$</td>
</tr>
<tr>
<td>UNSIGNED_INT</td>
<td>$2^{32} - 1$</td>
</tr>
<tr>
<td>INT</td>
<td>$2^{31} - 1$</td>
</tr>
<tr>
<td>UNSIGNED_INT_24_8</td>
<td>$2^8 - 1$</td>
</tr>
<tr>
<td>FLOAT_32_UNSIGNED_INT_24_8</td>
<td>$2^8 - 1$</td>
</tr>
</tbody>
</table>

Table 4.9: Index masks used by `ReadPixels`. Floating point data are not masked.

transfers a rectangle of pixel values from one region of the read framebuffer to another in the draw framebuffer.

`mask` is the bitwise OR of a number of values indicating which buffers are to be copied. The values are `COLOR_BUFFER_BIT`, `DEPTH_BUFFER_BIT`, and `STENCIL_BUFFER_BIT`, which are described in section 4.2.3. The pixels corresponding to these buffers are copied from the source rectangle bounded by the locations $(srcX_0, srcY_0)$ and $(srcX_1, srcY_1)$ to the destination rectangle bounded by the locations $(dstX_0, dstY_0)$ and $(dstX_1, dstY_1)$. The lower bounds of the rectangle are inclusive, while the upper bounds are exclusive.

When the color buffer is transferred, values are taken from the read buffer of the read framebuffer and written to each of the draw buffers of the draw framebuffer.

The actual region taken from the read framebuffer is limited to the intersection of the source buffers being transferred, which may include the color buffer selected by the read buffer, the depth buffer, and/or the stencil buffer depending on `mask`. The actual region written to the draw framebuffer is limited to the intersection of the destination buffers being written, which may include multiple draw buffers, the depth buffer, and/or the stencil buffer depending on `mask`. Whether or not the source or destination regions are altered due to these limits, the scaling and offset applied to pixels being transferred is performed as though no such limits were present.

If the source and destination rectangle dimensions do not match, the source image is stretched to fit the destination rectangle. `filter` must be `LINEAR` or `NEAREST`, and specifies the method of interpolation to be applied if the image is stretched. `LINEAR` filtering is allowed only for the color buffer; if `mask` includes `DEPTH_BUFFER_BIT` or `STENCIL_BUFFER_BIT`, and `filter` is not `NEAREST`, no copy is performed and an `INVALID_OPERATION` error is generated. If the source and

OpenGL 4.2 (Core Profile) - April 27, 2012
4.3. READING AND COPYING PIXELS

destination dimensions are identical, no filtering is applied. If either the source or
destination rectangle specifies a negative width or height (X1 < X0 or Y1 < Y0),
the image is reversed in the corresponding direction. If both the source and des-
tination rectangles specify a negative width or height for the same direction, no
reversal is performed. If a linear filter is selected and the rules of \textit{LINEAR} sam-
pling would require sampling outside the bounds of a source buffer, it is as though
\texttt{CLAMP\_TO\_EDGE} texture sampling were being performed. If a linear filter is se-
lected and sampling would be required outside the bounds of the specified source
region, but within the bounds of a source buffer, the implementation may choose
to clamp while sampling or not.

If the source and destination buffers are identical, and the source and destina-
tion rectangles overlap, the result of the blit operation is undefined.

When values are taken from the read buffer, no linearization is performed even
if the format of the buffer is \texttt{SRGB}.

When values are written to the draw buffers, blit operations bypass most of the
fragment pipeline. The only fragment operations which affect a blit are the pixel
ownership test, the scissor test, and \texttt{sRGB} conversion (see section 4.1.8). \texttt{Color},
depth, and stencil masks (see section 4.2.2) are ignored.

If the read framebuffer is layered (see section 4.4.7), pixel values are read from
layer zero. If the draw framebuffer is layered, pixel values are written to layer zero.
If both read and draw framebuffers are layered, the blit operation is still performed
only on layer zero.

If a buffer is specified in \textit{mask} and does not exist in both the read and draw
framebuffers, the corresponding bit is silently ignored.

If the color formats of the read and draw buffers do not match, and \textit{mask} in-
cludes \texttt{COLOR\_BUFFER\_BIT}, pixel groups are converted to match the destination
format. However, \texttt{colors} are clamped only if all draw color buffers have fixed-
point components. Format conversion is not supported for all data types, and an
\texttt{INVALID\_OPERATION} error is generated under any of the following conditions:

- The read buffer contains fixed-point or floating-point values and any draw
  buffer contains neither fixed-point nor floating-point values.
- The read buffer contains unsigned integer values and any draw buffer does
  not contain unsigned integer values.
- The read buffer contains signed integer values and any draw buffer does not
  contain signed integer values.

Calling \texttt{BlitFramebuffer} will result in an \texttt{INVALID\_FRAMEBUFFER\_-
OPERATION} error if the objects bound to \texttt{DRAW\_FRAMEBUFFER\_BINDING} and
\texttt{READ\_FRAMEBUFFER\_BINDING} are not framebuffer complete (section 4.4.4).
4.3. READING AND COPYING PIXELS

Calling BlitFramebuffer will result in an INVALID_OPERATION error if mask includes DEPTH_BUFFER_BIT or STENCIL_BUFFER_BIT, and the source and destination depth and stencil buffer formats do not match.

Calling BlitFramebuffer will result in an INVALID_OPERATION error if filter is LINEAR and read buffer contains integer data.

If SAMPLE_BUFFERS for the read framebuffer is greater than zero and SAMPLE_BUFFERS for the draw framebuffer is zero, the samples corresponding to each pixel location in the source are converted to a single sample before being written to the destination.

If SAMPLE_BUFFERS for the read framebuffer is zero and SAMPLE_BUFFERS for the draw framebuffer is greater than zero, the value of the source sample is replicated in each of the destination samples.

If SAMPLE_BUFFERS for either the read framebuffer or draw framebuffer is greater than zero, no copy is performed and an INVALID_OPERATION error is generated if the dimensions of the source and destination rectangles provided to BlitFramebuffer are not identical, or if the formats of the read and draw framebuffers are not identical.

If SAMPLE_BUFFERS for both the read and draw framebuffers are greater than zero, and the values of SAMPLES for the read and draw framebuffers are identical, the samples are copied without modification from the read framebuffer to the draw framebuffer. Otherwise, no copy is performed and an INVALID_OPERATION error is generated. Note that the samples in the draw buffer are not guaranteed to be at the same sample location as the read buffer, so rendering using this newly created buffer can potentially have geometry cracks or incorrect antialiasing. This may occur if the sizes of the framebuffers do not match or if the source and destination rectangles are not defined with the same \((X0,Y0)\) and \((X1,Y1)\) bounds.

4.3.3 Pixel Draw/Read State

The state required for pixel operations consists of the parameters that are set with PixelStore. This state has been summarized in table 3.1. Additional state includes an integer indicating the current setting of ReadBuffer and a three-valued integer controlling clamping during final conversion. For the default framebuffer, in the initial state the read buffer is BACK if there is a back buffer; FRONT if there is no back buffer; and NONE if no default framebuffer is associated with the context. The initial value of read color clamping is FIXED_ONLY. State set with PixelStore is GL client state.
4.4 Framebuffer Objects

As described in chapter 1 and section 2.1, the GL renders into (and reads values from) a framebuffer. There are two classes of framebuffers: window system-provided and application-created.

Initially, the GL uses the default framebuffer. The storage, dimensions, allocation, and format of the images attached to this framebuffer are managed entirely by the window system. Consequently, the state of the default framebuffer, including its images, can not be changed by the GL, nor can the default framebuffer be deleted by the GL.

The routines described in the following sections, however, can be used to create, destroy, and modify the state and attachments of framebuffer objects.

Framebuffer objects encapsulate the state of a framebuffer in a similar manner to the way texture objects encapsulate the state of a texture. In particular, a framebuffer object encapsulates state necessary to describe a collection of color, depth, and stencil logical buffers (other types of buffers are not allowed). For each logical buffer, a framebuffer-attachable image can be attached to the framebuffer to store the rendered output for that logical buffer. Examples of framebuffer-attachable images include texture images and renderbuffer images. Renderbuffers are described further in section 4.4.2.

By allowing the images of a renderbuffer to be attached to a framebuffer, the GL provides a mechanism to support off-screen rendering. Further, by allowing the images of a texture to be attached to a framebuffer, the GL provides a mechanism to support render to texture.

4.4.1 Binding and Managing Framebuffer Objects

The default framebuffer for rendering and readback operations is provided by the window system. In addition, named framebuffer objects can be created and operated upon. The name space for framebuffer objects is the unsigned integers, with zero reserved by the GL for the default framebuffer.

A framebuffer object is created by binding a name returned by GenFramebuffers (see below) to DRAW_FRAMEBUFFER or READ_FRAMEBUFFER. The binding is effected by calling

```c
void BindFramebuffer( enum target, uint framebuffer );
```

with target set to the desired framebuffer target and framebuffer set to the framebuffer object name. The resulting framebuffer object is a new state vector, comprising all the state and with the same initial values listed in table 6.24, as well
as one set of the state values listed in table 6.25 for each attachment point of the framebuffer, with the same initial values. There are the value of \texttt{MAX\_COLOR\_ATTACHMENTS} color attachment points, plus one each for the depth and stencil attachment points.

\textbf{BindFramebuffer} may also be used to bind an existing framebuffer object to \texttt{DRAW\_FRAMEBUFFER} and/or \texttt{READ\_FRAMEBUFFER}. If the bind is successful no change is made to the state of the newly bound framebuffer object, and any previous binding to \texttt{target} is broken.

\textbf{BindFramebuffer} fails and an \texttt{INVALID\_OPERATION} error is generated if \texttt{framebuffer} is not zero or a name returned from a previous call to \texttt{GenFramebuffers}, or if such a name has since been deleted with \texttt{DeleteFramebuffers}.

If a framebuffer object is bound to \texttt{DRAW\_FRAMEBUFFER} or \texttt{READ\_FRAMEBUFFER}, it becomes the target for rendering or readback operations, respectively, until it is deleted or another framebuffer object is bound to the corresponding bind point. Calling \textbf{BindFramebuffer} with \texttt{target} set to \texttt{FRAMEBUFFER} binds \texttt{framebuffer} to both the draw and read targets.

While a framebuffer object is bound, GL operations on the target to which it is bound affect the images attached to the bound framebuffer object, and queries of the target to which it is bound return state from the bound object. Queries of the values specified in tables 6.62 and 6.24 are derived from the framebuffer object bound to \texttt{DRAW\_FRAMEBUFFER}, with the exception of those marked as properties of the read framebuffer, which are derived from the framebuffer object bound to \texttt{READ\_FRAMEBUFFER}.

The initial state of \texttt{DRAW\_FRAMEBUFFER} and \texttt{READ\_FRAMEBUFFER} refers to the default framebuffer. In order that access to the default framebuffer is not lost, it is treated as a framebuffer object with the name of zero. The default framebuffer is therefore rendered to and read from while zero is bound to the corresponding targets. On some implementations, the properties of the default framebuffer can change over time (e.g., in response to window system events such as attaching the context to a new window system drawable.)

Framebuffer objects (those with a non-zero name) differ from the default framebuffer in a few important ways. First and foremost, unlike the default framebuffer, framebuffer objects have modifiable attachment points for each logical buffer in the framebuffer. Framebuffer-attachable images can be attached to and detached from these attachment points, which are described further in section 4.4.2. Also, the size and format of the images attached to framebuffer objects are controlled entirely within the GL interface, and are not affected by window system events, such as pixel format selection, window resizes, and display mode changes.

Additionally, when rendering to or reading from an application created-framebuffer object,
The pixel ownership test always succeeds. In other words, framebuffer objects own all of their pixels.

There are no visible color buffer bitplanes. This means there is no color buffer corresponding to the back, front, left, or right color bitplanes.

The only color buffer bitplanes are the ones defined by the framebuffer attachment points named `COLOR_ATTACHMENT0` through `COLOR_ATTACHMENTn`.

The only depth buffer bitplanes are the ones defined by the framebuffer attachment point `DEPTH_ATTACHMENT`.

The only stencil buffer bitplanes are the ones defined by the framebuffer attachment point `STENCIL_ATTACHMENT`.

If the attachment sizes are not all identical, rendering will be limited to the largest area that can fit in all of the attachments (an intersection of rectangles having a lower left of (0, 0) and an upper right of (width, height) for each attachment).

If the number of layers of each attachment are not all identical, rendering will be limited to the smallest number of layers of any attachment.

If the attachment sizes are not all identical, the values of pixels outside the common intersection area after rendering are undefined.

Framebuffer objects are deleted by calling

```c
void DeleteFramebuffers( sizei n, const uint *framebuffers );
```

`framebuffers` contains `n` names of framebuffer objects to be deleted. After a framebuffer object is deleted, it has no attachments, and its name is again unused. If a framebuffer that is currently bound to one or more of the targets `DRAW_FRAMEBUFFER` or `READ_FRAMEBUFFER` is deleted, it is as though `BindFramebuffer` had been executed with the corresponding `target` and `framebuffer` zero.

Unused names in `framebuffers` that have been marked as used for the purposes of `GenFramebuffers` are marked as unused again. Unused names in `framebuffers` are silently ignored, as is the value zero.

The command

```c
void GenFramebuffers( sizei n, uint *framebuffers );
```

OpenGL 4.2 (Core Profile) - April 27, 2012
returns \( n \) previously unused framebuffer object names in \( \text{framebuffers} \). These names are marked as used, for the purposes of \( \text{GenFramebuffers} \) only, but they acquire state and type only when they are first bound.

The names bound to the draw and read framebuffer bindings can be queried by calling \( \text{GetIntegerv} \) with the symbolic constants \( \text{DRAW_FRAMEBUFFER_BINDING} \) and \( \text{READ_FRAMEBUFFER_BINDING} \), respectively. \( \text{FRAMEBUFFER_BINDING} \) is equivalent to \( \text{DRAW_FRAMEBUFFER_BINDING} \).

4.4.2 Attaching Images to Framebuffer Objects

Framebuffer-attachable images may be attached to, and detached from, framebuffer objects. In contrast, the image attachments of the default framebuffer may not be changed by the GL.

A single framebuffer-attachable image may be attached to multiple framebuffer objects, potentially avoiding some data copies, and possibly decreasing memory consumption.

For each logical buffer, a framebuffer object stores a set of state which defines the logical buffer’s attachment point. The attachment point state contains enough information to identify the single image attached to the attachment point, or to indicate that no image is attached. The per-logical buffer attachment point state is listed in table 6.25.

There are several types of framebuffer-attachable images:

- The image of a renderbuffer object, which is always two-dimensional.
- A single level of a one-dimensional texture, which is treated as a two-dimensional image with a height of one.
- A single level of a two-dimensional, \( \text{two-dimensional multisample} \), or rectangle texture.
- A single face of a cube map texture level, which is treated as a two-dimensional image.
- A single layer of a one-or two-dimensional array texture, \( \text{two-dimensional multisample array texture} \), or three-dimensional texture, which is treated as a two-dimensional image.
- A single layer-face of a cube map array texture, which is treated as a two-dimensional image.
4.4. FRAMEBUFFER OBJECTS

Additionally, an entire level of a three-dimensional, cube map, cube map array, or one-or two-dimensional array texture can be attached to an attachment point. Such attachments are treated as an array of two-dimensional images, arranged in layers, and the corresponding attachment point is considered to be layered (also see section 4.4.7).

Renderbuffer Objects

A renderbuffer is a data storage object containing a single image of a renderable internal format. The commands described below allocate and delete a renderbuffer’s image, and attach a renderbuffer’s image to a framebuffer object.

The name space for renderbuffer objects is the unsigned integers, with zero reserved by the GL. A renderbuffer object is created by binding a name returned by GenRenderbuffers (see below) to RENDERBUFFER. The binding is effected by calling

```c
void BindRenderbuffer( enum target, uint renderbuffer );
```

with `target` set to RENDERBUFFER and `renderbuffer` set to the renderbuffer object name. If `renderbuffer` is not zero, then the resulting renderbuffer object is a new state vector, initialized with a zero-sized memory buffer, and comprising all the state and with the same initial values listed in table 6.27. Any previous binding to `target` is broken.

`BindRenderbuffer` may also be used to bind an existing renderbuffer object. If the bind is successful, no change is made to the state of the newly bound renderbuffer object, and any previous binding to `target` is broken.

While a renderbuffer object is bound, GL operations on the target to which it is bound affect the bound renderbuffer object, and queries of the target to which a renderbuffer object is bound return state from the bound object.

The name zero is reserved. A renderbuffer object cannot be created with the name zero. If `renderbuffer` is zero, then any previous binding to `target` is broken and the `target` binding is restored to the initial state.

In the initial state, the reserved name zero is bound to RENDERBUFFER. There is no renderbuffer object corresponding to the name zero, so client attempts to modify or query renderbuffer state for the target RENDERBUFFER while zero is bound will generate GL errors, as described in section 6.1.3.

The current RENDERBUFFER binding can be determined by calling `GetIntegerv` with the symbolic constant RENDERBUFFER_BINDING.

`BindRenderbuffer` fails and an INVALID_OPERATION error is generated if `renderbuffer` is not zero or a name returned from a previous call to GenRenderbuffers, or if such a name has since been deleted with DeleteRenderbuffers.
4.4. FRAMEBUFFER OBJECTS

Renderbuffer objects are deleted by calling

```c
void DeleteRenderbuffers( sizei n, const uint *renderbuffers );
```

where `renderbuffers` contains `n` names of renderbuffer objects to be deleted. After a renderbuffer object is deleted, it has no contents, and its name is again unused. If a renderbuffer that is currently bound to `RENDERBUFFER` is deleted, it is as though `BindRenderbuffer` had been executed with the `target` `RENDERBUFFER` and `name` of zero. Additionally, special care must be taken when deleting a renderbuffer if the image of the renderbuffer is attached to a framebuffer object (see section 4.4.2). Unused names in `renderbuffers` that have been marked as used for the purposes of `GenRenderbuffers` are marked as unused again. Unused names in `renderbuffers` are silently ignored, as is the value zero.

The command

```c
void GenRenderbuffers( sizei n, uint *renderbuffers );
```
returns `n` previously unused renderbuffer object names in `renderbuffers`. These names are marked as used, for the purposes of `GenRenderbuffers` only, but they acquire renderbuffer state only when they are first bound.

The command

```c
void RenderbufferStorageMultisample( enum target,
   sizei samples, enum internalformat, sizei width,
   sizei height );
```

establishes the data storage, format, dimensions, and number of samples of a renderbuffer object’s image. `target` must be `RENDERBUFFER`. `internalformat` must be color-renderable, depth-renderable, or stencil-renderable (as defined in section 4.4.4). `width` and `height` are the dimensions in pixels of the renderbuffer. If either `width` or `height` is greater than the value of `MAX_RENDERBUFFER_SIZE`, then the error `INVALID_VALUE` is generated. If `samples` is greater than the maximum number of samples supported for `internalformat`, then the error `INVALID_OPERATION` is generated (see `GetInternalformativ` in section 6.1.15). If the GL is unable to create a data store of the requested size, the error `OUT_OF_MEMORY` is generated.

Upon success, `RenderbufferStorageMultisample` deletes any existing data store for the renderbuffer image and the contents of the data store after calling `RenderbufferStorageMultisample` are undefined. `RENDERBUFFER_WIDTH`
### 4.4. FRAMEBUFFER OBJECTS

<table>
<thead>
<tr>
<th>Sized Internal Format</th>
<th>Base Internal Format</th>
<th>S bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>STENCIL_INDEX1</td>
<td>STENCIL_INDEX</td>
<td>1</td>
</tr>
<tr>
<td>STENCIL_INDEX4</td>
<td>STENCIL_INDEX</td>
<td>4</td>
</tr>
<tr>
<td>STENCIL_INDEX8</td>
<td>STENCIL_INDEX</td>
<td>8</td>
</tr>
<tr>
<td>STENCIL_INDEX16</td>
<td>STENCIL_INDEX</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 4.10: Correspondence of sized internal formats to base internal formats for formats that can be used only with renderbuffers.

is set to width, RENDERBUFFER_HEIGHT is set to height, and RENDERBUFFER_INTERNAL_FORMAT is set to internalformat.

If samples is zero, then RENDERBUFFER_SAMPLES is set to zero. Otherwise samples represents a request for a desired minimum number of samples. Since different implementations may support different sample counts for multisampled rendering, the actual number of samples allocated for the renderbuffer image is implementation-dependent. However, the resulting value for RENDERBUFFER_SAMPLES is guaranteed to be greater than or equal to samples and no more than the next larger sample count supported by the implementation.

A GL implementation may vary its allocation of internal component resolution based on any RenderbufferStorage parameter (except target), but the allocation and chosen internal format must not be a function of any other state and cannot be changed once they are established.

The command

```c
void RenderbufferStorage( enum target, enum internalformat,
 sizei width, sizei height );
```

is equivalent to calling RenderbufferStorageMultisample with samples equal to zero.

**Required Renderbuffer Formats**

Implementations are required to support the same internal formats for renderbuffers as the required formats for textures enumerated in section 3.9.3, with the exception of the color formats labelled “texture-only”. Requesting one of these internal formats for a renderbuffer will allocate at least the internal component sizes and exactly the component types shown for that format in tables 3.12- 3.13.

Implementations must support creation of renderbuffers in these required formats with up to the value of MAX_SAMPLES multisamples, with the exception
that the signed and unsigned integer formats are required only to support creation of renderbuffers with up to the value of `MAX_INTEGER_SAMPLES` multisamples, which must be at least one.

### Attaching Renderbuffer Images to a Framebuffer

A renderbuffer can be attached as one of the logical buffers of a currently bound framebuffer object by calling

```c
void FramebufferRenderbuffer(
    enum target,
    enum attachment, enum renderbuffertarget,
    uint renderbuffer);
```

*`target` must be `DRAW_FRAMEBUFFER`, `READ_FRAMEBUFFER`, or `FRAMEBUFFER`. `FRAMEBUFFER` is equivalent to `DRAW_FRAMEBUFFER`. An `INVALID_OPERATION` error is generated if the value of the corresponding binding is zero. `attachment` should be set to one of the attachment points of the framebuffer listed in table 4.11.*

*`renderbuffertarget` must be `RENDERBUFFER` and `renderbuffer` should be set to the name of the renderbuffer object to be attached to the framebuffer. `renderbuffer` must be either zero or the name of an existing renderbuffer object of type `renderbuffertarget`, otherwise an `INVALID_OPERATION` error is generated. If `renderbuffer` is zero, then the value of `renderbuffertarget` is ignored.*

If `renderbuffer` is not zero and if `FramebufferRenderbuffer` is successful, then the renderbuffer named `renderbuffer` will be used as the logical buffer identified by `attachment` of the framebuffer object currently bound to `target`. The value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` for the specified attachment point is set to `RENDERBUFFER` and the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` is set to `renderbuffer`. All other state values of the attachment point specified by `attachment` are set to their default values listed in table 6.25. No change is made to the state of the renderbuffer object and any previous attachment to the attachment logical buffer of the framebuffer object bound to framebuffer `target` is broken. If the attachment is not successful, then no change is made to the state of either the renderbuffer object or the framebuffer object.

Calling `FramebufferRenderbuffer` with the `renderbuffer` name zero will detach the image, if any, identified by `attachment`, in the framebuffer object currently bound to `target`. All state values of the attachment point specified by `attachment` in the object bound to `target` are set to their default values listed in table 6.25.

Setting `attachment` to the value `DEPTH_STENCIL_ATTACHMENT` is a special case causing both the depth and stencil attachments of the framebuffer object to be set to `renderbuffer`, which should have base internal format `DEPTH_STENCIL`.

OpenGL 4.2 (Core Profile) - April 27, 2012
If a renderbuffer object is deleted while its image is attached to one or more attachment points in a currently bound framebuffer object, then it is as if `FramebufferRenderbuffer` had been called, with a `renderbuffer` of zero, for each attachment point to which this image was attached in that framebuffer object. In other words, the renderbuffer image is first detached from all attachment points in that framebuffer object. Note that the renderbuffer image is specifically not detached from any non-bound framebuffer objects. Detaching the image from any non-bound framebuffer objects is the responsibility of the application.

<table>
<thead>
<tr>
<th>Name of attachment</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLOR_ATTACHMENT(i) (see caption)</td>
</tr>
<tr>
<td>DEPTH_ATTACHMENT</td>
</tr>
<tr>
<td>STENCIL_ATTACHMENT</td>
</tr>
<tr>
<td>DEPTH_STENCIL_ATTACHMENT</td>
</tr>
</tbody>
</table>

Table 4.11: Framebuffer attachment points. \(i\) in `COLOR_ATTACHMENT\(i\)` may range from zero to the value of `MAX_COLOR_ATTACHMENTS` - 1.

### Attaching Texture Images to a Framebuffer

The GL supports copying the rendered contents of the framebuffer into the images of a texture object through the use of the routines `CopyTexImage*` and `CopyTexSubImage*`. Additionally, the GL supports rendering directly into the images of a texture object.

To render directly into a texture image, a specified level of a texture object can be attached as one of the logical buffers of the currently bound framebuffer object by calling:

```c
void FramebufferTexture( enum target, enum attachment,
                        uint texture, int level );
```

`target` must be `DRAW_FRAMEBUFFER`, `READ_FRAMEBUFFER`, or `FRAMEBUFFER`. `FRAMEBUFFER` is equivalent to `DRAW_FRAMEBUFFER`. An `INVALID_OPERATION` error is generated if the value of the corresponding binding is zero. `attachment` must be one of the attachment points of the framebuffer listed in table 4.11.

If `texture` is non-zero, the specified mipmap `level` of the texture object named `texture` is attached to the framebuffer attachment point named by `attachment`. An `INVALID_VALUE` error is generated if `texture` is not the name of a texture object,
or if \texttt{level} is not a supported texture level number for \texttt{texture}. An \texttt{INVALID_OPERATION} error is generated if \texttt{texture} is the name of a buffer texture.

If \texttt{texture} is the name of a three-dimensional texture, cube map texture, one-or two-dimensional array texture, or two-dimensional multisample array texture, the texture level attached to the framebuffer attachment point is an array of images, and the framebuffer attachment is considered layered.

Additionally, a specified image from a texture object can be attached as one of the logical buffers of a currently bound framebuffer object by calling one of the following routines, depending on the type of the texture:

\begin{verbatim}
void FramebufferTexture1D( enum target, enum attachment,  
    enum textarget, uint texture, int level);
void FramebufferTexture2D( enum target, enum attachment,  
    enum textarget, uint texture, int level);
void FramebufferTexture3D( enum target, enum attachment,  
    enum textarget, uint texture, int level, int layer);
\end{verbatim}

In all three routines, \texttt{target} must be \texttt{DRAW_FRAMEBUFFER}, \texttt{READ_FRAMEBUFFER}, or \texttt{FRAMEBUFFER}. \texttt{FRAMEBUFFER} is equivalent to \texttt{DRAW_FRAMEBUFFER}. An \texttt{INVALID_OPERATION} error is generated if the value of the corresponding binding is zero. \texttt{attachment} must be one of the attachment points of the framebuffer listed in table 4.11.

If \texttt{texture} is not zero, then \texttt{texture} must either name an existing texture object with a target of \texttt{textarget}, or \texttt{texture} must name an existing cube map texture and \texttt{textarget} must be one of \texttt{TEXTURE_CUBE_MAP_POSITIVE_X}, \texttt{TEXTURE_CUBE_MAP_NEGATIVE_X}, \texttt{TEXTURE_CUBE_MAP_POSITIVE_Y}, \texttt{TEXTURE_CUBE_MAP_NEGATIVE_Y}, \texttt{TEXTURE_CUBE_MAP_POSITIVE_Z}, \texttt{TEXTURE_CUBE_MAP_NEGATIVE_Z}, \texttt{TEXTURE_CUBE_MAP_POSITIVE_W}, or \texttt{TEXTURE_CUBE_MAP_NEGATIVE_W}. Otherwise, an \texttt{INVALID_OPERATION} error is generated.

\texttt{level} specifies the mipmap level of the texture image to be attached to the framebuffer.

If \texttt{textarget} is \texttt{TEXTURE_RECTANGLE} or \texttt{TEXTURE_2D_MULTISAMPLE}, then \texttt{level} must be zero. If \texttt{textarget} is \texttt{TEXTURE_3D}, then \texttt{level} must be greater than or equal to zero and less than or equal to \(\log_2\) of the value of \texttt{MAX_3D_TEXTURE_SIZE}. If \texttt{textarget} is one of \texttt{TEXTURE_CUBE_MAP_POSITIVE_X}, \texttt{TEXTURE_CUBE_MAP_NEGATIVE_X}, \texttt{TEXTURE_CUBE_MAP_POSITIVE_Y}, \texttt{TEXTURE_CUBE_MAP_NEGATIVE_Y}, \texttt{TEXTURE_CUBE_MAP_POSITIVE_Z}, \texttt{TEXTURE_CUBE_MAP_NEGATIVE_Z}, \texttt{TEXTURE_CUBE_MAP_POSITIVE_W}, \texttt{TEXTURE_CUBE_MAP_NEGATIVE_W}, \texttt{TEXTURE_CUBE_MAP_POSITIVE_UV}, \texttt{TEXTURE_CUBE_MAP_NEGATIVE_UV}, \texttt{TEXTURE_CUBE_MAP_POSITIVE_VW}, or \texttt{TEXTURE_CUBE_MAP_NEGATIVE_VW}, then \texttt{level} must be greater than or equal to zero and less than or equal to \(\log_2\) of the value of \texttt{MAX_CUBE_MAP_TEXTURE_SIZE}. For all other values of \texttt{textarget}, \texttt{level} must be greater than or equal to
zero and no larger than $\log_2$ of the value of MAX_TEXTURE_SIZE. Otherwise, an INVALID_VALUE error is generated.

layer specifies the layer of a two-dimensional image within a three-dimensional texture. An INVALID_VALUE error is generated if layer is larger than the value of MAX_3D_TEXTURE_SIZE minus one.

For FramebufferTexture1D, if texture is not zero, then texture must be TEXTURE_1D.

For FramebufferTexture2D, if texture is not zero, then texture must be one of TEXTURE_2D, TEXTURE_RECTANGLE, TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Z, or TEXTURE_2D_MULTISAMPLE.

For FramebufferTexture3D, if texture is not zero, then texture must be TEXTURE_3D.

The command

```c
void FramebufferTextureLayer( enum target,
   enum attachment, uint texture, int level, int layer );
```

operates identically to FramebufferTexture3D, except that it attaches a single layer of a three-dimensional, one-or two-dimensional array, cube map array, or two-dimensional multisample array texture level.

layer specifies the layer of a two-dimensional image within texture except for cube map array textures, where

layer mod 6

indicates a face of a cube map slice within the cube map array. The order of the faces is as described for cube map array targets to TexImage3D in section 3.9.3. An INVALID_VALUE error is generated if layer is larger than the value of MAX_3D_TEXTURE_SIZE minus one (for three-dimensional textures) or larger than the value of MAX_ARRAY_TEXTURE_LAYERS minus one (for array textures).

The error INVALID_VALUE is generated if texture is non-zero and layer is negative. The error INVALID_OPERATION is generated if texture is non-zero and is not the name of a three dimensional, two-dimensional multisample array, one-or two-dimensional array, or cube map array texture. Unlike FramebufferTexture3D, no texture parameter is accepted.

If texture is non-zero and the command does not result in an error, the framebuffer attachment state corresponding to attachment is updated as in the...
other `FramebufferTexture` commands, except that the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER` is set to `layer`.

### Effects of Attaching a Texture Image

The remaining comments in this section apply to all forms of `FramebufferTexture*`.

If `texture` is zero, any image or array of images attached to the attachment point named by `attachment` is detached. Any additional parameters (`level`, `textarget`, and/or `layer`) are ignored when `texture` is zero. All state values of the attachment point specified by `attachment` are set to their default values listed in table 6.25.

If `texture` is not zero, and if `FramebufferTexture*` is successful, then the specified texture image will be used as the logical buffer identified by `attachment` of the framebuffer object currently bound to `target`. State values of the specified attachment point are set as follows:

- The value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is set to `TEXTURE`.
- The value of `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` is set to `texture`.
- The value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL` is set to `level`.
- If `FramebufferTexture2D` is called and `texture` is a cube map texture, then the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE` is set to `textarget`; otherwise it is set to the default value (`NONE`).
- If `FramebufferTextureLayer` or `FramebufferTexture3D` is called, then the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER` is set to `layer`; otherwise it is set to zero.
- If `FramebufferTexture` is called and `texture` is the name of a three-dimensional, cube map, two-dimensional multisample array, or one-or two-dimensional array texture, the value of `FRAMEBUFFER_ATTACHMENT_LAYERED` is set to `TRUE`; otherwise it is set to `FALSE`.

All other state values of the attachment point specified by `attachment` are set to their default values listed in table 6.25. No change is made to the state of the texture object, and any previous attachment to the `attachment` logical buffer of the framebuffer object bound to framebuffer `target` is broken. If the attachment is not successful, then no change is made to the state of either the texture object or the framebuffer object.
4.4. FRAMEBUFFER OBJECTS

Setting attachment to the value `DEPTH_STENCIL_ATTACHMENT` is a special case causing both the depth and stencil attachments of the framebuffer object to be set to texture. texture must have base internal format `DEPTH_STENCIL`, or the depth and stencil framebuffer attachments will be incomplete (see section 4.4.4).

If a texture object is deleted while its image is attached to one or more attachment points in a currently bound framebuffer object, then it is as if `FramebufferTexture*` had been called, with a texture of zero, for each attachment point to which this image was attached in that framebuffer object. In other words, the texture image is first detached from all attachment points in that framebuffer object. Note that the texture image is specifically not detached from any non-bound framebuffer objects. Detaching the texture image from any non-bound framebuffer objects is the responsibility of the application.

4.4.3 Feedback Loops Between Textures and the Framebuffer

A feedback loop may exist when a texture object is used as both the source and destination of a GL operation. When a feedback loop exists, undefined behavior results. This section describes rendering feedback loops (see section 3.8) and texture copying feedback loops (see section 3.9.4) in more detail.

Rendering Feedback Loops

The mechanisms for attaching textures to a framebuffer object do not prevent a one-or two-dimensional texture level, a face of a cube map texture level, or a layer of a two-dimensional array or three-dimensional texture from being attached to the draw framebuffer while the same texture is bound to a texture unit. While this conditions holds, texturing operations accessing that image will produce undefined results, as described at the end of section 3.9.11. Conditions resulting in such undefined behavior are defined in more detail below. Such undefined texturing operations are likely to leave the final results of fragment processing operations undefined, and should be avoided.

Special precautions need to be taken to avoid attaching a texture image to the currently bound draw framebuffer object while the texture object is currently bound and enabled for texturing. Doing so could lead to the creation of a rendering feedback loop between the writing of pixels by GL rendering operations and the simultaneous reading of those same pixels when used as texels in the currently bound texture. In this scenario, the framebuffer will be considered framebuffer complete (see section 4.4.4), but the values of fragments rendered while in this state will be undefined. The values of texture samples may be undefined as well, as described under “Rendering Feedback Loops” in section 3.8.
4.4. FRAMEBUFFER OBJECTS

Specifically, the values of rendered fragments are undefined if all of the following conditions are true:

- an image from texture object $T$ is attached to the currently bound draw framebuffer object at attachment point $A$
- the texture object $T$ is currently bound to a texture unit $U$, and
- the current programmable vertex and/or fragment processing state makes it possible (see below) to sample from the texture object $T$ bound to texture unit $U$

while either of the following conditions are true:

- the value of TEXTURE_MIN_FILTER for texture object $T$ is NEAREST or LINEAR, and the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL for attachment point $A$ is equal to the value of TEXTURE_BASE_LEVEL for the texture object $T$
- the value of TEXTURE_MIN_FILTER for texture object $T$ is one of NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, or LINEAR_MIPMAP_LINEAR, and the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL for attachment point $A$ is within the the range specified by the current values of TEXTURE_BASE_LEVEL to $q$, inclusive, for the texture object $T$. ($q$ is defined in the Mipmapping discussion of section 3.9.11).

For the purpose of this discussion, it is possible to sample from the texture object $T$ bound to texture unit $U$ if the active fragment or vertex shader contains any instructions that might sample from the texture object $T$ bound to $U$, even if those instructions might only be executed conditionally.

Note that if TEXTURE_BASE_LEVEL and TEXTURE_MAX_LEVEL exclude any levels containing image(s) attached to the currently bound draw framebuffer object, then the above conditions will not be met (i.e., the above rule will not cause the values of rendered fragments to be undefined.)

Texture Copying Feedback Loops

Similarly to rendering feedback loops, it is possible for a texture image to be attached to the currently bound read framebuffer object while the same texture image is the destination of a CopyTexImage* operation, as described under “Texture Copying Feedback Loops” in section 3.9.4. While this condition holds, a texture
copying feedback loop between the writing of texels by the copying operation and the reading of those same texels when used as pixels in the read framebuffer may exist. In this scenario, the values of texels written by the copying operation will be undefined (in the same fashion that overlapping copies via BlitFramebuffer are undefined).

Specifically, the values of copied texels are undefined if all of the following conditions are true:

- an image from texture object $T$ is attached to the currently bound read framebuffer object at attachment point $A$
- the selected read buffer is attachment point $A$
- $T$ is bound to the texture target of a CopyTexImage* operation
- the level argument of the copying operation selects the same image that is attached to $A$

### 4.4.4 Framebuffer Completeness

A framebuffer must be framebuffer complete to effectively be used as the draw or read framebuffer of the GL.

The default framebuffer is always complete if it exists; however, if no default framebuffer exists (no window system-provided drawable is associated with the GL context), it is deemed to be incomplete.

A framebuffer object is said to be framebuffer complete if all of its attached images, and all framebuffer parameters required to utilize the framebuffer for rendering and reading, are consistently defined and meet the requirements defined below. The rules of framebuffer completeness are dependent on the properties of the attached images, and on certain implementation-dependent restrictions.

The internal formats of the attached images can affect the completeness of the framebuffer, so it is useful to first define the relationship between the internal format of an image and the attachment points to which it can be attached.

- The following base internal formats from table 3.11 are color-renderable: RED, RG, RGB, and RGBA. The sized internal formats from table 3.12 that have a color-renderable base internal format are also color-renderable. No other formats, including compressed internal formats, are color-renderable.
- An internal format is depth-renderable if it is DEPTH_COMPONENT or one of the formats from table 3.13 whose base internal format is DEPTH_-COMPONENT or DEPTH_STENCIL. No other formats are depth-renderable.
4.4. FRAMEBUFFER OBJECTS

- An internal format is *stencil-renderable* if it is **STENCIL_INDEX** or **DEPTH_STENCIL**, if it is one of the **STENCIL_INDEX** formats from table 4.10, or if it is one of the formats from table 3.13 whose base internal format is **DEPTH_STENCIL**. No other formats are stencil-renderable.

**Framebuffer Attachment Completeness**

If the value of **FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE** for the framebuffer attachment point *attachment* is not **NONE**, then it is said that a framebuffer-attachable image, named *image*, is attached to the framebuffer at the attachment point. *image* is identified by the state in *attachment* as described in section 4.4.2.

The framebuffer attachment point *attachment* is said to be *framebuffer attachment complete* if the value of **FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE** for *attachment* is **NONE** (i.e., no image is attached), or if all of the following conditions are true:

- *image* is a component of an existing object with the name specified by the value of **FRAMEBUFFER_ATTACHMENT_OBJECT_NAME**, and of the type specified by the value of **FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE**.
- The width and height of *image* are non-zero.
- If the value of **FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE** is **TEXTURE** and the value of **FRAMEBUFFER_ATTACHMENT_OBJECT_NAME** names a three-dimensional texture, then the value of **FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER** must be smaller than the depth of the texture.
- If the value of **FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE** is **TEXTURE** and the value of **FRAMEBUFFER_ATTACHMENT_OBJECT_NAME** names a one-or two-dimensional array or cube map array texture, then the value of **FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER** must be smaller than the number of layers in the texture.
- If *attachment* is **COLOR_ATTACHMENTi**, then *image* must have a color-renderable internal format.
- If *attachment* is **DEPTH_ATTACHMENT**, then *image* must have a depth-renderable internal format.
- If *attachment* is **STENCIL_ATTACHMENT**, then *image* must have a stencil-renderable internal format.
Whole Framebuffer Completeness

Each rule below is followed by an error token enclosed in { brackets }. The meaning of these errors is explained below and under “Effects of Framebuffer Completeness on Framebuffer Operations” later in section 4.4.4.

The framebuffer object target is said to be framebuffer complete if all the following conditions are true:

- if target is the default framebuffer, and the default framebuffer exists.

  { FRAMEBUFFER_UNDEFINED }

- All framebuffer attachment points are framebuffer attachment complete.

  { FRAMEBUFFER_INCOMPLETE_ATTACHMENT }

- There is at least one image attached to the framebuffer.

  { FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT }

- The combination of internal formats of the attached images does not violate an implementation-dependent set of restrictions.

  { FRAMEBUFFER_UNSUPPORTED }

- The value of RENDERBUFFER_SAMPLES is the same for all attached renderbuffers; the value of TEXTURE_SAMPLES is the same for all attached textures; and, if the attached images are a mix of renderbuffers and textures, the value of RENDERBUFFER_SAMPLES matches the value of TEXTURE_SAMPLES.

  { FRAMEBUFFER_INCOMPLETE_MULTISAMPLE }

- The value of TEXTURE_FIXED_SAMPLE_LOCATIONS is the same for all attached textures; and, if the attached images are a mix of renderbuffers and textures, the value of TEXTURE_FIXED_SAMPLE_LOCATIONS must be TRUE for all attached textures.

  { FRAMEBUFFER_INCOMPLETE_MULTISAMPLE }

- If any framebuffer attachment is layered, all populated attachments must be layered. Additionally, all populated color attachments must be from textures...
of the same target (three-dimensional, one-or two-dimensional array, cube map, or cube map array textures).

{ FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS }

The token in brackets after each clause of the framebuffer completeness rules specifies the return value of CheckFramebufferStatus (see below) that is generated when that clause is violated. If more than one clause is violated, it is implementation-dependent which value will be returned by CheckFramebufferStatus.

Performing any of the following actions may change whether the framebuffer is considered complete or incomplete:

- Binding to a different framebuffer with BindFramebuffer.
- Attaching an image to the framebuffer with FramebufferTexture* or FramebufferRenderbuffer.
- Detaching an image from the framebuffer with FramebufferTexture* or FramebufferRenderbuffer.
- Changing the internal format of a texture image that is attached to the framebuffer by calling CopyTexImage* or CompressedTexImage*.
- Changing the internal format of a renderbuffer that is attached to the framebuffer by calling RenderbufferStorage.
- Deleting, with DeleteTextures or DeleteRenderbuffers, an object containing an image that is attached to a currently bound framebuffer object. that is bound to the framebuffer.
- Associating a different window system-provided drawable, or no drawable, with the default framebuffer using a window system binding API such as those described in section 1.7.2.

Although the GL defines a wide variety of internal formats for framebuffer-attachable images, such as texture images and renderbuffer images, some implementations may not support rendering to particular combinations of internal formats. If the combination of formats of the images attached to a framebuffer object are not supported by the implementation, then the framebuffer is not complete under the clause labeled FRAMEBUFFER_UNSUPPORTED.
4.4. FRAMEBUFFER OBJECTS

Implementations are required to support certain combinations of framebuffer internal formats as described under “Required Framebuffer Formats” in section 4.4.4.

Because of the implementation-dependent clause of the framebuffer completeness test in particular, and because framebuffer completeness can change when the set of attached images is modified, it is strongly advised, though not required, that an application check to see if the framebuffer is complete prior to rendering. The status of the framebuffer object currently bound to target can be queried by calling

```c
enum CheckFramebufferStatus( enum target );
```

`target` must be DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER. If CheckFramebufferStatus generates an error, zero is returned. Otherwise, a value is returned that identifies whether or not the framebuffer object bound to target is complete, and if not complete the value identifies one of the rules of framebuffer completeness that is violated. If the framebuffer object is complete, then FRAMEBUFFER_COMPLETE is returned.

The values of SAMPLE_BUFFERS and SAMPLES are derived from the attachments of the currently bound draw framebuffer object. If the current DRAW_FRAMEBUFFER_BINDING is not framebuffer complete, then both SAMPLE_BUFFERS and SAMPLES are undefined. Otherwise, SAMPLES is equal to the value of RENDERBUFFER_SAMPLES or TEXTURE_SAMPLES (depending on the type of the attached images), which must all have the same value. Further, SAMPLE_BUFFERS is one if SAMPLES is non-zero. Otherwise, SAMPLE_BUFFERS is zero.

Required Framebuffer Formats

Implementations must support framebuffer objects with up to MAX_COLOR_ATTACHMENTS color attachments, a depth attachment, and a stencil attachment. Each color attachment may be in any of the required color formats for textures and renderbuffers described in sections 3.9.3 and 4.4.2. The depth attachment may be in any of the required depth or combined depth+stencil formats described in those sections, and the stencil attachment may be in any of the required combined depth+stencil formats. However, when both depth and stencil attachments are present, implementations are only required to support framebuffer objects where both attachments refer to the same image.

There must be at least one default framebuffer format allowing creation of a default framebuffer supporting front-buffered rendering.
4.4. FRAMEBUFFER OBJECTS

Effects of Framebuffer Completeness on Framebuffer Operations

Attempting to render to or read from a framebuffer which is not framebuffer complete will generate an INVALID_FRAMEBUFFER_OPERATION error. This means that rendering commands such as DrawArrays or one of the other drawing commands defined in section 2.8.3, as well as commands that read the framebuffer such as ReadPixels, CopyTexImage, and CopyTexSubImage, will generate the error INVALID_FRAMEBUFFER_OPERATION if called while the framebuffer is not framebuffer complete. This error is generated regardless of whether fragments are actually read from or written to the framebuffer. For example, it will be generated when a rendering command is called and the framebuffer is incomplete even if RASTERIZER_DISCARD is enabled.

4.4.5 Effects of Framebuffer State on Framebuffer Dependent Values

The values of the state variables listed in table 6.62 may change when a change is made to DRAW_FRAMEBUFFER_BINDING, to the state of the currently bound draw framebuffer object, or to an image attached to that framebuffer object.

When DRAW_FRAMEBUFFER_BINDING is zero, the values of the state variables listed in table 6.62 are implementation defined.

When DRAW_FRAMEBUFFER_BINDING is non-zero, if the currently bound draw framebuffer object is not framebuffer complete, then the values of the state variables listed in table 6.62 are undefined.

When DRAW_FRAMEBUFFER_BINDING is non-zero and the currently bound draw framebuffer object is framebuffer complete, then the values of the state variables listed in table 6.62 are completely determined by DRAW_FRAMEBUFFER_BINDING, the state of the currently bound draw framebuffer object, and the state of the images attached to that framebuffer object. The actual sizes of the color, depth, or stencil bit planes can be obtained by querying an attachment point using GetFramebufferAttachmentParameteriv, or querying the object attached to that point. If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE at a particular attachment point is RENDERBUFFER, the sizes may be determined by calling GetRenderbufferParameteriv as described in section 6.1.3. If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE at a particular attachment point is TEXTURE, the sizes may be determined by calling GetTexParameter, as described in section 6.1.3.

4.4.6 Mapping between Pixel and Element in Attached Image

When DRAW_FRAMEBUFFER_BINDING is non-zero, an operation that writes to the framebuffer modifies the image attached to the selected logical buffer, and an oper-
4.4. FRAMEBUFFER OBJECTS

ation that reads from the framebuffer reads from the image attached to the selected logical buffer.

If the attached image is a renderbuffer image, then the window coordinates \((x_w, y_w)\) corresponds to the value in the renderbuffer image at the same coordinates.

If the attached image is a texture image, then the window coordinates \((x_w, y_w)\) correspond to the texel \((i, j, k)\) from figure 3.7 as follows:

\[
i = (x_w - b) \\
j = (y_w - b) \\
k = (\text{layer} - b)
\]

where \(b\) is the texture image’s border width and \text{layer} is the value of \text{FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER} for the selected logical buffer. For a two-dimensional texture, \(k\) and \text{layer} are irrelevant; for a one-dimensional texture, \(j, k,\) and \text{layer} are irrelevant.

\((x_w, y_w)\) corresponds to a border texel if \(x_w, y_w,\) or \text{layer} is less than the border width, or if \(x_w, y_w,\) or \text{layer} is greater than or equal to the border width plus the width, height, or depth, respectively, of the texture image.

Conversion to Framebuffer-Attachable Image Components

When an enabled color value is written to the framebuffer while the draw framebuffer binding is non-zero, for each draw buffer the R, G, B, and A values are converted to internal components as described in table 3.11, according to the table row corresponding to the internal format of the framebuffer-attachable image attached to the selected logical buffer, and the resulting internal components are written to the image attached to logical buffer. The masking operations described in section 4.2.2 are also effective.

Conversion to RGBA Values

When a color value is read while the read framebuffer binding is non-zero, or is used as the source of a logical operation or for blending while the draw framebuffer binding is non-zero, components of that color taken from the framebuffer-attachable image attached to the selected logical buffer are first converted to R, G, B, and A values according to table 3.23 and the internal format of the attached image.
### 4.4.7 Layered Framebuffers

A framebuffer is considered to be layered if it is complete and all of its populated attachments are layered. When rendering to a layered framebuffer, each fragment generated by the GL is assigned a layer number. The layer number for a fragment is zero if

- geometry shaders are disabled, or
- the current geometry shader does not statically assign a value to the built-in output variable `gl_Layer`.

Otherwise, the layer for each point, line, or triangle emitted by the geometry shader is taken from the `gl_Layer` output of one of the vertices of the primitive. The vertex used is implementation-dependent. To get defined results, all vertices of each primitive emitted should set the same value for `gl_Layer`. Since the `EndPrimitive` built-in function starts a new output primitive, defined results can be achieved if `EndPrimitive` is called between two vertices emitted with different layer numbers. A layer number written by a geometry shader has no effect if the framebuffer is not layered.

When fragments are written to a layered framebuffer, the fragment’s layer number selects an image from the array of images at each attachment point to use for the stencil test (see section 4.1.4), depth buffer test (see section 4.1.5), and for blending and color buffer writes (see section 4.1.7). If the fragment’s layer number is negative, or greater than or equal to the minimum number of layers of any attachment, the effects of the fragment on the framebuffer contents are undefined.

When the `Clear` or `ClearBuffer*` commands are used to clear a layered framebuffer attachment, all layers of the attachment are cleared.

When commands such as `ReadPixels` read from a layered framebuffer, the image at layer zero of the selected attachment is always used to obtain pixel values.

When cube map texture levels are attached to a layered framebuffer, there are six layers, numbered zero through five. Each layer number corresponds to a cube map face, as shown in table 4.12.

When cube map array texture levels are attached to a layered framebuffer, the layer number corresponds to a layer-face. The layer-face can be translated into an array layer and a cube map face by

\[
\text{array-layer} = \left\lfloor \frac{\text{layer}}{6} \right\rfloor, \\
\text{face} = \text{layer} \mod 6
\]
### Table 4.12: Layer numbers for cube map texture faces. The layers are numbered in the same sequence as the cube map face token values.

<table>
<thead>
<tr>
<th>Layer Number</th>
<th>Cube Map Face</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TEXTURE_CUBE_MAP_POSITIVE_X</td>
</tr>
<tr>
<td>1</td>
<td>TEXTURE_CUBE_MAP_NEGATIVE_X</td>
</tr>
<tr>
<td>2</td>
<td>TEXTURE_CUBE_MAP_POSITIVE_Y</td>
</tr>
<tr>
<td>3</td>
<td>TEXTURE_CUBE_MAP_NEGATIVE_Y</td>
</tr>
<tr>
<td>4</td>
<td>TEXTURE_CUBE_MAP_POSITIVE_Z</td>
</tr>
<tr>
<td>5</td>
<td>TEXTURE_CUBE_MAP_NEGATIVE_Z</td>
</tr>
</tbody>
</table>

The face number corresponds to the cube map faces as shown in table 4.12.
Chapter 5

Special Functions

This chapter describes additional GL functionality that does not fit easily into any of the preceding chapters. This functionality consists of flushing and finishing (used to synchronize the GL command stream), and hints.

5.1 Timer Queries

Timer queries use query objects to track the amount of time needed to fully complete a set of GL commands, or to determine the current time of the GL.

When BeginQuery and EndQuery are called with a target of TIME_ELAPSED, the GL prepares to start and stop the timer used for timer queries. The timer is started or stopped when the effects from all previous commands on the GL client and server state and the framebuffer have been fully realized. The BeginQuery and EndQuery commands may return before the timer is actually started or stopped. When the timer query timer is finally stopped, the elapsed time (in nanoseconds) is written to the corresponding query object as the query result value, and the query result for that object is marked as available.

A timer query object is created with the command

```c
void QueryCounter( uint id, enum target );
```

target must be TIMESTAMP. If id is an unused query object name, the name is marked as used and associated with a new query object of type TIMESTAMP. Otherwise id must be the name of an existing query object of that type.

When QueryCounter is called, the GL records the current time into the corresponding query object. The time is recorded after all previous commands on the GL client and server state and the framebuffer have been fully realized. When the time is recorded, the query result for that object is marked available. QueryCounter
5.2. FLUSH AND FINISH

Timer queries can be used within a `BeginQuery / EndQuery` block where the target is `TIME_ELAPSED` and it does not affect the result of that query object. `QueryCounter` fails and an `INVALID_OPERATION` error is generated if `id` is not a name returned from a previous call to `GenQueries`, or if such a name has since been deleted with `DeleteQueries`.

If `id` is already in use within a `BeginQuery / EndQuery` block, or if `id` is the name of an existing query object whose type does not match `target`, an `INVALID_OPERATION` error is generated.

The current time of the GL may be queried by calling `GetIntegerv` or `GetIntegerv64` with the symbolic constant `TIMESTAMP`. This will return the GL time after all previous commands have reached the GL server but have not yet necessarily executed. By using a combination of this synchronous get command and the asynchronous timestamp query object target, applications can measure the latency between when commands reach the GL server and when they are realized in the framebuffer.

5.2 Flush and Finish

The command

```c
void Flush(void);
```

indicates that all commands that have previously been sent to the GL must complete in finite time.

The command

```c
void Finish(void);
```

forces all previous GL commands to complete. `Finish` does not return until all effects from previously issued commands on GL client and server state and the framebuffer are fully realized.

5.3 Sync Objects and Fences

Sync objects act as a *synchronization primitive* - a representation of events whose completion status can be tested or waited upon. Sync objects may be used for synchronization with operations occurring in the GL state machine or in the graphics pipeline, and for synchronizing between multiple graphics contexts, among other purposes.
Sync objects have a status value with two possible states: *signaled* and *unsignaled*. Events are associated with a sync object. When a sync object is created, its status is set to unsignaled. When the associated event occurs, the sync object is signaled (its status is set to signaled). The GL may be asked to wait for a sync object to become signaled.

Initially, only one specific type of sync object is defined: the fence sync object, whose associated event is triggered by a fence command placed in the GL command stream. Fence sync objects are used to wait for partial completion of the GL command stream, as a more flexible form of *Finish*.

The command

```c
sync FenceSync( enum condition, bitfield flags );
```

creates a new fence sync object, inserts a fence command in the GL command stream and associates it with that sync object, and returns a non-zero name corresponding to the sync object.

When the specified *condition* of the sync object is satisfied by the fence command, the sync object is signaled by the GL, causing any *ClientWaitSync* or *WaitSync* commands (see below) blocking on *sync* to *unblock*. No other state is affected by *FenceSync* or by execution of the associated fence command.

*condition* must be *SYNC_GPU_COMMANDS_COMPLETE*. This condition is satisfied by completion of the fence command corresponding to the sync object and all preceding commands in the same command stream. The sync object will not be signaled until all effects from these commands on GL client and server state and the framebuffer are fully realized. Note that completion of the fence command occurs once the state of the corresponding sync object has been changed, but commands waiting on that sync object may not be unblocked until some time after the fence command completes.

*flags* must be 0.

Each sync object contains a number of *properties* which determine the state of the object and the behavior of any commands associated with it. Each property has a *property name* and *property value*. The initial property values for a sync object created by *FenceSync* are shown in table 5.1.

Properties of a sync object may be queried with *GetSynciv* (see section 6.1.8). The *SYNC_STATUS* property will be changed to *SIGNALED* when *condition* is satisfied.

If *FenceSync* fails to create a sync object, zero will be returned and a GL error will be generated as described. An *INVALID_ENUM* error is generated if *condition* is an invalid value. For placeholders, see table 5.1.

---

1 *flags* is a placeholder for anticipated future extensions of fence sync object capabilities.
### 5.3. SYNC OBJECTS AND FENCES

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJECT_TYPE</td>
<td>SYNC_FENCE</td>
</tr>
<tr>
<td>SYNC_CONDITION</td>
<td>condition</td>
</tr>
<tr>
<td>SYNC_STATUS</td>
<td>UNSIGNALED</td>
</tr>
<tr>
<td>SYNC_FLAGS</td>
<td>flags</td>
</tr>
</tbody>
</table>

Table 5.1: Initial properties of a sync object created with FenceSync.

is not SYNC_GPU_COMMANDS_COMPLETE. If flags is not zero, an INVALID_VALUE error is generated.

A sync object can be deleted by passing its name to the command

```
void DeleteSync(sync sync);
```

If the fence command corresponding to the specified sync object has completed, or if no ClientWaitSync or WaitSync commands are blocking on sync, the object is deleted immediately. Otherwise, sync is flagged for deletion and will be deleted when it is no longer associated with any fence command and is no longer blocking any ClientWaitSync or WaitSync command. In either case, after returning from DeleteSync the sync name is invalid and can no longer be used to refer to the sync object.

DeleteSync will silently ignore a sync value of zero. An INVALID_VALUE error is generated if sync is neither zero nor the name of a sync object.

#### 5.3.1 Waiting for Sync Objects

The command

```
enum ClientWaitSync(sync sync, bitfield flags, uint64 timeout);
```

causes the GL to block, and will not return until the sync object sync is signaled, or until the specified timeout period expires. timeout is in units of nanoseconds. timeout is adjusted to the closest value allowed by the implementation-dependent timeout accuracy, which may be substantially longer than one nanosecond, and may be longer than the requested period.

If sync is signaled at the time ClientWaitSync is called, then ClientWaitSync returns immediately. If sync is unsignaled at the time ClientWaitSync is called, then ClientWaitSync will block and will wait up to timeout nanoseconds.
for *sync* to become signaled. *flags* controls command flushing behavior, and may be `SYNC_FLUSH_COMMANDS_BIT`, as discussed in section 5.3.2.

**ClientWaitSync** returns one of four status values. A return value of `ALREADY_SIGNALED` indicates that *sync* was signaled at the time **ClientWaitSync** was called. `ALREADY_SIGNALED` will always be returned if *sync* was signaled, even if the value of *timeout* is zero. A return value of `TIMEOUT_EXPIRED` indicates that the specified timeout period expired before *sync* was signaled. A return value of `CONDITION_SATISFIED` indicates that *sync* was signaled before the timeout expired. Finally, if an error occurs, in addition to generating a GL error as specified below, **ClientWaitSync** immediately returns `WAIT_FAILED` without blocking.

If the value of *timeout* is zero, then **ClientWaitSync** does not block, but simply tests the current state of *sync*. `TIMEOUT_EXPIRED` will be returned in this case if *sync* is not signaled, even though no actual wait was performed.

If *sync* is not the name of a sync object, an `INVALID_VALUE` error is generated. If *flags* contains any bits other than `SYNC_FLUSH_COMMANDS_BIT`, an `INVALID_VALUE` error is generated.

The command

```c
void WaitSync( sync sync, bitfield flags,
              uint64 timeout);
```

is similar to **ClientWaitSync**, but instead of blocking and not returning to the application until *sync* is signaled, **WaitSync** returns immediately, instead causing the GL server to block \(^2\) until *sync* is signaled \(^3\).

*sync* has the same meaning as for **ClientWaitSync**.

*timeout* must currently be the special value `TIMEOUT_IGNORED`, and is not used. Instead, **WaitSync** will always wait no longer than an implementation-dependent timeout. The duration of this timeout in nanoseconds may be queried by calling `GetInteger64v` with the symbolic constant `MAX_SERVER_WAIT_TIMEOUT`. There is currently no way to determine whether **WaitSync** unblocked because the timeout expired or because the sync object being waited on was signaled.

*flags* must be 0.

If an error occurs, **WaitSync** generates a GL error as specified below, and does not cause the GL server to block.

\(^2\)The GL server may choose to wait either in the CPU executing server-side code, or in the GPU hardware if it supports this operation.

\(^3\) **WaitSync** allows applications to continue to queue commands from the client in anticipation of the sync being signalled, increasing client-server parallelism.
If \textit{sync} is not the name of a sync object, an \texttt{INVALID\_VALUE} error is generated. If \textit{timeout} is not \texttt{TIMEOUT\_IGNORED} or \texttt{flags} is not zero, an \texttt{INVALID\_VALUE} error is generated\footnote{flags and \textit{timeout} are placeholders for anticipated future extensions of sync object capabilities. They must have these reserved values in order that existing code calling \texttt{WaitSync} operate properly in the presence of such extensions.}.

\section*{Multiple Waiters}

It is possible for both the GL client to be blocked on a sync object in a \texttt{ClientWaitSync} command, the GL server to be blocked as the result of a previous \texttt{WaitSync} command, and for additional \texttt{WaitSync} commands to be queued in the GL server, all for a single sync object. When such a sync object is signaled in this situation, the client will be unblocked, the server will be unblocked, and all such queued \texttt{WaitSync} commands will continue immediately when they are reached.

See appendix D.2 for more information about blocking on a sync object in multiple GL contexts.

\subsection*{5.3.2 Signalling}

A fence sync object enters the signalled state only once the corresponding fence command has completed and signalled the sync object.

If the sync object being blocked upon will not be signalled in finite time (for example, by an associated fence command issued previously, but not yet flushed to the graphics pipeline), then \texttt{ClientWaitSync} may hang forever. To help prevent this behavior\footnote{The simple flushing behavior defined by \texttt{SYNC\_FLUSH\_COMMANDS\_BIT} will not help when waiting for a fence command issued in another context’s command stream to complete. Applications which block on a fence sync object must take additional steps to assure that the context from which the corresponding fence command was issued has flushed that command to the graphics pipeline.}, if the \texttt{SYNC\_FLUSH\_COMMANDS\_BIT} bit is set in \texttt{flags}, and \texttt{sync} is unsignaled when \texttt{ClientWaitSync} is called, then the equivalent of \texttt{Flush} will be performed before blocking on \texttt{sync}.

If a sync object is marked for deletion while a client is blocking on that object in a \texttt{ClientWaitSync} command, or a GL server is blocking on that object as a result of a prior \texttt{WaitSync} command, deletion is deferred until the sync object is signalled and all blocked GL clients and servers are unblocked.

Additional constraints on the use of sync objects are discussed in appendix D.

State must be maintained to indicate which sync object names are currently in use. The state require for each sync object in use is an integer for the specific type, an integer for the condition, and a bit indicating whether the object is signalled.
5.4. **HINTS**

<table>
<thead>
<tr>
<th>Target</th>
<th>Hint description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINE_SMOOTH_HINT</td>
<td>Line sampling quality</td>
</tr>
<tr>
<td>POLYGON_SMOOTH_HINT</td>
<td>Polygon sampling quality</td>
</tr>
<tr>
<td>TEXTURE_COMPRESSION_HINT</td>
<td>Quality and performance of texture image compression</td>
</tr>
<tr>
<td>FRAGMENT_SHADER_DERIVATIVE_HINT</td>
<td>Derivative accuracy for fragment processing built-in functions</td>
</tr>
</tbody>
</table>

Table 5.2: Hint targets and descriptions.

or unsignaled. The initial values of sync object state are defined as specified by FenceSync.

### 5.4 Hints

Certain aspects of GL behavior, when there is room for variation, may be controlled with hints. A hint is specified using

```c
void Hint(enum target, enum hint);
```

*target* is a symbolic constant indicating the behavior to be controlled, and *hint* is a symbolic constant indicating what type of behavior is desired. The possible *targets* are described in table 5.2; for each *target*, *hint* must be one of FASTEST, indicating that the most efficient option should be chosen; NICEST, indicating that the highest quality option should be chosen; and DONT_CARE, indicating no preference in the matter.

For the texture compression hint, a *hint* of FASTEST indicates that texture images should be compressed as quickly as possible, while NICEST indicates that the texture images be compressed with as little image degradation as possible. FASTEST should be used for one-time texture compression, and NICEST should be used if the compression results are to be retrieved by `GetCompressedTexImage` (section 6.1.4) for reuse.

The interpretation of hints is implementation-dependent. An implementation may ignore them entirely.

The initial value of all hints is DONT_CARE.
Chapter 6

State and State Requests

The state required to describe the GL machine is enumerated in section 6.2. Most state is set through the calls described in previous chapters, and can be queried using the calls described in section 6.1.

6.1 Querying GL State

6.1.1 Simple Queries

Much of the GL state is completely identified by symbolic constants. The values of these state variables can be obtained using a set of `Get` commands. There are five commands for obtaining simple state variables:

```c
void GetBooleanv(enum pname, boolean *data);
void GetIntegerv(enum pname, int *data);
void GetInteger64v(enum pname, int64 *data);
void GetFloatv(enum pname, float *data);
void GetDoublev(enum pname, double *data);
```

The commands obtain boolean, integer, 64-bit integer, floating-point, or double-precision state variables. `pname` is a symbolic constant indicating the state variable to return. `data` is a pointer to a scalar or array of the indicated type in which to place the returned data.

Indexed simple state variables are queried with the commands

```c
void GetBooleaniv( enum target, uint index, boolean *data );
void GetIntegerv( enum target, uint index, int *data );
void GetInteger64v( enum target, uint index, int64 *data );
void GetFloatv( enum target, uint index, float *data );
void GetDoublev( enum target, uint index, double *data );
```
6.1. QUERYING GL STATE

```c
void GetFloativ(enum target, uint index, float *data);
void GetDoubleiv(enum target, uint index, double *data);
void GetInteger64iv(enum target, uint index, int64_t *data);
```

target is the name of the indexed state and index is the index of the particular element being queried. data is a pointer to a scalar or array of the indicated type in which to place the returned data. An INVALID_VALUE error is generated if index is outside the valid range for the indexed state target.

Finally,

```c
boolean IsEnabled(enum cap);
```

can be used to determine if cap is currently enabled (as with Enable) or disabled, and

```c
boolean IsEnabledi(enum target, uint index);
```

can be used to determine if the indexed state corresponding to target and index is enabled or disabled. An INVALID_VALUE error is generated if index is outside the valid range for the indexed state target.

6.1.2 Data Conversions

If a Get command is issued that returns value types different from the type of the value being obtained, a type conversion is performed. If GetBooleaniv is called, a floating-point or integer value converts to FALSE if and only if it is zero (otherwise it converts to TRUE). If any of the other simple queries are called, a boolean value of TRUE or FALSE is interpreted as 1 or 0, respectively. If GetIntegerv or GetInteger64v are called, a floating-point value is rounded to the nearest integer, unless the value is an RGBA color component, a DepthRange value, or a depth buffer clear value. In these cases, the Get command converts the floating-point value to an integer according to the INT entry of table 4.8; a value not in [−1, 1] converts to an undefined value. If GetFloatv is called, a boolean value of TRUE or FALSE is interpreted as 1.0 or 0.0, respectively, an integer is coerced to floating-point, and a double-precision floating-point value is converted to single-precision. Analogous conversions are carried out in the case of GetDoublev.

If a value is so large in magnitude that it cannot be represented with the requested type, then the nearest value representable using the requested type is returned. When querying bitmasks (such as SAMPLE_MASK_VALUE or STENCIL_WRITEMASK) with GetIntegerv, the mask value is treated as a signed integer, so
that mask values with the high bit set will not be clamped when returned as signed integers.

Unless otherwise indicated, multi-valued state variables return their multiple values in the same order as they are given as arguments to the commands that set them. For instance, the two DepthRange parameters are returned in the order $n$ followed by $f$.

Most texture state variables are qualified by the value of ACTIVE_TEXTURE to determine which server texture state vector is queried. Table 6.14 indicates those state variables which are qualified by ACTIVE_TEXTURE during state queries.

Vertex array state variables are qualified by the value of VERTEX_ARRAY_BINDING to determine which vertex array object is queried. Tables 6.5 and 6.6 define the set of state stored in a vertex array object.

### 6.1.3 Enumerated Queries

Other commands exist to obtain state variables that are identified by a category as well as a symbolic constant.

The commands

```c
void GetTexParameter{if}v( enum target, enum value, T data );
void GetTexParameterI{i ui}v( enum target, enum value, T data );
```

place information about texture parameter value for the specified target into data. value must be IMAGE_FORMAT_COMPATIBILITY_TYPE, TEXTURE_IMMUTABLE_FORMAT, or one of the symbolic values in table 3.16.

**target** may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_RECTANGLE, TEXTURE_CUBE_MAP, or TEXTURE_CUBE_MAP_ARRAY, indicating the currently bound one-, two-, three-dimensional, one- or two-dimensional array, rectangular, cube map, or cube map array texture object.

Querying value TEXTURE_BORDER_COLOR with GetTexParameterIiv or GetTexParameterIuiv returns the border color values as signed integers or unsigned integers, respectively; otherwise the values are returned as described in section 6.1.2. If the border color is queried with a type that does not match the original type with which it was specified, the result is undefined.

```c
void GetTexLevelParameter{if}v( enum target, int lod, enum value, T data );
```
places information about texture image parameter value for level-of-detail lod of the specified target into data. value must be one of the symbolic values in table 6.17.

target may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP_ARRAY, TEXTURE_RECTANGLE, TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, TEXTURE_CUBE_MAP_NEGATIVE_Z, TEXTURE_2D_MULTISAMPLE, TEXTURE_2D_MULTISAMPLE_ARRAY, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_3D, PROXY_TEXTURE_1D_ARRAY, PROXY_TEXTURE_2D_ARRAY, PROXY_TEXTURE_CUBE_MAP_ARRAY, PROXY_TEXTURE_RECTANGLE, PROXY_TEXTURE_CUBE_MAP, PROXY_TEXTURE_2D_MULTISAMPLE, or PROXY_TEXTURE_2D_MULTISAMPLE_ARRAY, indicating the one-, two-, or three-dimensional texture, one-or two-dimensional array texture, cube map array texture, rectangular texture, one of the six distinct 2D images making up the cube map texture object, two-dimensional multisample texture, two-dimensional multisample array texture; or the one-, two-, three-dimensional, one-or two-dimensional array, cube map array, rectangular, cube map, two-dimensional multisample, or two-dimensional multisample array proxy state vector.

target may also be TEXTURE_BUFFER, indicating the texture buffer. In the case lod must be zero or an INVALID_VALUE error is generated.

Note that TEXTURE_CUBE_MAP is not a valid target parameter for GetTexLevelParameter, because it does not specify a particular cube map face.

lod determines which level-of-detail’s state is returned. If lod is less than zero or larger than the maximum allowable level-of-detail, then an INVALID_VALUE error is generated.

For texture images with uncompressed internal formats, queries of value TEXTURE_RED_TYPE, TEXTURE_GREEN_TYPE, TEXTURE_BLUE_TYPE, TEXTURE_ALPHA_TYPE, and TEXTURE_DEPTH_TYPE return the data type used to store the component. Types NONE, SIGNED_NORMALIZED, UNSIGNED_NORMALIZED, FLOAT, INT, and UNSIGNED_INT respectively indicate missing, signed normalized fixed-point, unsigned normalized fixed-point, floating-point, signed unnormalized integer, and unsigned unnormalized integer components. Queries of value TEXTURE_RED_SIZE, TEXTURE_GREEN_SIZE, TEXTURE_BLUE_SIZE, TEXTURE_ALPHA_SIZE, TEXTURE_DEPTH_SIZE, TEXTURE_STENCIL_SIZE, and TEXTURE_SHARED_SIZE return the actual resolutions of the stored image array components, not the resolutions specified when the image array was defined.

For texture images with compressed internal formats, the types returned spec-
ify how components are interpreted after decompression, while the resolutions returned specify the component resolution of an uncompressed internal format that produces an image of roughly the same quality as the compressed image in question. Since the quality of the implementation’s compression algorithm is likely data-dependent, the returned component sizes should be treated only as rough approximations.

Querying value TEXTURE_COMPRESSED_IMAGE_SIZE returns the size (in ubytes) of the compressed texture image that would be returned by GetCompressedTexImage (section 6.1.4). Querying TEXTURE_COMPRESSED_IMAGE_SIZE is not allowed on texture images with an uncompressed internal format or on proxy targets and will result in an INVALID_OPERATION error if attempted.

Queries of value TEXTURE_INTERNAL_FORMAT, TEXTURE_WIDTH, TEXTURE_HEIGHT, and TEXTURE_DEPTH return the internal format, width, height, and depth, respectively, as specified when the image array was created.

6.1.4 Texture Queries

The command

        void GetTexImage( enum tex, int lod, enum format,
                enum type, void *img );

is used to obtain texture images. It is somewhat different from the other Get* commands; tex is a symbolic value indicating which texture (or texture face in the case of a cube map texture target name) is to be obtained. TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP_ARRAY, and TEXTURE_RECTANGLE indicate a one-, two-, or three-dimensional, one- or two-dimensional array, cube map array, or rectangular texture respectively. TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, and TEXTURE_CUBE_MAP_NEGATIVE_Z indicate the respective face of a cube map texture. lod is a level-of-detail number, format is a pixel format from table 3.3, type is a pixel type from table 3.2.

Any of the following mismatches between format and the internal format of the texture image will generate an INVALID_OPERATION error:

- format is a color format (one of the formats in table 3.3 whose target is the color buffer) and the base internal format of the texture image is not a color format.
6.1. QUERYING GL STATE

- **format** is `DEPTH_COMPONENT` and the base internal format is not `DEPTH_COMPONENT` or `DEPTH_STENCIL`.

- **format** is `DEPTH_STENCIL` and the base internal format is not `DEPTH_STENCIL`.

- **format** is one of the integer formats in table 3.3 and the internal format of the texture image is not integer, or **format** is not one of the integer formats in table 3.3 and the internal format is integer.

GetTexImage obtains component groups from a texture image with the indicated level-of-detail. If **format** is a color format then the components are assigned among R, G, B, and A according to table 6.1, starting with the first group in the first row, and continuing by obtaining groups in order from each row and proceeding from the first row to the last, and from the first image to the last for three-dimensional textures. One- and two-dimensional array and cube map array textures are treated as two-, three-, and three-dimensional images, respectively, where the layers are treated as rows or images. If **format** is `DEPTH_COMPONENT`, then each depth component is assigned with the same ordering of rows and images. If **format** is `DEPTH_STENCIL`, then each depth component and each stencil index is assigned with the same ordering of rows and images.

These groups are then packed and placed in client or pixel buffer object memory. If a pixel pack buffer is bound (as indicated by a non-zero value of `PIXEL_PACK_BUFFER_BINDING`), **img** is an offset into the pixel pack buffer; otherwise, **img** is a pointer to client memory. Pixel storage modes that are applicable to ReadPixels are applied.

For three-dimensional, two-dimensional array, and cube map array textures, pixel storage operations are applied as if the image were two-dimensional, except that the additional pixel storage state values `PACK_IMAGE_HEIGHT` and `PACK_SKIP_IMAGES` are applied. The correspondence of texels to memory locations is as defined for TexImage3D in section 3.9.3.

The row length, number of rows, image depth, and number of images are determined by the size of the texture image (including any borders). Calling GetTexImage with **lod** less than zero or larger than the maximum allowable causes the error INVALID_VALUE. Calling GetTexImage with a **format** of `STENCIL_INDEX` causes the error INVALID_ENUM. Calling GetTexImage with a non-zero **lod** when **tex** is `TEXTURE_RECTANGLE` causes the error INVALID_VALUE. If a pixel pack buffer object is bound and packing the texture image into the buffer’s memory would exceed the size of the buffer, an INVALID_OPERATION error results. If a pixel pack buffer object is bound and **img** is not evenly divisible by the number of
6.1. QUERYING GL STATE

<table>
<thead>
<tr>
<th>Base Internal Format</th>
<th>R</th>
<th>G</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED</td>
<td>$R_i$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RG</td>
<td>$R_i$</td>
<td>$G_i$</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RGB</td>
<td>$R_i$</td>
<td>$G_i$</td>
<td>$B_i$</td>
<td>1</td>
</tr>
<tr>
<td>RGBA</td>
<td>$R_i$</td>
<td>$G_i$</td>
<td>$B_i$</td>
<td>$A_i$</td>
</tr>
</tbody>
</table>

Table 6.1: Texture, table, and filter return values. $R_i$, $G_i$, $B_i$, and $A_i$ are components of the internal format that are assigned to pixel values R, G, B, and A. If a requested pixel value is not present in the internal format, the specified constant value is used.

Basic machine units needed to store in memory the GL data type corresponding to type (see Table 3.2), an INVALID_OPERATION error results.

The command

```c
void GetCompressedTexImage( enum target, int lod, 
                          void *img );
```

is used to obtain texture images stored in compressed form. The parameters `target`, `lod`, and `img` are interpreted in the same manner as in `GetTexImage`. When called, `GetCompressedTexImage` writes $n$ ubytes of compressed image data to the pixel pack buffer or client memory pointed to by `img`, where $n$ is the value of `TEXTURE_COMPRESSED_IMAGE_SIZE` for the texture. The compressed image data is formatted according to the definition of the texture’s internal format.

By default the pixel storage modes `PACK_ROW_LENGTH`, `PACK_SKIP_ROWS`, `PACK_SKIP_PIXELS`, `PACK_IMAGE_HEIGHT` and `PACK_SKIP_IMAGES` are ignored for compressed images. To enable `PACK_SKIP_PIXELS` and `PACK_ROW_LENGTH`, the values of `PACK_COMPRESSED_BLOCK_SIZE` and `PACK_COMPRESSED_BLOCK_WIDTH` must both be non-zero. To also enable `PACK_SKIPROWS` and `PACK_IMAGE_HEIGHT`, the value of `PACK_COMPRESSED_BLOCK_HEIGHT` must be non-zero. And to also enable `PACK_SKIP_IMAGES`, the value of `PACK_COMPRESSED_BLOCK_DEPTH` must be non-zero. All parameters must be consistent with the compressed format to produce the desired results. When the pixel storage modes are active, the correspondence of texels to memory locations is as defined for `CompressedTexImage3D` in section 3.9.5.

Calling `GetCompressedTexImage` with an `lod` value less than zero or greater than the maximum allowable causes an `INVALID_VALUE` error. Calling `GetCompressedTexImage` with a texture image stored with an uncompressed internal format causes an `INVALID_OPERATION` error. If a pixel pack buffer object is bound
and \(img + n\) is greater than the size of the buffer, an \texttt{INVALID\_OPERATION} error results.

The command

\begin{verbatim}
boolean IsTexture( uint texture );
\end{verbatim}

returns \texttt{TRUE} if \textit{texture} is the name of a texture object. If \textit{texture} is zero, or is a non-zero value that is not the name of a texture object, or if an error condition occurs, \texttt{IsTexture} returns \texttt{FALSE}. A name returned by \texttt{GenTextures}, but not yet bound, is not the name of a texture object.

### 6.1.5 Sampler Queries

The command

\begin{verbatim}
boolean IsSampler( uint sampler );
\end{verbatim}

may be called to determine whether \textit{sampler} is the name of a sampler object. \texttt{IsSampler} will return \texttt{TRUE} if \textit{sampler} is the name of a sampler object previously returned from a call to \texttt{GenSamplers} and \texttt{FALSE} otherwise. Zero is not the name of a sampler object.

The current values of the parameters of a sampler object may be queried by calling

\begin{verbatim}
void GetSamplerParameter\{if\}v( uint sampler,
   enum pname, T *params );
void GetSamplerParameter\{iui\}v( uint sampler,
   enum pname, T *params );
\end{verbatim}

\textit{sampler} is the name of the sampler object from which to retrieve parameters. \textit{pname} is the name of the parameter to be queried. \textit{params} is the address of an array into which the current value of the parameter will be placed. \texttt{GetSamplerParameter*} accepts the same values for \textit{pname} as \texttt{SamplerParameter*} (see section 3.9.2). An \texttt{INVALID\_OPERATION} error is generated if \textit{sampler} is not the name of a sampler object previously returned from a call to \texttt{GenSamplers}. An \texttt{INVALID\_ENUM} error is generated if \textit{pname} is not the name of a parameter accepted by \texttt{GetSamplerParameter*}.

Querying \texttt{TEXTURE\_BORDER\_COLOR} with \texttt{GetSamplerParameter\{iv\}} or \texttt{GetSamplerParameter\{iui\}} returns the border color values as signed integers or unsigned integers, respectively; otherwise the values are returned as described in section 6.1.2. If the border color is queried with a type that does not match the original type with which it was specified, the result is undefined.
6.1. QUERYING GL STATE

6.1.6 String Queries

String queries return pointers to UTF-8 encoded, null-terminated static strings describing properties of the current GL context. The command

```c
ubyte *GetString( enum name );
```

accepts name values of RENDERER, VENDOR, VERSION, and SHADINGLANGUAGE_VERSION. The format of the RENDERER and VENDOR strings is implementation-dependent. The VERSION and SHADING_LANGUAGE_VERSION strings are laid out as follows:

```
<version number><space><vendor-specific information>
```

The version number is either of the form major_number.minor_number or major_number.minor_number.release_number, where the numbers all have one or more digits. The minor_number for SHADING_LANGUAGE_VERSION is always two digits, matching the OpenGL Shading Language Specification release number. For example, this query might return the string "4.20" while the corresponding VERSION query returns "4.2". The release_number and vendor specific information are optional. However, if present, then they pertain to the server and their format and contents are implementation-dependent.

GetString returns the version number (in the VERSION string) that can be supported by the current GL context. Thus, if the client and server support different versions a compatible version is returned.

The version, profile, and additional properties of the context may also be queried by calling GetInteger with values MAJOR_VERSION and MINOR_VERSION, which respectively return the same values as major_number and minor_number in the VERSION string; with value CONTEXT_PROFILE_MASK, which returns a mask containing one of the bits in table 6.2, corresponding to the API profile implemented by the context (see appendix E.1); or with value CONTEXT_FLAGS, which returns a set of flags defining additional properties of a context. If CONTEXT_FLAG_FORWARD_COMPATIBLE_BIT is set in CONTEXT_FLAGS, then the context is a forward-compatible context as defined in appendix E, and the deprecated features described in that appendix are not supported; otherwise the context is a full context, and all features described in the specification are supported.

Indexed strings are queried with the command

```c
ubyte *GetStringi( enum name, uint index );
```

Applications making copies of these static strings should never use a fixed-length buffer, because the strings may grow unpredictably between releases, resulting in buffer overflow when copying.
### 6.1. QUERYING GL STATE

<table>
<thead>
<tr>
<th>Value</th>
<th>OpenGL Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONTEXT_CORE_PROFILE_BIT</td>
<td>Core</td>
</tr>
<tr>
<td>CONTEXT_COMPATIBILITY_PROFILE_BIT</td>
<td>Compatibility</td>
</tr>
</tbody>
</table>

Table 6.2: Context profile bits returned by the CONTEXT_PROFILE_MASK query.

- name is the name of the indexed state and index is the index of the particular element being queried. name may only be EXTENSIONS, indicating that the extension name corresponding to the indexth supported extension should be returned. index may range from zero to the value of NUM_EXTENSIONS minus one. There is no defined relationship between any particular extension name and the index values; an extension name may correspond to a different index in different GL contexts and/or implementations.

  An INVALID_VALUE error is generated if index is outside the valid range for the indexed state name.

#### 6.1.7 Asynchronous Queries

The command

```java
boolean IsQuery( uint id );
```

returns TRUE if id is the name of a query object. If id is zero, or if id is a non-zero value that is not the name of a query object, IsQuery returns FALSE.

Information about an indexed query target can be queried with the commands

```java
void GetQueryIndexediv( enum target, uint index, 
   enum pname, int *params );
void GetQueryiv( enum target, enum pname, int *params );
```

- target identifies the query target, and must be one of SAMPLES_PASSED or ANY_SAMPLES_PASSED for occlusion queries, TIME_ELAPSED or TIMESTAMP for timer queries, or PRIMITIVES_GENERATED or TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN for primitive queries. index is the index of the query target, and must be between zero and a target-specific maximum. If index is outside of this range, the error INVALID_VALUE is generated. Calling GetQueryiv is equivalent to calling GetQueryIndexediv with index set to zero.

  If pname is CURRENT_QUERY, the name of the currently active query for the specified index of target, or zero if no query is active for that index of target, will be placed in params.
6.1. QUERYING GL STATE

If `pname` is `QUERY_COUNTER_BITS`, `index` is ignored and the implementation-dependent number of bits used to hold the query result for `target` will be placed in `params`. The number of query counter bits may be zero, in which case the counter contains no useful information.

For primitive queries (`PRIMITIVES_GENERATED` and `TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN`) if the number of bits is non-zero, the minimum number of bits allowed is 32.

For occlusion queries with `target` `ANY_SAMPLES_PASSED`, if the number of bits is non-zero, the minimum number of bits is 1. For occlusion queries with `target` `SAMPLES_PASSED`, if the number of bits is non-zero, the minimum number of bits allowed is 32, a function of the implementation’s maximum viewport dimensions (`MAX_VIEWPORT_DIMS`). The counter must be able to represent at least two overdraws for every pixel in the viewport. The formula to compute the allowable minimum value (where `n` is the minimum number of bits) is:

\[ \text{allowable minimum value} = 2^{n} - 1 \]

For timer queries (`target` `TIME_ELAPSED` and `TIMESTAMP`), if the number of bits is non-zero, the minimum number of bits allowed is 30. This will allow at least 1 second of timing.

The state of a query object can be queried with the commands

```c
void GetQueryObjectiv( uint id, enum pname, int *params );
void GetQueryObjectuiv( uint id, enum pname, uint *params );
void GetQueryObjecti64v( uint id, enum pname, int64 *params );
void GetQueryObjectui64v( uint id, enum pname, uint64 *params );
```

If `id` is not the name of a query object, or if the query object named by `id` is currently active, then an `INVALID_OPERATION` error is generated. `pname` must be `QUERY_RESULT` or `QUERY_RESULT_AVAILABLE`.

If `pname` is `QUERY_RESULT`, then the query object’s result value is returned as a single integer in `params`. If the value is so large in magnitude that it cannot be represented with the requested type, then the nearest value representable using the requested type is returned. If the number of query counter bits for `target` is zero, then the result is returned as a single integer with the value zero.

There may be an indeterminate delay before the above query returns. If `pname` is `QUERY_RESULT_AVAILABLE`, `FALSE` is returned if such a delay would be required; otherwise `TRUE` is returned. It must always be true that if any query object returns a result available of `TRUE`, all queries of the same type issued prior to that query must also return `TRUE`.

OpenGL 4.2 (Core Profile) - April 27, 2012
6.1. QUERYING GL STATE

Querying the state for any given query object forces that occlusion query to complete within a finite amount of time.

If multiple queries are issued using the same object name prior to calling GetQueryObject*, the result and availability information returned will always be from the last query issued. The results from any queries before the last one will be lost if they are not retrieved before starting a new query on the same target and id.

6.1.8 Sync Object Queries

Properties of sync objects may be queried using the command

```c
void GetSynciv( sync sync, enum pname, sizei bufsize,
        sizei *length, int *values );
```

The value or values being queried are returned in the parameters `length` and `values`.

On success, GetSynciv replaces up to `bufSize` integers in `values` with the corresponding property values of the object being queried. The actual number of integers replaced is returned in `*length`. If `length` is `NULL`, no length is returned.

If `pname` is `OBJECT_TYPE`, a single value representing the specific type of the sync object is placed in `values`. The only type supported is `SYNC_FENCE`.

If `pname` is `SYNC_STATUS`, a single value representing the status of the sync object (`SIGNALED` or `UNSIGNALED`) is placed in `values`.

If `pname` is `SYNC_CONDITION`, a single value representing the condition of the sync object is placed in `values`. The only condition supported is `SYNC_GPU_COMMANDS_COMPLETE`.

If `pname` is `SYNC_FLAGS`, a single value representing the flags with which the sync object was created is placed in `values`. No flags are currently supported.

If `sync` is not the name of a sync object, an `INVALID_VALUE` error is generated. If `pname` is not one of the values described above, an `INVALID_ENUM` error is generated. If an error occurs, nothing will be written to `values` or `length`.

The command

```c
boolean IsSync( sync sync );
```

returns `TRUE` if `sync` is the name of a sync object. If `sync` is not the name of a sync object, or if an error condition occurs, `IsSync` returns `FALSE` (note that zero is not the name of a sync object).

Sync object names immediately become invalid after calling DeleteSync, as discussed in sections 5.3 and D.1, but the underlying sync object will not be deleted until it is no longer associated with any fence command and no longer blocking any *WaitSync command.

OpenGL 4.2 (Core Profile) - April 27, 2012
6.1.9 Buffer Object Queries

The command

```c
boolean IsBuffer(uint buffer);
```

returns TRUE if `buffer` is the name of an buffer object. If `buffer` is zero, or if `buffer` is a non-zero value that is not the name of an buffer object, `IsBuffer` returns FALSE.

The commands

```c
void GetBufferParameteriv( enum target, enum pname, int *data );
void GetBufferParameteri64v( enum target, enum pname, int64 *data );
```

return information about a bound buffer object. `target` must be one of the targets listed in table 2.8, and `pname` must be one of the buffer object parameters in table 2.9, other than `BUFFER_MAP_POINTER`. The value of the specified parameter of the buffer object bound to `target` is returned in `data`.

The command

```c
void GetBufferSubData( enum target, intptr offset, sizeiptr size, void *data );
```

queries the data contents of a buffer object. `target` must be one of the targets listed in table 2.8. `offset` and `size` indicate the range of data in the buffer object that is to be queried, in terms of basic machine units. `data` specifies a region of client memory, `size` basic machine units in length, into which the data is to be retrieved.

An error is generated if `GetBufferSubData` is executed for a buffer object that is currently mapped.

While the data store of a buffer object is mapped, the pointer to the data store can be queried by calling

```c
void GetBufferPointerv( enum target, enum pname, void **params );
```

with `target` set to one of the targets listed in table 2.8 and `pname` set to `BUFFER_MAP_POINTER`. The single buffer map pointer is returned in `params`. `GetBufferPointerv` returns the NULL pointer value if the buffer’s data store is not currently mapped, or if the requesting client did not map the buffer object’s data store, and the implementation is unable to support mappings on multiple clients.
6.1. QUERYING GL STATE

Uniform Buffer Object Binding Queries

To query which buffer objects are bound to the array of uniform buffer binding points and will be used as the storage for active uniform blocks, call \texttt{GetIntegeri} with \textit{param} set to \texttt{UNIFORM_BUFFER_BINDING}. \textit{index} must be in the range zero to the value of \texttt{MAX_UNIFORM_BUFFER_BINDINGS} minus one. The name of the buffer object bound to \textit{index} is returned in \textit{values}. If no buffer object is bound for \textit{index}, zero is returned in \textit{values}.

To query the starting offset or size of the range of each buffer object binding used for uniform buffers, call \texttt{GetInteger64i} with \textit{param} set to \texttt{UNIFORM_BUFFER_START} or \texttt{UNIFORM_BUFFER_SIZE} respectively. \textit{index} must be in the range zero to the value of \texttt{MAX_UNIFORM_BUFFER_BINDINGS} minus one. If the parameter (starting offset or size) was not specified when the buffer object was bound (e.g. if bound with \texttt{BindBufferBase}), or if no buffer object is bound to \textit{index}, zero is returned.

Transform Feedback Buffer Object Binding Queries

To query which buffer objects are bound to the array of transform feedback binding points and will be used when transform feedback is active, call \texttt{GetIntegeri} with \textit{param} set to \texttt{TRANSFORM_FEEDBACK_BUFFER_BINDING}. \textit{index} must be in the range zero to the value of \texttt{MAX_TRANSFORM_FEEDBACK_BUFFERS} minus one. The name of the buffer object bound to \textit{index} is returned in \textit{values}. If no buffer object is bound for \textit{index}, zero is returned in \textit{values}. The error INVALID_VALUE is generated if \textit{index} is greater than or equal to the value of \texttt{MAX_TRANSFORM_FEEDBACK_BUFFERS}.

To query the starting offset or size of the range of each buffer object binding used for transform feedback, call \texttt{GetInteger64i} with \textit{param} set to \texttt{TRANSFORM_FEEDBACK_BUFFER_START} or \texttt{TRANSFORM_FEEDBACK_BUFFER_SIZE} respectively. \textit{index} must be in the range 0 to the value of \texttt{MAX_TRANSFORM_FEEDBACK_BUFFERS} minus one. If the parameter (starting offset or size) was not specified when the buffer object was bound (e.g. if bound with \texttt{BindBufferBase}), or if no buffer object is bound to \textit{index}, zero is returned. The error INVALID_VALUE is generated if \textit{index} is greater than or equal to the value of \texttt{MAX_TRANSFORM_FEEDBACK_BUFFERS}.

Atomic Counter Buffer Object Binding Queries

To query which buffer objects are bound to the array of atomic counter buffer binding points and will be used as the storage for atomic counter buffers, call \texttt{GetIntegeri} with \textit{param} set to \texttt{ATOMIC_COUNTER_BUFFER_BINDING}. \textit{index} must
be in the range zero to the value of `MAX_ATOMIC_COUNTER_BUFFER_BINDINGS` minus one. The name of the buffer object bound to `index` is returned in `values`. If no buffer object is bound for `index`, zero is returned in `values`. The error `INVALID_VALUE` is generated if `index` is greater than or equal to the value of `MAX_ATOMIC_COUNTER_BUFFER_BINDINGS`.

To query the starting offset or size of the range of each buffer object binding used for atomic counters, call `GetInteger64i` with `param` set to `ATOMIC_COUNTER_BUFFER_START` or `ATOMIC_COUNTER_BUFFER_SIZE` respectively. `index` must be in the range 0 to the value of `MAX_ATOMIC_COUNTER_BUFFER_BINDINGS` minus one. If the parameter (starting offset or size) was not specified when the buffer object was bound (e.g. if bound with `BindBufferBase`), or if no buffer object is bound to `index`, zero is returned. The error `INVALID_VALUE` is generated if `index` is greater than or equal to the value of `MAX_ATOMIC_COUNTER_BUFFER_BINDINGS`.

6.1.10 Vertex Array Object Queries

The command

```c
boolean IsVertexArray( uint array );
```

returns `TRUE` if `array` is the name of a vertex array object. If `array` is zero, or a non-zero value that is not the name of a vertex array object, `IsVertexArray` returns `FALSE`. No error is generated if `array` is not a valid vertex array object name.

6.1.11 Transform Feedback Queries

The command

```c
boolean IsTransformFeedback( uint id );
```

returns `TRUE` if `id` is the name of a transform feedback object. If `id` is zero, or a non-zero value that is not the name of a transform feedback object, `IsTransformFeedback` returns `FALSE`. No error is generated if `id` is not a valid transform feedback object name.

6.1.12 Shader and Program Queries

State stored in shader or program objects can be queried by commands that accept shader or program object names. These commands will generate the error...
6.1. QUERYING GL STATE

INVALID_VALUE if the provided name is not the name of either a shader or program object, and INVALID_OPERATION if the provided name identifies an object of the other type. If an error is generated, variables used to hold return values are not modified.

The command

```cpp
boolean IsShader(uint shader);
```

returns TRUE if `shader` is the name of a shader object. If `shader` is zero, or a non-zero value that is not the name of a shader object, `IsShader` returns FALSE. No error is generated if `shader` is not a valid shader object name.

The command

```cpp
void GetShaderiv(uint shader, enum pname, int *params);
```

returns properties of the shader object named `shader` in `params`. The parameter value to return is specified by `pname`.

If `pname` is SHADER_TYPE, VERTEX_SHADER, TESS_CONTROL_SHADER, TESS_EVALUATION_SHADER, GEOMETRY_SHADER, or FRAGMENT_SHADER is returned if `shader` is a vertex, tessellation control, tessellation evaluation, geometry, or fragment shader object respectively.

If `pname` is DELETE_STATUS, TRUE is returned if the shader has been flagged for deletion and FALSE is returned otherwise.

If `pname` is COMPILE_STATUS, TRUE is returned if the shader was last compiled successfully, and FALSE is returned otherwise.

If `pname` is INFO_LOG_LENGTH, the length of the info log, including a null terminator, is returned. If there is no info log, zero is returned.

If `pname` is SHADER_SOURCE_LENGTH, the length of the concatenation of the source strings making up the shader source, including a null terminator, is returned. If no source has been defined, zero is returned.

The command

```cpp
boolean IsProgram(uint program);
```

returns TRUE if `program` is the name of a program object. If `program` is zero, or a non-zero value that is not the name of a program object, `IsProgram` returns FALSE. No error is generated if `program` is not a valid program object name.

The command

```cpp
void GetProgramiv(uint program, enum pname, int *params);
```

OpenGL 4.2 (Core Profile) - April 27, 2012
returns properties of the program object named \textit{program} in \textit{params}. The parameter value to return is specified by \textit{pname}.

If \textit{pname} is \texttt{DELETE\_STATUS}, \texttt{TRUE} is returned if the program has been flagged for deletion, and \texttt{FALSE} is returned otherwise.

If \textit{pname} is \texttt{LINK\_STATUS}, \texttt{TRUE} is returned if the program was last compiled successfully, and \texttt{FALSE} is returned otherwise.

If \textit{pname} is \texttt{VALIDATE\_STATUS}, \texttt{TRUE} is returned if the last call to \texttt{ValidateProgram} with \textit{program} was successful, and \texttt{FALSE} is returned otherwise.

If \textit{pname} is \texttt{INFO\_LOG\_LENGTH}, the length of the info log, including a null terminator, is returned. If there is no info log, zero is returned.

If \textit{pname} is \texttt{ATTACHED\_SHADERS}, the number of objects attached is returned.

If \textit{pname} is \texttt{ACTIVE\_ATTRIBUTES}, the number of active attributes in \textit{program} is returned. If no active attributes exist, zero is returned.

If \textit{pname} is \texttt{INFO\_LOG\_LENGTH}, the length of the longest active attribute name, including a null terminator, is returned. If no active attributes exist, zero is returned.

If \textit{pname} is \texttt{ACTIVE\_UNIFORMS}, the number of active uniforms is returned. If no active uniforms exist, zero is returned.

If \textit{pname} is \texttt{ACTIVE\_ATTRIBUTES\_MAX\_LENGTH}, the length of the longest active attribute name, including a null terminator, is returned. If no active attributes exist, zero is returned.

If \textit{pname} is \texttt{ACTIVE\_UNIFORM\_MAX\_LENGTH}, the length of the longest active uniform name, including a null terminator, is returned. If no active uniforms exist, zero is returned.

If \textit{pname} is \texttt{TRANSFORM\_FEEDBACK\_BUFFER\_MODE}, the buffer mode used when transform feedback is active is returned. It can be one of \texttt{SEPARATE\_ATTRIBS} or \texttt{INTERLEAVED\_ATTRIBS}.

If \textit{pname} is \texttt{TRANSFORM\_FEEDBACK\_VARYINGS}, the number of output variables to capture in transform feedback mode for the program is returned.

If \textit{pname} is \texttt{TRANSFORM\_FEEDBACK\_VARYING\_MAX\_LENGTH}, the length of the longest output variable name specified to be used for transform feedback, including a null terminator, is returned. If no outputs are used for transform feedback, zero is returned.

If \textit{pname} is \texttt{ACTIVE\_UNIFORM\_BLOCKS}, the number of uniform blocks for \textit{program} containing active uniforms is returned.

If \textit{pname} is \texttt{ACTIVE\_UNIFORM\_BLOCK\_MAX\_NAME\_LENGTH}, the length of the longest active uniform block name, including the null terminator, is returned.

If \textit{pname} is \texttt{GEOMETRY\_VERTICES\_OUT}, the maximum number of vertices the geometry shader will output is returned.

If \textit{pname} is \texttt{GEOMETRY\_INPUT\_TYPE}, the geometry shader input type, which must be one of \texttt{POINTS}, \texttt{LINES}, \texttt{LINES\_ADJACENCY}, \texttt{TRIANGLES} or \texttt{TRIANGLES\_ADJACENCY}, is returned.
If `pname` is `GEOMETRY_OUTPUT_TYPE`, the geometry shader output type, which must be one of `POINTS`, `LINE_STRIP` or `TRIANGLE_STRIP`, is returned.

If `pname` is `GEOMETRY_SHADER_INVOCATIONS`, the number of geometry shader invocations per primitive will be returned.

If `GEOMETRY_VERTICES_OUT`, `GEOMETRY_INPUT_TYPE`, `GEOMETRY_OUTPUT_TYPE`, or `GEOMETRY_SHADER_INVOCATIONS` are queried for a program which has not been linked successfully, or which does not contain objects to form a geometry shader, then an `INVALID_OPERATION` error is generated.

If `pname` is `TESS_CONTROL_OUTPUT_VERTICES`, the number of vertices in the tessellation control shader output patch is returned. If `TESS_CONTROL_OUTPUT_VERTICES` is queried for a program which has not been linked successfully, or which does not contain objects to form a tessellation control shader, then an `INVALID_OPERATION` error is generated.

If `pname` is `TESS_GEN_MODE`, `QUADS`, `TRIANGLES`, or `ISOLINES` is returned, depending on the primitive mode declaration in the tessellation evaluation shader. If `pname` is `TESS_GEN_SPACING`, `EQUAL`, `FRACTIONAL_EVEN`, or `FRACTIONAL_ODD` is returned, depending on the spacing declaration in the tessellation evaluation shader. If `pname` is `TESS_GEN_VERTEX_ORDER`, `CCW` or `CW` is returned, depending on the vertex order declaration in the tessellation evaluation shader. If `pname` is `TESS_GEN_POINT_MODE`, `TRUE` is returned if point mode is enabled in a tessellation evaluation shader declaration; `FALSE` is returned otherwise. If any of the `pname` values in this paragraph are queried for a program which has not been linked successfully, or which does not contain objects to form a tessellation evaluation shader, then an `INVALID_OPERATION` error is generated.

If `pname` is `PROGRAM_SEPARABLE`, `TRUE` is returned if the program has been flagged for use as a separable program object that can be bound to individual shader stages with `UseProgramStages`.

If `pname` is `PROGRAM_BINARY_RETRIEVABLE_HINT`, the current value of whether the binary retrieval hint is enabled for `program` is returned.

If `pname` is `ACTIVE_ATOMIC_COUNTER_BUFFERS`, the number of active atomic counter buffers used by `program` is returned.

The command

```c
boolean IsProgramPipeline( uint pipeline );
```

returns `TRUE` if `pipeline` is the name of a program pipeline object. If `pipeline` is zero, or a non-zero value that is not the name of a program pipeline object, `IsProgramPipeline` returns `FALSE`. No error is generated if `pipeline` is not a valid program pipeline object name.

The command

OpenGL 4.2 (Core Profile) - April 27, 2012
6.1. QUERYING GL STATE

```c
void GetProgramPipelineiv( uint pipeline, enum pname, 
    int *params );
```

returns properties of the program pipeline object named `pipeline` in `params`. The parameter value to return is specified by `pname`.

If `pipeline` is a name that has been generated (without subsequent deletion) by `GenProgramPipelines`, but refers to a program pipeline object that has not been previously bound, the GL first creates a new state vector in the same manner as when `BindProgramPipeline` creates a new program pipeline object. If `pipeline` is not a name returned from a previous call to `GenProgramPipelines` or if such a name has since been deleted by `DeleteProgramPipelines`, an `INVALID_OPERATION` error is generated.

If `pname` is `ACTIVE_PROGRAM`, the name of the active program object of the program pipeline object is returned.

If `pname` is `VERTEX_SHADER`, the name of the current program object for the vertex shader type of the program pipeline object is returned.

If `pname` is `FRAGMENT_SHADER`, the name of the current program object for the fragment shader type of the program pipeline object is returned.

If `pname` is `GEOMETRY_SHADER`, the name of the current program object for the geometry shader type of the program pipeline object is returned;

If `pname` is `TESS_CONTROL_SHADER`, the name of the current program object for the tessellation control shader type of the program pipeline object is returned;

If `pname` is `TESS_EVALUATION_SHADER`, the name of the current program object for the tessellation evaluation shader type of the program pipeline object is returned.

If `pname` is `VALIDATE_STATUS`, the validation status of the program pipeline object, as determined by `ValidateProgramPipeline` (see section 2.11.12) is returned.

If `pname` is `INFO_LOG_LENGTH`, the length of the info log, including a null terminator, is returned. If there is no info log, zero is returned.

If `pname` is not the name of an accepted parameter, an `INVALID_ENUM` error is generated.

The command

```c
void GetAttachedShaders( uint program, sizei maxCount, 
    sizei *count, uint *shaders );
```

returns the names of shader objects attached to `program` in `shaders`. The actual number of shader names written into `shaders` is returned in `count`. If no shaders are attached, `count` is set to zero. If `count` is `NULL` then it is ignored. The maximum
number of shader names that may be written into shaders is specified by maxCount. The number of objects attached to program is given by can be queried by calling GetProgramiv with ATTACHED_SHADERS.

A string that contains information about the last compilation attempt on a shader object, last link or validation attempt on a program object, or last validation attempt on a program pipeline object, called the info log, can be obtained with the commands

```c
void GetShaderInfoLog( uint shader, sizei bufSize,
    sizei *length, char *infoLog );
void GetProgramInfoLog( uint program, sizei bufSize,
    sizei *length, char *infoLog );
void GetProgramPipelineInfoLog( uint pipeline,
    sizei bufSize, sizei *length, char *infoLog );
```

These commands return an info log string for the corresponding type of object in infoLog. This string will be null-terminated. The actual number of characters written into infoLog, excluding the null terminator, is returned in length. If length is NULL, then no length is returned. The maximum number of characters that may be written into infoLog, including the null terminator, is specified by bufSize. The number of characters in the info log for a shader object, program object, or program pipeline object can be queried respectively with GetShaderiv, GetProgramiv, or GetProgramPipelineiv with pname INFO_LOG_LENGTH.

If shader is a shader object, GetShaderInfoLog will return either an empty string or information about the last compilation attempt for that object. If program is a program object, GetProgramInfoLog will return either an empty string or information about the last link attempt or last validation attempt for that object. If pipeline is a program pipeline object, GetProgramPipelineInfoLog will return either an empty string or information about the last validation attempt for that object.

The info log is typically only useful during application development and an application should not expect different GL implementations to produce identical info logs.

The command

```c
void GetShaderSource( uint shader, sizei bufSize,
    sizei *length, char *source );
```

returns in source the string making up the source code for the shader object shader. The string source will be null-terminated. The actual number of characters written into source, excluding the null terminator, is returned in length. If length is NULL,
6.1. QUERYING GL STATE

no length is returned. The maximum number of characters that may be written into source, including the null terminator, is specified by bufSize. The string source is a concatenation of the strings passed to the GL using ShaderSource. The length of this concatenation is given by SHADER_SOURCE_LENGTH, which can be queried with GetShaderiv.

The command

```c
void GetShaderPrecisionFormat( enum shadertype,
    enum precisiontype, int *range, int *precision);
```

returns the range and precision for different numeric formats supported by the shader compiler. shadertype must be VERTEX_SHADER or FRAGMENT_SHADER. precisiontype must be one of LOW_FLOAT, MEDIUM_FLOAT, HIGH_FLOAT, LOW_INT, MEDIUM_INT or HIGH_INT. range points to an array of two integers in which encodings of the format’s numeric range are returned. If min and max are the smallest and largest values representable in the format, then the values returned are defined to be

\[
\begin{align*}
    range[0] &= \lfloor \log_2(|min|) \rfloor \\
    range[1] &= \lfloor \log_2(|max|) \rfloor
\end{align*}
\]

precision points to an integer in which the \( \log_2 \) value of the number of bits of precision of the format is returned. If the smallest representable value greater than 1 is \( 1 + \epsilon \), then *precision will contain \( [-\log_2(\epsilon)] \), and every value in the range

\([-2^{range[0]}, 2^{range[1]}]\)

can be represented to at least one part in \( 2^{\text{precision}} \). For example, an IEEE single-precision floating-point format would return \( range[0] = 127, range[1] = 127 \), and *precision = 23, while a 32-bit two’s-complement integer format would return \( range[0] = 31, range[1] = 30 \), and *precision = 0.

The minimum required precision and range for formats corresponding to the different values of precisiontype are described in section 4.5 of the OpenGL Shading Language specification.

The commands

```c
void GetVertexAttribdv( uint index, enum pname,
    double *params);
void GetVertexAttribfv( uint index, enum pname,
    float *params);
```
void GetVertexAttribiv( uint index, enum pname, int *params );
void GetVertexAttribIiv( uint index, enum pname, int *params );
void GetVertexAttribIuiv( uint index, enum pname, uint *params );
void GetVertexAttribLdv( uint index, enum pname, double *params );

obtain the vertex attribute state named by pname for the generic vertex attribute numbered index and places the information in the array params. pname must be one of VERTEX_ATTRIB_ARRAY_BUFFER_BINDING, VERTEX_ATTRIB_ARRAY_ENABLED, VERTEX_ATTRIB_ARRAY_SIZE, VERTEX_ATTRIB_ARRAY_STRIDE, VERTEX_ATTRIB_ARRAY_TYPE, VERTEX_ATTRIB_ARRAY_NORMALIZED, VERTEX_ATTRIB_ARRAY_INTEGER, VERTEX_ATTRIB_ARRAY_DIVISOR, or CURRENT_VERTEX_ATTRIB. Note that all the queries except CURRENT_VERTEX_ATTRIB return values stored in the currently bound vertex array object (the value of VERTEX_ARRAY_BINDING).

An INVALID_VALUE error is generated if index is greater than or equal to MAX_VERTEX_ATTRIBS.

If no vertex array object is bound (see section 2.10), an INVALID_OPERATION error is generated.

All but CURRENT_VERTEX_ATTRIB return information about generic vertex attribute arrays. The enable state of a generic vertex attribute array is set by the command EnableVertexAttribArray and cleared by DisableVertexAttribArray. The size, stride, type, normalized flag, and unconverted integer flag are set by the commands VertexAttribPointer and VertexAttribIPointer. The normalized flag is always set to FALSE by VertexAttribPointer. The unconverted integer flag is always set to FALSE by VertexAttribPointer and TRUE by VertexAttribIPointer.

The query CURRENT_VERTEX_ATTRIB returns the current value for the generic attribute index. GetVertexAttribdv and GetVertexAttribfv read and return the current attribute values as four floating-point values; GetVertexAttribiv reads them as floating-point values and converts them to four integer values; GetVertexAttribIiv reads and returns them as four signed integers; GetVertexAttribIuiv reads and returns them as four unsigned integers; and GetVertexAttribLdv reads and returns them as four double-precision floating-point values. The results of the query are undefined if the current attribute values are read using one data type but were specified using a different one.

The command
void GetVertexAttribPointerv( uint index, enum pname, void **pointer );

obtains the pointer named pname for the vertex attribute numbered index and places the information in the array pointer. pname must be VERTEX_ATTRIB_ARRAY_-POINTER. The value returned is queried from the currently bound vertex array object.

An INVALID_VALUE error is generated if index is greater than or equal to the value of MAX_VERTEX_ATTRIBS.

If no vertex array object is bound (see section 2.10), an INVALID_OPERATION error is generated.

The commands

void GetUniformfv( uint program, int location, float *params );
void GetUniformdv( uint program, int location, double *params );
void GetUniformiv( uint program, int location, int *params );
void GetUniformuiv( uint program, int location, uint *params );

return the value or values of the uniform at location location of the default uniform block for program object program in the array params. The type of the uniform at location determines the number of values returned. The error INVALID_OPERATION is generated if program has not been linked successfully, or if location is not a valid location for program. In order to query the values of an array of uniforms, a GetUniform* command needs to be issued for each array element. If the uniform queried is a matrix, the values of the matrix are returned in column major order. If an error occurred, params will not be modified.

The command

void GetUniformSubroutineuiv( enum shadertype, int location, uint *params );

returns the value of the subroutine uniform at location location for shader stage shadertype of the current program. If location is greater than or equal to the value of ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS for the shader currently in use at shader stage shadertype, the error INVALID_VALUE is generated. If no program is active, the error INVALID_OPERATION is generated.

The command

OpenGL 4.2 (Core Profile) - April 27, 2012
6.1. QUERYING GL STATE

```c
void GetProgramStageiv( uint program, enum shadertype, 
    enum pname, int *values );
```

returns properties of the program object `program` specific to the programmable stage corresponding to `shadertype` in `values`. The parameter value to return is specified by `pname`. If `pname` is `ACTIVE_SUBROUTINE_UNIFORMS`, the number of active subroutine variables in the stage is returned. If `pname` is `ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS`, the number of active subroutine variable locations in the stage is returned. If `pname` is `ACTIVE_SUBROUTINES`, the number of active subroutines in the stage is returned. If `pname` is `ACTIVE_SUBROUTINE_UNIFORM_MAX_LENGTH` or `ACTIVE_SUBROUTINE_MAX_LENGTH`, the length of the longest subroutine uniform or subroutine name, respectively, for the stage is returned. The returned name length includes space for a null terminator. If there is no shader of type `shadertype` in `program`, the values returned will be consistent with a shader with no subroutines or subroutine uniforms.

6.1.13 Framebuffer Object Queries

The command

```c
boolean IsFramebuffer( uint framebuffer );
```

returns `TRUE` if `framebuffer` is the name of an framebuffer object. If `framebuffer` is zero, or if `framebuffer` is a non-zero value that is not the name of an framebuffer object, `IsFramebuffer` returns `FALSE`.

The command

```c
void GetFramebufferAttachmentParameteriv( enum target, 
    enum attachment, enum pname, int *params );
```

returns information about attachments of a bound framebuffer object. `target` must be `DRAW_FRAMEBUFFER`, `READ_FRAMEBUFFER`, or `FRAMEBUFFER`. `FRAMEBUFFER` is equivalent to `DRAW_FRAMEBUFFER`.

If the default framebuffer is bound to `target`, then `attachment` must be one of `FRONT_LEFT`, `FRONT_RIGHT`, `BACK_LEFT`, or `BACK_RIGHT`, identifying a color buffer; `DEPTH`, identifying the depth buffer; or `STENCIL`, identifying the stencil buffer.

If a framebuffer object is bound to `target`, then `attachment` must be one of the attachment points of the framebuffer listed in table 4.11.

If `attachment` is `DEPTH_STENCIL_ATTACHMENT`, and different objects are bound to the depth and stencil attachment points of `target`, the query will fail and
6.1. QUERYING GL STATE

generate an INVALID_OPERATION error. If the same object is bound to both attachment points, information about that object will be returned.

Upon successful return from `GetFramebufferAttachmentParameteriv`, if `pname` is `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE`, then `param` will contain one of `NONE`, `FRAMEBUFFER_DEFAULT`, `TEXTURE`, or `RENDERBUFFER`, identifying the type of object which contains the attached image. Other values accepted for `pname` depend on the type of object, as described below.

If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is `NONE`, no framebuffer is bound to `target`. In this case querying `pname` `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` will return zero, and all other queries will generate an INVALID_OPERATION error.

If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is not `NONE`, these queries apply to all other framebuffer types:

- If `pname` is `FRAMEBUFFER_ATTACHMENT_RED_SIZE`, `FRAMEBUFFER_ATTACHMENT_GREEN_SIZE`, `FRAMEBUFFER_ATTACHMENT_BLUE_SIZE`, `FRAMEBUFFER_ATTACHMENT_ALPHA_SIZE`, `FRAMEBUFFER_ATTACHMENT_DEPTH_SIZE`, or `FRAMEBUFFER_ATTACHMENT_STENCIL_SIZE`, then `param` will contain the number of bits in the corresponding red, green, blue, alpha, depth, or stencil component of the specified `attachment`. Zero is returned if the requested component is not present in `attachment`.

- If `pname` is `FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE`, `param` will contain the format of components of the specified `attachment`, one of `FLOAT`, `INT`, `UNSIGNED_INT`, `SIGNED_NORMALIZED`, or `UNSIGNED_NORMALIZED` for floating-point, signed integer, unsigned integer, signed normalized fixed-point, or unsigned normalized fixed-point components respectively. Only color buffers may have integer components.

- If `pname` is `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING`, `param` will contain the encoding of components of the specified `attachment`, one of `LINEAR` or `SRGB` for linear or sRGB-encoded components, respectively. Only color buffer components may be sRGB-encoded; such components are treated as described in sections 4.1.7 and 4.1.8. For the default framebuffer, color encoding is determined by the implementation. For framebuffer objects, components are sRGB-encoded if the internal format of a color attachment is one of the color-renderable SRGB formats described in section 3.9.18.

If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is `RENDERBUFFER`, then

OpenGL 4.2 (Core Profile) - April 27, 2012
• If \texttt{pname} is \texttt{FRAMEBUFFER_ATTACHMENT_OBJECT_NAME}, \texttt{params} will contain the name of the renderbuffer object which contains the attached image.

If the value of \texttt{FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE} is \texttt{TEXTURE}, then

• If \texttt{pname} is \texttt{FRAMEBUFFER_ATTACHMENT_OBJECT_NAME}, then \texttt{params} will contain the name of the texture object which contains the attached image.

• If \texttt{pname} is \texttt{FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL}, then \texttt{params} will contain the mipmap level of the texture object which contains the attached image.

• If \texttt{pname} is \texttt{FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE} and the texture object named \texttt{FRAMEBUFFER_ATTACHMENT_OBJECT_NAME} is a cube map texture, then \texttt{params} will contain the cube map face of the cube-map texture object which contains the attached image. Otherwise \texttt{params} will contain the value zero.

• If \texttt{pname} is \texttt{FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER} and the texture object named \texttt{FRAMEBUFFER_ATTACHMENT_OBJECT_NAME} is a layer of a three-dimensional texture or a one-or two-dimensional array texture, then \texttt{params} will contain the number of the texture layer which contains the attached image. Otherwise \texttt{params} will contain the value zero.

• If \texttt{pname} is \texttt{FRAMEBUFFER_ATTACHMENT_LAYERED}, then \texttt{params} will contain \texttt{TRUE} if an entire level of a three-dimensional texture, cube map texture, or one-or two-dimensional array texture is attached. Otherwise, \texttt{params} will contain \texttt{FALSE}.

Any combinations of framebuffer type and \texttt{pname} not described above will generate an \texttt{INVALID_ENUM} error.

### 6.1.14 Renderbuffer Object Queries

The command

```plaintext
boolean IsRenderbuffer(uint renderbuffer);
```

returns \texttt{TRUE} if \texttt{renderbuffer} is the name of a renderbuffer object. If \texttt{renderbuffer} is zero, or if \texttt{renderbuffer} is a non-zero value that is not the name of a renderbuffer object, \texttt{IsRenderbuffer} returns \texttt{FALSE}.

The command
6.1. QUERYING GL STATE

```c
void GetRenderbufferParameteriv(enum target, enum pname, int* params);
```

returns information about a bound renderbuffer object. `target` must be `RENDERBUFFER` and `pname` must be one of the symbolic values in table 6.27. If the renderbuffer currently bound to `target` is zero, then an `INVALID_OPERATION` error is generated.

Upon successful return from `GetRenderbufferParameteriv`, if `pname` is `RENDERBUFFER_WIDTH`, `RENDERBUFFER_HEIGHT`, `RENDERBUFFER_INTERNAL_FORMAT`, or `RENDERBUFFER_SAMPLES`, then `params` will contain the width in pixels, height in pixels, internal format, or number of samples, respectively, of the image of the renderbuffer currently bound to `target`.

If `pname` is `RENDERBUFFER_RED_SIZE`, `RENDERBUFFER_GREEN_SIZE`, `RENDERBUFFER_BLUE_SIZE`, `RENDERBUFFER_ALPHA_SIZE`, `RENDERBUFFER_DEPTH_SIZE`, or `RENDERBUFFER_STENCIL_SIZE`, then `params` will contain the actual resolutions (not the resolutions specified when the image array was defined) for the red, green, blue, alpha depth, or stencil components, respectively, of the image of the renderbuffer currently bound to `target`.

Otherwise, an `INVALID_ENUM` error is generated.

6.1.15 Internal Format Queries

Information about implementation-dependent support for internal formats can be queried with the command

```c
void GetInternalformativ(enum target, enum internalformat, enum pname, sizei bufSize, int* params);
```

`internalformat` must be color-renderable, depth-renderable or stencil-renderable (as defined in section 4.4.4).

`target` indicates the usage of the `internalformat`, and must be one of `RENDERBUFFER`, `TEXTURE_2D_MULTISAMPLE`, or `TEXTURE_2D_MULTISAMPLE_ARRAY`, corresponding respectively to usage as a renderbuffer, two-dimensional multisample texture, or two-dimensional multisample array texture.

No more than `bufSize` integers will be written into `params`. If more data are available, they will be ignored and no error will be generated.

`pname` indicates the information to query, and is one of the following:

If `pname` is `NUM_SAMPLE_COUNTS`, the number of sample counts that would be returned by querying `SAMPLES` is returned in `params`.  

OpenGL 4.2 (Core Profile) - April 27, 2012
6.2. STATE TABLES

If \( \text{pname} \) is \text{SAMPLES}, the sample counts supported for \text{internalformat} and \text{target} are written into \text{params}, in descending numeric order. Only positive values are returned.

Querying \text{SAMPLES} with a \text{bufSize} of one will return just the maximum supported number of samples for this format.

The maximum value in \text{SAMPLES} is guaranteed to be at least the lowest of the following:

- The value of \text{MAX_INTEGER_SAMPLES}, if \text{internalformat} is a signed or unsigned integer format.
- The value of \text{MAX_DEPTH_TEXTURE_SAMPLES}, if \text{internalformat} is a depth/stencil-renderable format and \text{target} is \text{TEXTURE_2D_MULTISAMPLE} or \text{TEXTURE_2D_MULTISAMPLE_ARRAY}.
- The value of \text{MAX_COLOR_TEXTURE_SAMPLES}, if \text{internalformat} is a color-renderable format and \text{target} is \text{TEXTURE_2D_MULTISAMPLE} or \text{TEXTURE_2D_MULTISAMPLE_ARRAY}.
- The value of \text{MAX_SAMPLES}.

An \text{INVALID_ENUM} error is generated if \text{internalformat} is not color-, depth- or stencil-renderable; if \text{target} is not one of \text{TEXTURE_2D_MULTISAMPLE}, \text{TEXTURE_2D_MULTISAMPLE_ARRAY} or \text{RENDERBUFFER}; or if \text{pname} is not \text{SAMPLES} or \text{NUM_SAMPLE_COUNTS}.

An \text{INVALID_VALUE} error is generated if \text{bufSize} is negative.

6.2 State Tables

The tables on the following pages indicate which state variables are obtained with what commands. State variables that can be obtained using any of \text{GetBooleanv}, \text{GetIntegerv}, \text{GetInteger64v}, \text{GetFloatv}, or \text{GetDoublev} are listed with just one of these commands – the one that is most appropriate given the type of the data to be returned. These state variables cannot be obtained using \text{IsEnabled}. However, state variables for which \text{IsEnabled} is listed as the query command can also be obtained using \text{GetBooleanv}, \text{GetIntegerv}, \text{GetInteger64v}, \text{GetFloatv}, and \text{GetDoublev}. State variables for which any other command is listed as the query command can be obtained by using that command or any of its typed variants, although information may be lost when not using the listed command. Unless otherwise specified, when floating-point state is returned as integer values or integer
6.2. STATE TABLES

<table>
<thead>
<tr>
<th>Type code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>Boolean</td>
</tr>
<tr>
<td>$BMU$</td>
<td>Basic machine units</td>
</tr>
<tr>
<td>$C$</td>
<td>Color (floating-point R, G, B, and A values)</td>
</tr>
<tr>
<td>$Z$</td>
<td>Integer</td>
</tr>
<tr>
<td>$Z^+$</td>
<td>Non-negative integer or enumerated token value</td>
</tr>
<tr>
<td>$Z_{k, k^*}$</td>
<td>$k$-valued integer ($k^*$ indicates $k$ is minimum)</td>
</tr>
<tr>
<td>$R$</td>
<td>Floating-point number</td>
</tr>
<tr>
<td>$R^+$</td>
<td>Non-negative floating-point number</td>
</tr>
<tr>
<td>$R^{[a,b]}$</td>
<td>Floating-point number in the range $[a,b]$</td>
</tr>
<tr>
<td>$R^k$</td>
<td>$k$-tuple of floating-point numbers</td>
</tr>
<tr>
<td>$S$</td>
<td>null-terminated string</td>
</tr>
<tr>
<td>$I$</td>
<td>Image</td>
</tr>
<tr>
<td>$Y$</td>
<td>Pointer (data type unspecified)</td>
</tr>
<tr>
<td>$n \times type$</td>
<td>$n$ copies of type $type$ ($n^*$ indicates $n$ is minimum)</td>
</tr>
</tbody>
</table>

Table 6.3: State Variable Types

State is returned as floating-point values it is converted in the fashion described in section 6.1.2.

State table entries indicate a type for each variable. Table 6.3 explains these types. The type actually identifies all state associated with the indicated description; in certain cases only a portion of this state is returned. This is the case with textures, where only the selected texture or texture parameter is returned.

The $M$ and $m$ entries for initial minmax table values represent the maximum and minimum possible representable values, respectively.
### Table 6.4. Current Values and Associated Data

<table>
<thead>
<tr>
<th>Command</th>
<th>Type</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetIntegerv</td>
<td>$\mathbb{Z}$+</td>
<td>3</td>
<td>Number of vertices in input patch</td>
<td>2.6.1</td>
</tr>
<tr>
<td>GetFloatv</td>
<td>$4 \times \mathbb{R}$</td>
<td>(1.0, 1.0, 1.0, 1.0)</td>
<td>Default outer tess. level w/o control shader</td>
<td>2.12.2</td>
</tr>
<tr>
<td>GetFloatv</td>
<td>$2 \times \mathbb{R}$</td>
<td>(1.0, 1.0)</td>
<td>Default inner tess. level w/o control shader</td>
<td>2.12.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Type</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PatchVertices</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PatchDefaultOuterLevel</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PatchDefaultInnerLevel</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PatchColor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## 6.2. STATE TABLES

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetVertexAttribiv</td>
<td>FALSE</td>
<td>4</td>
<td>Vertex attrib array enable</td>
<td>2.8</td>
</tr>
<tr>
<td>GetVertexAttribiv</td>
<td>0</td>
<td>Vertex attrib array stride</td>
<td>2.8</td>
<td></td>
</tr>
<tr>
<td>GetVertexAttribiv</td>
<td>FLOAT</td>
<td>Vertex attrib array type</td>
<td>2.8</td>
<td></td>
</tr>
<tr>
<td>GetVertexAttribiv</td>
<td>FALSE</td>
<td>Vertex attrib array normalized</td>
<td>2.8</td>
<td></td>
</tr>
<tr>
<td>GetVertexAttribiv</td>
<td>0</td>
<td>Vertex attrib array instance divisor</td>
<td>2.8.3</td>
<td></td>
</tr>
<tr>
<td>GetVertexAttribPointerv</td>
<td>NULL</td>
<td>Vertex attrib array instance pointer</td>
<td>2.8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Type</th>
<th>Get value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetVertexAttribiv</td>
<td>16 * X</td>
<td>VERTEX_ATTRIB_ARRAY_ENABLED</td>
<td>Vertex attrib array enabled</td>
<td>2.8</td>
</tr>
<tr>
<td>GetVertexAttribiv</td>
<td>16 * X</td>
<td>VERTEX_ATTRIB_ARRAY_SIZE</td>
<td>Vertex attrib array size</td>
<td>2.8</td>
</tr>
<tr>
<td>GetVertexAttribiv</td>
<td>16 * X</td>
<td>VERTEX_ATTRIB_ARRAY_STRIDE</td>
<td>Vertex attrib array stride</td>
<td>2.8</td>
</tr>
<tr>
<td>GetVertexAttribiv</td>
<td>16 * X</td>
<td>VERTEX_ATTRIB_ARRAY_TYPE</td>
<td>Vertex attrib array type</td>
<td>2.8</td>
</tr>
<tr>
<td>GetVertexAttribiv</td>
<td>16 * X</td>
<td>VERTEX_ATTRIB_ARRAY_NORMALIZED</td>
<td>Vertex attrib array normalized</td>
<td>2.8</td>
</tr>
<tr>
<td>GetVertexAttribiv</td>
<td>16 * X</td>
<td>VERTEX_ATTRIB_ARRAY_INTEGER</td>
<td>Vertex attrib array has unconverted integers</td>
<td>2.8</td>
</tr>
<tr>
<td>GetVertexAttribiv</td>
<td>16 * Y</td>
<td>VERTEX_ATTRIB_ARRAY_DIVISOR</td>
<td>Vertex attrib array instance divisor</td>
<td>2.8.3</td>
</tr>
<tr>
<td>GetVertexAttribPointerv</td>
<td>NULL</td>
<td>VERTEX_ATTRIB_ARRAY_POINTER</td>
<td>Vertex attrib array instance pointer</td>
<td>2.8</td>
</tr>
</tbody>
</table>

Table 6.5. Vertex Array Object State (cont.)
### Table 6.6. Vertex Array Object State (cont.)

<table>
<thead>
<tr>
<th>Command</th>
<th>Type</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetIntegerv</td>
<td>$Z^+$</td>
<td>0</td>
<td>Array element binding</td>
<td>2.9.7</td>
</tr>
<tr>
<td>GetVertextAttrib</td>
<td>$16 \times Z^+$</td>
<td>0</td>
<td>Array attribute binding</td>
<td>2.9</td>
</tr>
<tr>
<td>Get Command</td>
<td>Type</td>
<td>Initial Value</td>
<td>Description</td>
<td>Sec.</td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
<td>---------------</td>
<td>-------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>ARRAY BUFFER BINDING</td>
<td>0</td>
<td>Current buffer binding</td>
<td>2.9</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>DRAW INDIRECT BUFFER BINDING</td>
<td>0</td>
<td>Indirect command buffer binding</td>
<td>2.9.8</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>VERTEX ARRAY BINDING</td>
<td>0</td>
<td>Current vertex array object binding</td>
<td>2.10</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>PRIMITIVE RESTART B</td>
<td>0</td>
<td>FALSE - Primitive restart enable</td>
<td>2.8</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>PRIMITIVE RESTART INDEX</td>
<td>0</td>
<td>Primitive restart index</td>
<td>2.8</td>
</tr>
</tbody>
</table>

Table 6.7. Vertex Array Data (not in Vertex Array objects)
<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>$n \times BMU$</td>
<td>GetBufferSubData</td>
<td>-</td>
<td>Buffer data</td>
<td>2.9</td>
</tr>
<tr>
<td>BUFFER_SIZE</td>
<td>$n \times Z^+$</td>
<td>GetBufferParameteri64v</td>
<td>0</td>
<td>Buffer data size</td>
<td>2.9</td>
</tr>
<tr>
<td>BUFFER_USAGE</td>
<td>$n \times Z_0$</td>
<td>GetBufferParameteriv</td>
<td>STATIC_DRAW</td>
<td>Buffer usage pattern</td>
<td>2.9</td>
</tr>
<tr>
<td>BUFFER_ACCESS</td>
<td>$n \times Z_3$</td>
<td>GetBufferParameteriv</td>
<td>READ_WRITE</td>
<td>Buffer access flag</td>
<td>2.9.3</td>
</tr>
<tr>
<td>BUFFER_ACCESS_FLAGS</td>
<td>$n \times Z^+$</td>
<td>GetBufferParameteriv</td>
<td>0</td>
<td>Extended buffer access flag</td>
<td>2.9.3</td>
</tr>
<tr>
<td>BUFFER_MAPPED</td>
<td>$n \times B$</td>
<td>GetBufferParameteriv</td>
<td>FALSE</td>
<td>Buffer map flag</td>
<td>2.9.3</td>
</tr>
<tr>
<td>BUFFER_MAP_POINTER</td>
<td>$n \times Y^*$</td>
<td>GetBufferPointeriv</td>
<td>NULL</td>
<td>Mapped buffer pointer</td>
<td>2.9.3</td>
</tr>
<tr>
<td>BUFFER_MAP_OFFSET</td>
<td>$n \times Z^+$</td>
<td>GetBufferParameteri64v</td>
<td>0</td>
<td>Start of mapped buffer range</td>
<td>2.9.3</td>
</tr>
<tr>
<td>BUFFER_MAP_LENGTH</td>
<td>$n \times Z^+$</td>
<td>GetBufferParameteri64v</td>
<td>0</td>
<td>Size of mapped buffer range</td>
<td>2.9.3</td>
</tr>
</tbody>
</table>
## 6.2. STATE TABLES

<table>
<thead>
<tr>
<th>Command</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIEWPORT</td>
<td>GetFloatv</td>
<td>0.1</td>
<td>Viewport origin &amp; extent</td>
<td>2.14.1</td>
</tr>
<tr>
<td>DEPTH RANGE</td>
<td>GetDoublev</td>
<td>0</td>
<td>Depth range near &amp; far</td>
<td>2.14.1</td>
</tr>
<tr>
<td>CLIP DISTANCE</td>
<td>GetEnabled</td>
<td>FALSE</td>
<td>6th user clipping plane enabled</td>
<td>2.20</td>
</tr>
<tr>
<td>DEPTH CLAMP</td>
<td>GetEnabled</td>
<td>FALSE</td>
<td>Depth-clamping enabled</td>
<td>2.20</td>
</tr>
<tr>
<td>TRANSFORM FEEDBACK BINDING</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Object bound for transform feedback operations</td>
<td>2.17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Get Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIEWPORT</td>
<td>16 * 4 \times R</td>
</tr>
<tr>
<td>DEPTH RANGE</td>
<td>6 * B</td>
</tr>
<tr>
<td>CLIP DISTANCE</td>
<td>B</td>
</tr>
<tr>
<td>DEPTH CLAMP</td>
<td>Z+</td>
</tr>
</tbody>
</table>

Table 6.9. Transformation state
<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLAMP.READ.COLOR</td>
<td>$Z_3$</td>
<td>GetIntegerv</td>
<td>FIXED_.-_ONLY</td>
<td>Read color clamping</td>
<td>4.3.1</td>
</tr>
<tr>
<td>PROVOKING.VERTEX</td>
<td>$Z_2$</td>
<td>GetIntegerv</td>
<td>LAST_.-<em>VERTEX</em>.-_CONVENTION</td>
<td>Provoking vertex convention</td>
<td>2.19</td>
</tr>
</tbody>
</table>
### 6.2. STATE TABLES

**GL 4.2 (Core Profile) - April 27, 2012**

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RASTERIZER,Discard</td>
<td>B</td>
<td>isEnabled</td>
<td>FALSE</td>
<td>Discard primitives before rasterization</td>
<td>3.1</td>
</tr>
<tr>
<td>POINT SIZE</td>
<td>R⁺</td>
<td>GetFloatv</td>
<td>1.0</td>
<td>Point size</td>
<td>3.4</td>
</tr>
<tr>
<td>POINT,FADE,THRESHOLD,SIZE</td>
<td>R⁺</td>
<td>GetFloatv</td>
<td>1.0</td>
<td>Threshold for alpha attenuation</td>
<td>3.4</td>
</tr>
<tr>
<td>POINT,SPRITE,COORD,ORIGIN</td>
<td>Z²</td>
<td>GetIntegerv</td>
<td>UPPER_LEFT</td>
<td>Origin orientation for point sprites</td>
<td>3.4</td>
</tr>
<tr>
<td>LINE,WIDTH</td>
<td>R⁺</td>
<td>GetFloatv</td>
<td>1.0</td>
<td>Line width</td>
<td>3.5</td>
</tr>
<tr>
<td>LINE,SMOOTH</td>
<td>B</td>
<td>isEnabled</td>
<td>FALSE</td>
<td>Line antialiasing on</td>
<td>3.5</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Initial Value</td>
<td>Description</td>
<td>Sec.</td>
</tr>
<tr>
<td>-----------------</td>
<td>------</td>
<td>-------------</td>
<td>---------------</td>
<td>-------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>CULL_FACE</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Polygon culling enabled</td>
<td>3.6.1</td>
</tr>
<tr>
<td>CULL_FACE_MODE</td>
<td>Z_3</td>
<td>GetIntegerv</td>
<td>BACK</td>
<td>Cull front-/back-facing polygons</td>
<td>3.6.1</td>
</tr>
<tr>
<td>FRONT_FACE</td>
<td>Z_2</td>
<td>GetIntegerv</td>
<td>CCW</td>
<td>Polygon frontface CW/CCW indicator</td>
<td>3.6.1</td>
</tr>
<tr>
<td>POLYGON_SMOOTH</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Polygon antialiasing on</td>
<td>3.6</td>
</tr>
<tr>
<td>POLYGON_MODE</td>
<td>Z_3</td>
<td>GetIntegerv</td>
<td>FILL</td>
<td>Polygon rasterization mode (front &amp; back)</td>
<td>3.6.3</td>
</tr>
<tr>
<td>POLYGON_OFFSET_FACTOR</td>
<td>R</td>
<td>GetFloatv</td>
<td>0</td>
<td>Polygon offset factor</td>
<td>3.6.4</td>
</tr>
<tr>
<td>POLYGON_OFFSET_UNITS</td>
<td>R</td>
<td>GetFloatv</td>
<td>0</td>
<td>Polygon offset units</td>
<td>3.6.4</td>
</tr>
<tr>
<td>POLYGON_OFFSET_POINT</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Polygon offset enable for POINT mode rasterization</td>
<td>3.6.4</td>
</tr>
<tr>
<td>POLYGON_OFFSET_LINE</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Polygon offset enable for LINE mode rasterization</td>
<td>3.6.4</td>
</tr>
<tr>
<td>POLYGON_OFFSET_FILL</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Polygon offset enable for FILL mode rasterization</td>
<td>3.6.4</td>
</tr>
</tbody>
</table>
### Table 6.13. Multisampling

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTISAMPLE</td>
<td>TRUE</td>
<td>Multisample rasterization</td>
<td>3.3.1</td>
</tr>
<tr>
<td>SAMPLELPIPHA</td>
<td>FALSE</td>
<td>Modify coverage from alpha</td>
<td>4.1.3</td>
</tr>
<tr>
<td>SAMPLELPIPHA</td>
<td>FALSE</td>
<td>Set alpha to maximum</td>
<td>4.1.3</td>
</tr>
<tr>
<td>SAMPLELPIPHAVALE</td>
<td>1</td>
<td>Coverage mask value</td>
<td>4.1.3</td>
</tr>
<tr>
<td>SAMPLELPIPHAINVERT</td>
<td>FALSE</td>
<td>Invert coverage mask value</td>
<td>4.1.3</td>
</tr>
<tr>
<td>SAMPLELPIPHASHADING</td>
<td>FALSE</td>
<td>Sample shading enable</td>
<td>4.1.3</td>
</tr>
<tr>
<td>SAMPLELPIPHASHADINGVALUE</td>
<td>0</td>
<td>Sample shading mask enable</td>
<td>4.1.3</td>
</tr>
<tr>
<td>SAMPLELPIPHASK</td>
<td>FALSE</td>
<td>Sample mask enable</td>
<td>4.1.3</td>
</tr>
<tr>
<td>SAMPLELPIPHASKWORD</td>
<td>all bits of all words set</td>
<td>Sample mask words</td>
<td>4.1.3</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Initial Value</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-----------------------</td>
<td>-------------</td>
<td>---------------</td>
</tr>
<tr>
<td>TEXTURE_BINDING_1D</td>
<td>80 * 3 * Z+</td>
<td>GetIntegerv</td>
<td>0</td>
</tr>
<tr>
<td>TEXTURE_BINDING_1D_ARRAY</td>
<td>80 * Z+</td>
<td>GetIntegerv</td>
<td>0</td>
</tr>
<tr>
<td>TEXTURE_BINDING_2D_ARRAY</td>
<td>80 * Z+</td>
<td>GetIntegerv</td>
<td>0</td>
</tr>
<tr>
<td>TEXTURE_BINDING_CUBE_MAP_ARRAY</td>
<td>80 * Z+</td>
<td>GetIntegerv</td>
<td>0</td>
</tr>
<tr>
<td>TEXTURE_BINDING_RECTANGLE</td>
<td>80 * Z+</td>
<td>GetIntegerv</td>
<td>0</td>
</tr>
<tr>
<td>TEXTURE_BINDING_BUFFER</td>
<td>80 * Z+</td>
<td>GetIntegerv</td>
<td>0</td>
</tr>
<tr>
<td>TEXTURE_BINDING_CUBE_MAP</td>
<td>80 * Z+</td>
<td>GetIntegerv</td>
<td>0</td>
</tr>
<tr>
<td>TEXTURE_BINDING_2D_MULTISAMPLE</td>
<td>80 * Z+</td>
<td>GetIntegerv</td>
<td>0</td>
</tr>
<tr>
<td>TEXTURE_BINDING_2D_MULTISAMPLE_ARRAY</td>
<td>80 * Z+</td>
<td>GetIntegerv</td>
<td>0</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Initial Value</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------</td>
<td>-----------------</td>
<td>---------------</td>
</tr>
<tr>
<td>SAMPLER_BINDING</td>
<td>$80 \times Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
</tr>
<tr>
<td>TEXTURE_1D</td>
<td>$0 \times 3 \times I$</td>
<td>GetTexImage</td>
<td>see 3.9</td>
</tr>
<tr>
<td>TEXTURE_1D_ARRAY</td>
<td>$0 \times I$</td>
<td>GetTexImage</td>
<td>see 3.9</td>
</tr>
<tr>
<td>TEXTURE_2D_ARRAY</td>
<td>$0 \times I$</td>
<td>GetTexImage</td>
<td>see 3.9</td>
</tr>
<tr>
<td>TEXTURE_CUBE_MAP_ARRAY</td>
<td>$0 \times I$</td>
<td>GetTexImage</td>
<td>see 3.9</td>
</tr>
<tr>
<td>TEXTURE_RECTANGLE</td>
<td>$0 \times I$</td>
<td>GetTexImage</td>
<td>see 3.9</td>
</tr>
<tr>
<td>TEXTURE_CUBE_MAP_POSITIVE_X</td>
<td>$0 \times I$</td>
<td>GetTexImage</td>
<td>see 3.9.3</td>
</tr>
<tr>
<td>TEXTURE_CUBE_MAP_NEGATIVE_X</td>
<td>$0 \times I$</td>
<td>GetTexImage</td>
<td>see 3.9.3</td>
</tr>
<tr>
<td>TEXTURE_CUBE_MAP_POSITIVE_Y</td>
<td>$0 \times I$</td>
<td>GetTexImage</td>
<td>see 3.9.3</td>
</tr>
<tr>
<td>TEXTURE_CUBE_MAP_NEGATIVE_Y</td>
<td>$0 \times I$</td>
<td>GetTexImage</td>
<td>see 3.9.3</td>
</tr>
<tr>
<td>TEXTURE_CUBE_MAP_POSITIVE_Z</td>
<td>$0 \times I$</td>
<td>GetTexImage</td>
<td>see 3.9.3</td>
</tr>
<tr>
<td>TEXTURE_CUBE_MAP_NEGATIVE_Z</td>
<td>$0 \times I$</td>
<td>GetTexImage</td>
<td>see 3.9.3</td>
</tr>
</tbody>
</table>
### Table 6.16. Textures (state per texture object)

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXTURE_SWIZZLE_R</td>
<td>Z_N</td>
<td>GetTexParameter</td>
<td>RED</td>
<td>Red component swizzle</td>
<td>3.9.8</td>
</tr>
<tr>
<td>TEXTURE_SWIZZLE_G</td>
<td>Z_N</td>
<td>GetTexParameter</td>
<td>GREEN</td>
<td>Green component swizzle</td>
<td>3.9.8</td>
</tr>
<tr>
<td>TEXTURE_SWIZZLE_B</td>
<td>Z_N</td>
<td>GetTexParameter</td>
<td>BLUE</td>
<td>Blue component swizzle</td>
<td>3.9.8</td>
</tr>
<tr>
<td>TEXTURE_SWIZZLE_A</td>
<td>Z_N</td>
<td>GetTexParameter</td>
<td>ALPHA</td>
<td>Alpha component swizzle</td>
<td>3.9.8</td>
</tr>
<tr>
<td>TEXTURE_BORDER_COLOR</td>
<td>C_N</td>
<td>GetTexParameter</td>
<td>0.0,0.0,0.0,0.0</td>
<td>Border color</td>
<td>3.9</td>
</tr>
<tr>
<td>TEXTURE_MIN_FILTER</td>
<td>Z_N</td>
<td>GetTexParameter</td>
<td>see sec. 3.9.15</td>
<td>Minification function</td>
<td>3.9.11</td>
</tr>
<tr>
<td>TEXTURE_MAG_FILTER</td>
<td>Z_N</td>
<td>GetTexParameter</td>
<td>LINEAR</td>
<td>Magnification function</td>
<td>3.9.12</td>
</tr>
<tr>
<td>TEXTURE_WRAP_S</td>
<td>Z_N</td>
<td>GetTexParameter</td>
<td>see sec. 3.9.15</td>
<td>Texcoord s wrap mode</td>
<td>3.9.11</td>
</tr>
<tr>
<td>TEXTURE_WRAP_T</td>
<td>Z_N</td>
<td>GetTexParameter</td>
<td>see sec. 3.9.15</td>
<td>Texcoord t wrap mode (2D, 3D, cube map textures only)</td>
<td>3.9.11</td>
</tr>
<tr>
<td>TEXTURE_WRAP_R</td>
<td>Z_N</td>
<td>GetTexParameter</td>
<td>see sec. 3.9.15</td>
<td>Texcoord r wrap mode (3D textures only)</td>
<td>3.9.11</td>
</tr>
<tr>
<td>TEXTURE_MIN_LOD</td>
<td>R_N</td>
<td>GetTexParameterfv</td>
<td>-1000</td>
<td>Minimum level of detail</td>
<td>3.9</td>
</tr>
<tr>
<td>TEXTURE_MAX_LOD</td>
<td>R_N</td>
<td>GetTexParameterfv</td>
<td>1000</td>
<td>Maximum level of detail</td>
<td>3.9</td>
</tr>
<tr>
<td>TEXTURE_BASE_LEVEL</td>
<td>Z_N</td>
<td>GetTexParameterfv</td>
<td>0</td>
<td>Base texture array</td>
<td>3.9</td>
</tr>
<tr>
<td>TEXTURE_MAX_LEVEL</td>
<td>Z_N</td>
<td>GetTexParameterfv</td>
<td>1000</td>
<td>Max. texture array level</td>
<td>3.9</td>
</tr>
<tr>
<td>TEXTURE_LOD_BIAS</td>
<td>R_N</td>
<td>GetTexParameterfv</td>
<td>0.0</td>
<td>Texture level of detail bias (bias\text_{texobj})</td>
<td>3.9.11</td>
</tr>
<tr>
<td>TEXTURE_COMPARE_MODE</td>
<td>Z_N</td>
<td>GetTexParameteriv</td>
<td>NONE</td>
<td>Comparison mode</td>
<td>3.9.17</td>
</tr>
<tr>
<td>TEXTURE_COMPARE_FUNC</td>
<td>Z_N</td>
<td>GetTexParameteriv</td>
<td>LEQUAL</td>
<td>Comparison function</td>
<td>3.9.17</td>
</tr>
<tr>
<td>TEXTURE_IMMUTABLE_FORMAT</td>
<td>B</td>
<td>GetTexParameter</td>
<td>FALSE</td>
<td>Size and format immutable</td>
<td>3.9.16</td>
</tr>
<tr>
<td>IMAGE_FORMAT_COMPATIBILITY_TYPE</td>
<td>Z_N</td>
<td>GetTexParameteriv</td>
<td>see 3.9.20</td>
<td>Compatibility rules for texture use with image units</td>
<td>3.9.20</td>
</tr>
</tbody>
</table>
### Table 6.17. Textures (state per texture image)

<table>
<thead>
<tr>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXTURE WIDTH</td>
<td>GetTexImage</td>
<td>0</td>
<td>Specified width (2D/3D)</td>
</tr>
<tr>
<td>TEXTURE HEIGHT</td>
<td>GetTexImage</td>
<td>0</td>
<td>Specified height (2D/3D)</td>
</tr>
<tr>
<td>TEXTURE DEPTH</td>
<td>GetTexImage</td>
<td>0</td>
<td>Specified depth (3D)</td>
</tr>
<tr>
<td>TEXTURES SAMES</td>
<td>GetTexImage</td>
<td>0</td>
<td>Number of samples per texel</td>
</tr>
<tr>
<td>TEXTURE FIXTURE SAMPLE LOCATIONS</td>
<td>GetTexImage</td>
<td>TRUE</td>
<td>Whether the image uses a fixed sample pattern</td>
</tr>
<tr>
<td>TEXTURE INTERNAL FORMAT</td>
<td>GetTexImage</td>
<td>RGBA or R8</td>
<td>Internal format (see section 3.9.15)</td>
</tr>
<tr>
<td>TEXTURE SIZE</td>
<td>GetTexImage</td>
<td>0</td>
<td>Component resolution (x is RED, GREEN, BLUE, DEPTH, or ALPHA, or DEPTH)</td>
</tr>
<tr>
<td>TEXTURE TYPE</td>
<td>GetTexImage</td>
<td>NONE</td>
<td>Component type (x is RED, GREEN, BLUE, DEPTH, or ALPHA, or DEPTH)</td>
</tr>
<tr>
<td>TEXTURE COMRESSED</td>
<td>GetTexImage</td>
<td>FALSE</td>
<td>True if image has a compressed internal format</td>
</tr>
<tr>
<td>TEXTURE BUFFER DATA STORE BINDING</td>
<td>GetTexImage</td>
<td>0</td>
<td>Size (in ubyte) of compressed image</td>
</tr>
</tbody>
</table>

GetTexImage is a command used to get the state of a texture image. The table lists various parameters and their initial values, along with descriptions of what each parameter represents.
### TABLE 6.18. Textures (state per sampler object)

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXTURE BORDER COLOR</td>
<td>C</td>
<td>GetSamplerParameter</td>
<td>0.0,0.0,0.0,0.0</td>
<td>Border color</td>
<td>3.9</td>
</tr>
<tr>
<td>TEXTURE MIN FILTER</td>
<td>Z₀</td>
<td>GetSamplerParameter</td>
<td>see sec. 3.9.15</td>
<td>Minification function</td>
<td>3.9.11</td>
</tr>
<tr>
<td>TEXTURE MAG FILTER</td>
<td>Z₁</td>
<td>GetSamplerParameter</td>
<td>LINEAR</td>
<td>Magnification function</td>
<td>3.9.12</td>
</tr>
<tr>
<td>TEXTURE WRAP S</td>
<td>Z₂</td>
<td>GetSamplerParameter</td>
<td>see sec. 3.9.15</td>
<td>Texcoord s wrap mode</td>
<td>3.9.11</td>
</tr>
<tr>
<td>TEXTURE WRAP T</td>
<td>Z₃</td>
<td>GetSamplerParameter</td>
<td>see sec. 3.9.15</td>
<td>Texcoord t wrap mode (2D, 3D, cube map textures only)</td>
<td>3.9.11</td>
</tr>
<tr>
<td>TEXTURE WRAP R</td>
<td>Z₄</td>
<td>GetSamplerParameter</td>
<td>see sec. 3.9.15</td>
<td>Texcoord r wrap mode (3D textures only)</td>
<td>3.9.11</td>
</tr>
<tr>
<td>TEXTUREpearance LOD</td>
<td>R</td>
<td>GetSamplerParameterfv</td>
<td>-1000</td>
<td>Minimum level of detail</td>
<td>3.9</td>
</tr>
<tr>
<td>TEXTUREpearance LOD</td>
<td>R</td>
<td>GetSamplerParameterfv</td>
<td>1000</td>
<td>Maximum level of detail</td>
<td>3.9</td>
</tr>
<tr>
<td>TEXTUREpearance LOD BIAS</td>
<td>R</td>
<td>GetSamplerParameterfv</td>
<td>0.0</td>
<td>Texture level of detail bias (biasₜₑₓₒ𝙱𝙹)</td>
<td>3.9.11</td>
</tr>
<tr>
<td>TEXTURE_COMPARE_MODE</td>
<td>Z₂</td>
<td>GetSamplerParameteriv</td>
<td>NONE</td>
<td>Comparison mode</td>
<td>3.9.17</td>
</tr>
<tr>
<td>TEXTURE_COMPARE_FUNC</td>
<td>Z₃</td>
<td>GetSamplerParameteriv</td>
<td>LEQUAL</td>
<td>Comparison function</td>
<td>3.9.17</td>
</tr>
<tr>
<td>Get Value</td>
<td>Type</td>
<td>Initial Value</td>
<td>Get Command</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>---------------</td>
<td>-------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>TEXTURE</td>
<td>Z000</td>
<td>ACTIVE</td>
<td>GetIntegerv TEXTURE0</td>
<td>Active texture unit selector</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.19. Texture Environment and Generation
<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCISSOR_TEST</td>
<td>$16 \times B$</td>
<td>isEnabled</td>
<td>FALSE</td>
<td>Scissoring enabled</td>
<td>4.1.2</td>
</tr>
<tr>
<td>SCISSOR_BOX</td>
<td>$16 \times 4 \times Z$</td>
<td>GetInteger_v</td>
<td>see 4.1.2</td>
<td>Scissor box</td>
<td>4.1.2</td>
</tr>
<tr>
<td>STENCIL_TEST</td>
<td>$B$</td>
<td>isEnabled</td>
<td>FALSE</td>
<td>Stenciling enabled</td>
<td>4.1.4</td>
</tr>
<tr>
<td>STENCIL_FUNC</td>
<td>$Z_8$</td>
<td>GetInteger_v</td>
<td>ALWAYS</td>
<td>Front stencil function</td>
<td>4.1.4</td>
</tr>
<tr>
<td>STENCIL_VALUE_MASK</td>
<td>$Z^+$</td>
<td>GetInteger_v</td>
<td>see 4.1.4</td>
<td>Front stencil mask</td>
<td>4.1.4</td>
</tr>
<tr>
<td>STENCIL_REF</td>
<td>$Z^+$</td>
<td>GetInteger_v</td>
<td>0</td>
<td>Front stencil reference value</td>
<td>4.1.4</td>
</tr>
<tr>
<td>STENCIL_FAIL</td>
<td>$Z_8$</td>
<td>GetInteger_v</td>
<td>KEEP</td>
<td>Front stencil fail action</td>
<td>4.1.4</td>
</tr>
<tr>
<td>STENCIL_PASS_DEPTH_FAIL</td>
<td>$Z_8$</td>
<td>GetInteger_v</td>
<td>KEEP</td>
<td>Front stencil depth buffer fail action</td>
<td>4.1.4</td>
</tr>
<tr>
<td>STENCIL_PASS_DEPTH_PASS</td>
<td>$Z_8$</td>
<td>GetInteger_v</td>
<td>KEEP</td>
<td>Front stencil depth buffer pass action</td>
<td>4.1.4</td>
</tr>
<tr>
<td>STENCIL_BACK_FUNC</td>
<td>$Z_8$</td>
<td>GetInteger_v</td>
<td>ALWAYS</td>
<td>Back stencil function</td>
<td>4.1.4</td>
</tr>
<tr>
<td>STENCIL_BACK_VALUE_MASK</td>
<td>$Z^+$</td>
<td>GetInteger_v</td>
<td>see 4.1.4</td>
<td>Back stencil mask</td>
<td>4.1.4</td>
</tr>
<tr>
<td>STENCIL_BACK_REF</td>
<td>$Z^+$</td>
<td>GetInteger_v</td>
<td>0</td>
<td>Back stencil reference value</td>
<td>4.1.4</td>
</tr>
<tr>
<td>STENCIL_BACK_FAIL</td>
<td>$Z_8$</td>
<td>GetInteger_v</td>
<td>KEEP</td>
<td>Back stencil fail action</td>
<td>4.1.4</td>
</tr>
<tr>
<td>STENCIL_BACK_PASS_DEPTH_FAIL</td>
<td>$Z_8$</td>
<td>GetInteger_v</td>
<td>KEEP</td>
<td>Back stencil depth buffer fail action</td>
<td>4.1.4</td>
</tr>
<tr>
<td>STENCIL_BACK_PASS_DEPTH_PASS</td>
<td>$Z_8$</td>
<td>GetInteger_v</td>
<td>KEEP</td>
<td>Back stencil depth buffer pass action</td>
<td>4.1.4</td>
</tr>
<tr>
<td>DEPTH_TEST</td>
<td>$B$</td>
<td>isEnabled</td>
<td>FALSE</td>
<td>Depth buffer enabled</td>
<td>4.1.5</td>
</tr>
<tr>
<td>DEPTH_FUNC</td>
<td>$Z_8$</td>
<td>GetInteger_v</td>
<td>LESS</td>
<td>Depth buffer test function</td>
<td>4.1.5</td>
</tr>
</tbody>
</table>
### Table 6.21. Pixel Operations (cont.)

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLEND</td>
<td>8 × B</td>
<td>IsEnabledi</td>
<td>FALSE</td>
<td>Blending enabled for draw buffer i</td>
<td>4.1.7</td>
</tr>
<tr>
<td>BLEND_SRC_RGB</td>
<td>8 × Z₁₉</td>
<td>GetIntegeri_v</td>
<td>ONE</td>
<td>Blending source RGB function for draw buffer i</td>
<td>4.1.7</td>
</tr>
<tr>
<td>BLEND_SRC_ALPHA</td>
<td>8 × Z₁₉</td>
<td>GetIntegeri_v</td>
<td>ONE</td>
<td>Blending source A function for draw buffer i</td>
<td>4.1.7</td>
</tr>
<tr>
<td>BLEND_DST_RGB</td>
<td>8 × Z₁₉</td>
<td>GetIntegeri_v</td>
<td>ZERO</td>
<td>Blending dest. RGB function for draw buffer i</td>
<td>4.1.7</td>
</tr>
<tr>
<td>BLEND_DST_ALPHA</td>
<td>8 × Z₁₉</td>
<td>GetIntegeri_v</td>
<td>ZERO</td>
<td>Blending dest. A function for draw buffer i</td>
<td>4.1.7</td>
</tr>
<tr>
<td>BLEND_EQUATION_RGB</td>
<td>8 × Z₅</td>
<td>GetIntegeri_v</td>
<td>FUNC_ADD</td>
<td>RGB blending equation for draw buffer i</td>
<td>4.1.7</td>
</tr>
<tr>
<td>BLEND_EQUATION_ALPHA</td>
<td>8 × Z₅</td>
<td>GetIntegeri_v</td>
<td>FUNC_ADD</td>
<td>Alpha blending equation for draw buffer i</td>
<td>4.1.7</td>
</tr>
<tr>
<td>BLEND_COLOR</td>
<td>C</td>
<td>GetFloatv</td>
<td>0.0,0.0,0.0,0.0</td>
<td>Constant blend color</td>
<td>4.1.7</td>
</tr>
<tr>
<td>FRAMEBUFFER_SRGB</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>sRGB update and blending enable</td>
<td>4.1.7</td>
</tr>
<tr>
<td>DITHER</td>
<td>B</td>
<td>IsEnabled</td>
<td>TRUE</td>
<td>Dithering enabled</td>
<td>4.1.9</td>
</tr>
<tr>
<td>COLOR_LOGIC_OP</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Color logic op enabled</td>
<td>4.1.10</td>
</tr>
<tr>
<td>LOGIC_OP_MODE</td>
<td>Z₁₆</td>
<td>GetIntegerv</td>
<td>COPY</td>
<td>Logic op function</td>
<td>4.1.10</td>
</tr>
</tbody>
</table>
### Get value, Type, Get Command, Initial Value, Description, Sec.

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLOR_WRITEMASK</td>
<td>8 × 4 × B</td>
<td>GetBooleanv</td>
<td>(TRUE,TRUE,TRUE,TRUE)</td>
<td>Color write enables (R,G,B,A) for draw buffer i</td>
<td>4.2.2</td>
</tr>
<tr>
<td>DEPTH_WRITEMASK</td>
<td>B</td>
<td>GetBooleanv</td>
<td>TRUE</td>
<td>Depth buffer enabled for writing</td>
<td>4.2.2</td>
</tr>
<tr>
<td>STENCIL_WRITEMASK</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>1’s</td>
<td>Front stencil buffer writemask</td>
<td>4.2.2</td>
</tr>
<tr>
<td>STENCIL_BACK_WRITEMASK</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>1’s</td>
<td>Back stencil buffer writemask</td>
<td>4.2.2</td>
</tr>
<tr>
<td>COLOR_CLEAR_VALUE</td>
<td>C</td>
<td>GetFloatv</td>
<td>0.0,0.0,0.0,0.0</td>
<td>Color buffer clear value</td>
<td>4.2.3</td>
</tr>
<tr>
<td>DEPTH_CLEAR_VALUE</td>
<td>R+</td>
<td>GetFloatv</td>
<td>1</td>
<td>Depth buffer clear value</td>
<td>4.2.3</td>
</tr>
<tr>
<td>STENCIL_CLEAR_VALUE</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Stencil clear value</td>
<td>4.2.3</td>
</tr>
</tbody>
</table>

Table 6.22. Framebuffer Control

OpenGL 4.2 (Core Profile) - April 27, 2012
### Table 6.23. Framebuffer (state per target binding point)

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GetIntegerv</code></td>
<td><code>0</code></td>
<td>Framebuffer object bound to DRAW_FRAMEBUFFER</td>
<td>4.4.1</td>
</tr>
<tr>
<td><code>GetIntegerv</code></td>
<td><code>0</code></td>
<td>Framebuffer object bound to READ_FRAMEBUFFER</td>
<td>4.4.1</td>
</tr>
</tbody>
</table>
## Table 6.24. Framebuffer (state per framebuffer object)

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAW_BUFFERi</td>
<td>$8 \times Z_{11}$</td>
<td>GetIntegerv</td>
<td>see 4.2.1</td>
<td>Draw buffer selected for color output $i$</td>
<td>4.2.1</td>
</tr>
<tr>
<td>READ_BUFFER</td>
<td>$Z_{11}$</td>
<td>GetIntegerv</td>
<td>see 4.3.1</td>
<td>Read source buffer †</td>
<td>4.3.1</td>
</tr>
</tbody>
</table>

† This state is queried from the currently bound read framebuffer.
<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRAMEBUFFER_ATTACHMENTOBJECTTYPE</td>
<td>$Z_4$</td>
<td>GetFramebufferAttachmentParameteriv</td>
<td>NONE</td>
<td>Type of image attached to framebuffer attachment point</td>
<td>4.4.2</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENTOBJECTNAME</td>
<td>$Z^+$</td>
<td>GetFramebufferAttachmentParameteriv</td>
<td>0</td>
<td>Name of object attached to framebuffer attachment point</td>
<td>4.4.2</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENTTEXTURELEVEL</td>
<td>$Z^+$</td>
<td>GetFramebufferAttachmentParameteriv</td>
<td>0</td>
<td>Mipmap level of texture image attached, if object attached is texture</td>
<td>4.4.2</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENTTEXTURECUBEMAPFACE</td>
<td>$Z^+$</td>
<td>GetFramebufferAttachmentParameteriv</td>
<td>NONE</td>
<td>Cubemap face of texture image attached, if object attached is cubemap texture</td>
<td>4.4.2</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENTTEXTURELAYER</td>
<td>$Z$</td>
<td>GetFramebufferAttachmentParameteriv</td>
<td>0</td>
<td>Layer of texture image attached, if object attached is 3D texture</td>
<td>4.4.2</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENTLAYERED</td>
<td>$n \times B$</td>
<td>GetFramebufferAttachmentParameteriv</td>
<td>FALSE</td>
<td>Framebuffer attachment is layered</td>
<td>4.4.7</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENTCOLORENCODING</td>
<td>$Z_2$</td>
<td>GetFramebufferAttachmentParameteriv</td>
<td>-</td>
<td>Encoding of components in the attached image</td>
<td>6.1.13</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENTCOMPONENTTYPE</td>
<td>$Z_4$</td>
<td>GetFramebufferAttachmentParameteriv</td>
<td>-</td>
<td>Data type of components in the attached image</td>
<td>6.1.13</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENTxSIZE</td>
<td>$Z^+$</td>
<td>GetFramebufferAttachmentParameteriv</td>
<td>-</td>
<td>Size in bits of attached image’s $x$ component; $x$ is RED, GREEN, BLUE, ALPHA, DEPTH, or STENCIL</td>
<td>6.1.13</td>
</tr>
</tbody>
</table>

Table 6.25. Framebuffer (state per attachment point)
### Table 6.26. Renderbuffer (state per target and binding point)

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>GetInteger</td>
<td>0</td>
<td>Renderbuffer object bound to RENDERBUFFER</td>
<td>4.4.2</td>
</tr>
</tbody>
</table>

6.2. **STATE TABLES**
### Table 6.27. Renderbuffer (state per renderbuffer object)

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RENDERBUFFER_WIDTH</td>
<td>$Z^+$</td>
<td>GetRenderbufferParameteriv</td>
<td>0</td>
<td>Width of renderbuffer</td>
<td>4.4.2</td>
</tr>
<tr>
<td>RENDERBUFFER_HEIGHT</td>
<td>$Z^+$</td>
<td>GetRenderbufferParameteriv</td>
<td>0</td>
<td>Height of renderbuffer</td>
<td>4.4.2</td>
</tr>
<tr>
<td>RENDERBUFFER_INTERNAL_FORMAT</td>
<td>$Z^+$</td>
<td>GetRenderbufferParameteriv</td>
<td>RGBA</td>
<td>Internal format of renderbuffer</td>
<td>4.4.2</td>
</tr>
<tr>
<td>RENDERBUFFER_RED_SIZE</td>
<td>$Z^+$</td>
<td>GetRenderbufferParameteriv</td>
<td>0</td>
<td>Size in bits of renderbuffer image’s red component</td>
<td>4.4.2</td>
</tr>
<tr>
<td>RENDERBUFFER_GREEN_SIZE</td>
<td>$Z^+$</td>
<td>GetRenderbufferParameteriv</td>
<td>0</td>
<td>Size in bits of renderbuffer image’s green component</td>
<td>4.4.2</td>
</tr>
<tr>
<td>RENDERBUFFER_BLUE_SIZE</td>
<td>$Z^+$</td>
<td>GetRenderbufferParameteriv</td>
<td>0</td>
<td>Size in bits of renderbuffer image’s blue component</td>
<td>4.4.2</td>
</tr>
<tr>
<td>RENDERBUFFER_ALPHA_SIZE</td>
<td>$Z^+$</td>
<td>GetRenderbufferParameteriv</td>
<td>0</td>
<td>Size in bits of renderbuffer image’s alpha component</td>
<td>4.4.2</td>
</tr>
<tr>
<td>RENDERBUFFER_DEPTH_SIZE</td>
<td>$Z^+$</td>
<td>GetRenderbufferParameteriv</td>
<td>0</td>
<td>Size in bits of renderbuffer image’s depth component</td>
<td>4.4.2</td>
</tr>
<tr>
<td>RENDERBUFFER_STENCIL_SIZE</td>
<td>$Z^+$</td>
<td>GetRenderbufferParameteriv</td>
<td>0</td>
<td>Size in bits of renderbuffer image’s stencil component</td>
<td>4.4.2</td>
</tr>
<tr>
<td>RENDERBUFFER_SAMPLES</td>
<td>$Z^+$</td>
<td>GetRenderbufferParameteriv</td>
<td>0</td>
<td>Number of samples</td>
<td>4.4.2</td>
</tr>
</tbody>
</table>
### 6.2. STATE TABLES

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNPACK_SWAP_BYTES</td>
<td>B</td>
<td>GetBooleanv</td>
<td>FALSE</td>
<td>Value of UNPACK_SWAP_BYTES</td>
<td>3.7.1</td>
</tr>
<tr>
<td>UNPACK_LSB_FIRST</td>
<td>B</td>
<td>GetBooleanv</td>
<td>FALSE</td>
<td>Value of UNPACK_LSB_FIRST</td>
<td>3.7.1</td>
</tr>
<tr>
<td>UNPACK_IMAGE_HEIGHT</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Value of UNPACK_IMAGE_HEIGHT</td>
<td>3.7.1</td>
</tr>
<tr>
<td>UNPACK_SKIP_IMAGES</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Value of UNPACK_SKIP.Images</td>
<td>3.7.1</td>
</tr>
<tr>
<td>UNPACK_ROW_LENGTH</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Value of UNPACK_ROW_LENGTH</td>
<td>3.7.1</td>
</tr>
<tr>
<td>UNPACK_SKIP_ROWS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Value of UNPACK_SKIP_ROWS</td>
<td>3.7.1</td>
</tr>
<tr>
<td>UNPACK_SKIP_PIXELS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Value of UNPACK_SKIP_PIXELS</td>
<td>3.7.1</td>
</tr>
<tr>
<td>UNPACK_ALIGNMENT</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>4</td>
<td>Value of UNPACK_ALIGNMENT</td>
<td>3.7.1</td>
</tr>
<tr>
<td>UNPACK_COMPRESSED_BLOCK_WIDTH</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Value of UNPACK_COMPRESSED_BLOCK_WIDTH</td>
<td>3.7.1</td>
</tr>
<tr>
<td>UNPACK_COMPRESSED_BLOCK_HEIGHT</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Value of UNPACK_COMPRESSED_BLOCK_HEIGHT</td>
<td>3.7.1</td>
</tr>
<tr>
<td>UNPACK_COMPRESSED_BLOCK_DEPTH</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Value of UNPACK_COMPRESSED_BLOCK_DEPTH</td>
<td>3.7.1</td>
</tr>
<tr>
<td>UNPACK_COMPRESSED_BLOCK_SIZE</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Value of UNPACK_COMPRESSED_BLOCK_SIZE</td>
<td>3.7.1</td>
</tr>
<tr>
<td>PIXEL_UNPACK_BUFFER_BINDING</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Pixel unpack buffer binding</td>
<td>6.1.9</td>
</tr>
</tbody>
</table>

Table 6.28. Pixels

OpenGL 4.2 (Core Profile) - April 27, 2012
<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PACK_SWAP_BYTES</td>
<td>B</td>
<td>GetBooleanv</td>
<td>FALSE</td>
<td>Value of PACK_SWAP_BYTES</td>
<td>4.3.1</td>
</tr>
<tr>
<td>PACK_LSB_FIRST</td>
<td>B</td>
<td>GetBooleanv</td>
<td>FALSE</td>
<td>Value of PACK_LSB_FIRST</td>
<td>4.3.1</td>
</tr>
<tr>
<td>PACK_IMAGE_HEIGHT</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Value of PACK_IMAGE_HEIGHT</td>
<td>4.3.1</td>
</tr>
<tr>
<td>PACK_SKIP_IMAGES</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Value of PACK_SKIP_IMAGES</td>
<td>4.3.1</td>
</tr>
<tr>
<td>PACK_ROW_LENGTH</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Value of PACK_ROW_LENGTH</td>
<td>4.3.1</td>
</tr>
<tr>
<td>PACK_SKIP_ROWS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Value of PACK_SKIP_ROWS</td>
<td>4.3.1</td>
</tr>
<tr>
<td>PACK_SKIP_PIXELS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Value of PACK_SKIP_PIXELS</td>
<td>4.3.1</td>
</tr>
<tr>
<td>PACK_ALIGNMENT</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>4</td>
<td>Value of PACK_ALIGNMENT</td>
<td>4.3.1</td>
</tr>
<tr>
<td>PACK_COMPRESSED_BLOCK_WIDTH</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Value of PACK_COMPRESSED_BLOCK_WIDTH</td>
<td>4.3.1</td>
</tr>
<tr>
<td>PACK_COMPRESSED_BLOCK_HEIGHT</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Value of PACK_COMPRESSED_BLOCK_HEIGHT</td>
<td>4.3.1</td>
</tr>
<tr>
<td>PACK_COMPRESSED_BLOCK_DEPTH</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Value of PACK_COMPRESSED_BLOCK_DEPTH</td>
<td>4.3.1</td>
</tr>
<tr>
<td>PACK_COMPRESSED_BLOCK_SIZE</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Value of PACK_COMPRESSED_BLOCK_SIZE</td>
<td>4.3.1</td>
</tr>
<tr>
<td>PIXEL_PACK_BUFFER_BINDING</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Pixel pack buffer binding</td>
<td>4.3.1</td>
</tr>
<tr>
<td>Get value</td>
<td>Command</td>
<td>Initial Value</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
<td>--------------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.30. Shader Object State
### 6.2. STATE TABLES

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Get value</th>
<th>Type</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetProgramPipelineiv</td>
<td>0</td>
<td>ACTIVE_PROGRAM</td>
<td>0</td>
<td>Program object updated by Uniform* when PPO bound</td>
<td>2.11.4</td>
</tr>
<tr>
<td>GetProgramPipelineiv</td>
<td>0</td>
<td>VERTEX_SHADER</td>
<td>0</td>
<td>Name of current vertex shader program object</td>
<td>2.11.4</td>
</tr>
<tr>
<td>GetProgramPipelineiv</td>
<td>0</td>
<td>GEOMETRY_SHADER</td>
<td>0</td>
<td>Name of current geometry shader program object</td>
<td>2.11.4</td>
</tr>
<tr>
<td>GetProgramPipelineiv</td>
<td>0</td>
<td>FRAGMENT_SHADER</td>
<td>0</td>
<td>Name of current fragment shader program object</td>
<td>2.11.4</td>
</tr>
<tr>
<td>GetProgramPipelineiv</td>
<td>0</td>
<td>TESS_CONTROL_SHADER</td>
<td>0</td>
<td>Name of current TCS program object</td>
<td>2.11.4</td>
</tr>
<tr>
<td>GetProgramPipelineiv</td>
<td>0</td>
<td>TESS_EVALUATION_SHADER</td>
<td>0</td>
<td>Name of current TES program object</td>
<td>2.11.4</td>
</tr>
<tr>
<td>GetProgramPipelineiv</td>
<td>0</td>
<td>VALIDATE_STATUS</td>
<td>FALSE</td>
<td>Validate status of program pipeline object</td>
<td>2.11.4</td>
</tr>
<tr>
<td>GetProgramPipelineInfoLog</td>
<td>0</td>
<td>INFO_LOG_LENGTH</td>
<td>empty</td>
<td>Length of info log</td>
<td>6.11.2</td>
</tr>
</tbody>
</table>

Table 6.31. Program Pipeline Object State
## 6.2. State Tables

### Table 6.32. Program Object State

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetIntegerv</td>
<td>0</td>
<td>Name of current program object</td>
<td>2.11.3</td>
</tr>
<tr>
<td>GetProgramiv</td>
<td>0</td>
<td>Current program pipeline object</td>
<td>2.11.4</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>FALSE</td>
<td>Program object can be bound for separate pipeline stages</td>
<td>2.11.3</td>
</tr>
<tr>
<td>GetProgramiv</td>
<td>FALSE</td>
<td>Program object deleted</td>
<td>2.11.3</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>FALSE</td>
<td>Last link attempt succeeded</td>
<td>6.11.12</td>
</tr>
<tr>
<td>GetProgramiv</td>
<td>FALSE</td>
<td>Last validate attempt succeeded</td>
<td>6.11.3</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>FALSE</td>
<td>Number of attached shader objects</td>
<td>6.11.2</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>0</td>
<td>Shader objects attached</td>
<td>2.11.3</td>
</tr>
<tr>
<td>GetProgramiv</td>
<td>empty</td>
<td>Info log for program object</td>
<td>6.11.2</td>
</tr>
<tr>
<td>GetProgramiv</td>
<td>empty</td>
<td>Length of program object</td>
<td>2.11.3</td>
</tr>
<tr>
<td>GetProgramiv</td>
<td>FALSE</td>
<td>Retrievable binary hint enabled</td>
<td>2.11.5</td>
</tr>
<tr>
<td>GetProgramiv</td>
<td>0</td>
<td>Binary representation of program</td>
<td>2.11.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENTPROGRAM</td>
<td>( \mathbb{Z}^+ )</td>
<td>Name of current program</td>
</tr>
<tr>
<td>PROGRAMPIPELINEBINDING</td>
<td>( \mathbb{B} )</td>
<td>Current program pipeline object</td>
</tr>
<tr>
<td>PROGRAMSEPARABLE</td>
<td>( \mathbb{B} )</td>
<td>Program object can be bound for separate pipeline stages</td>
</tr>
<tr>
<td>PROGRAMOBJECT</td>
<td>( \mathbb{B} )</td>
<td>Program object deleted</td>
</tr>
<tr>
<td>PROGRAMLINKED</td>
<td>( \mathbb{B} )</td>
<td>Last link attempt succeeded</td>
</tr>
<tr>
<td>PROGRAMVALIDATE</td>
<td>( \mathbb{B} )</td>
<td>Last validate attempt succeeded</td>
</tr>
<tr>
<td>PROGRAMATTACHEDSHADERS</td>
<td>( \mathbb{Z}^+ )</td>
<td>Number of attached shader objects</td>
</tr>
<tr>
<td>PROGRAMINFOLOG</td>
<td>empty</td>
<td>Info log for program object</td>
</tr>
<tr>
<td>PROGRAMINFOLength</td>
<td>( \mathbb{B} )</td>
<td>Length of program object</td>
</tr>
<tr>
<td>PROGRAMRETRIEVABLE</td>
<td>FALSE</td>
<td>Retrievable binary hint enabled</td>
</tr>
<tr>
<td>PROGRAMBINARY</td>
<td>( 0 \times \mathbb{BMU} )</td>
<td>Binary representation of program</td>
</tr>
</tbody>
</table>

OpenGL 4.2 (Core Profile) - April 27, 2012
<table>
<thead>
<tr>
<th>Get Command</th>
<th>Initial Value</th>
<th>Get value</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetUniformiv</td>
<td>0</td>
<td>-</td>
<td>Z*</td>
<td>Number of active uniforms</td>
</tr>
<tr>
<td>GetUniformLocation</td>
<td>-</td>
<td>-</td>
<td>Z*</td>
<td>Location of active uniforms</td>
</tr>
<tr>
<td>GetActiveUniform</td>
<td>-</td>
<td>empty</td>
<td>Z*</td>
<td>Type of active uniform name length</td>
</tr>
<tr>
<td>GetActiveUniform</td>
<td>-</td>
<td>empty</td>
<td>Z*</td>
<td>Name of active uniform</td>
</tr>
<tr>
<td>GetProgramiv</td>
<td>0</td>
<td>0</td>
<td>Z*</td>
<td>Maximum active uniform name length</td>
</tr>
<tr>
<td>GetAttribLocation</td>
<td>-</td>
<td>-</td>
<td>Z*</td>
<td>Location of generic attribute</td>
</tr>
<tr>
<td>GetActiveAttrib</td>
<td>-</td>
<td>empty</td>
<td>Z*</td>
<td>Type of active attribute</td>
</tr>
<tr>
<td>GetActiveAttrib</td>
<td>-</td>
<td>empty</td>
<td>Z*</td>
<td>Name of active attribute</td>
</tr>
<tr>
<td>GetProgramiv</td>
<td>0</td>
<td>0</td>
<td>Z*</td>
<td>Maximum active attribute name length</td>
</tr>
</tbody>
</table>

Table 6.33. Program Object State (cont.)
<table>
<thead>
<tr>
<th>Type</th>
<th>Get Command</th>
<th>Description</th>
<th>Initial Value</th>
<th>Get value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEOMETRY</td>
<td>GetProgramiv</td>
<td>TRIANGLES</td>
<td>0</td>
<td>Z_5</td>
</tr>
<tr>
<td>INPUT TYPE</td>
<td>GetProgramiv</td>
<td>OUT Z</td>
<td>5</td>
<td>Z_3</td>
</tr>
<tr>
<td>OUTPUT TYPE</td>
<td>GetProgramiv</td>
<td>TRIANGLE_STRIP</td>
<td>1</td>
<td>Z_2</td>
</tr>
<tr>
<td>SHADER INVOCA-</td>
<td>GetProgramiv</td>
<td>INTERLEAVED_ATTRIBS</td>
<td>0</td>
<td>Z_1</td>
</tr>
<tr>
<td>TRANSFORM FEED-</td>
<td>GetProgramiv</td>
<td>INTERLEAVED_ATTRIBS</td>
<td>0</td>
<td>Z_1</td>
</tr>
<tr>
<td>BACK BUFFER -</td>
<td>GetProgramiv</td>
<td>TRAVERSAL_FEEDBACK_VARYING</td>
<td>0</td>
<td>Z_0 × char</td>
</tr>
<tr>
<td>TRANSFORM_FEED-</td>
<td>GetProgramiv</td>
<td>TRANSFORM_FEEDBACK_VARYING</td>
<td>0</td>
<td>Z_0 × char</td>
</tr>
<tr>
<td>BACK BUFFER -</td>
<td>GetProgramiv</td>
<td>NAME_LENGTH</td>
<td>No. of times a geom. shader should be executed for each input primitive</td>
<td>Z_0</td>
</tr>
<tr>
<td>NUMBER OF OUTPUT VERTICES</td>
<td>GetProgramiv</td>
<td>No. of outputs to stream to buffer object(s)</td>
<td>Z_0</td>
<td></td>
</tr>
<tr>
<td>MAX LENGTH</td>
<td>GetProgramiv</td>
<td>Max. transform feedback output variable name length</td>
<td>Z_0</td>
<td></td>
</tr>
<tr>
<td>NAME</td>
<td>GetTransformFeedbackVarying</td>
<td>Name of each transform feedback output variable</td>
<td>Z_0</td>
<td></td>
</tr>
<tr>
<td>SIZE</td>
<td>GetTransformFeedbackVarying</td>
<td>Size of each transform feedback output variable</td>
<td>Z_0</td>
<td></td>
</tr>
<tr>
<td>TYPE</td>
<td>GetTransformFeedbackVarying</td>
<td>Type of each transform feedback output variable</td>
<td>Z_0</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.34. Program Object State (cont.)
### 6.2. STATE TABLES

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Type</th>
<th>Description</th>
<th>Initial Value</th>
<th>Get</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetIntegerv</td>
<td>$Z^+$</td>
<td>Uniform buffer object bound to the context for buffer object manipulation</td>
<td>0</td>
<td>GetInteger</td>
<td>2.11.7</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>$n \times Z^+$</td>
<td>Uniform buffer object bound to the specified context binding point</td>
<td>0</td>
<td>GetInteger</td>
<td>6.1.9</td>
</tr>
<tr>
<td>GetInteger64v</td>
<td>$n \times Z^+$</td>
<td>Start of bound uniform buffer region</td>
<td>0</td>
<td>GetInteger64</td>
<td>2.11.7</td>
</tr>
<tr>
<td>GetInteger64v</td>
<td>$n \times Z^+$</td>
<td>Size of bound uniform buffer region</td>
<td>0</td>
<td>GetInteger64</td>
<td>6.1.9</td>
</tr>
<tr>
<td>GetProgramiv</td>
<td>$Z^+$</td>
<td>Length of longest active uniform block name</td>
<td>0</td>
<td>GetProgram</td>
<td>2.11.7</td>
</tr>
<tr>
<td>GetActiveUniformiv</td>
<td>$0 \times Z^+$</td>
<td>Type of active uniform</td>
<td>-</td>
<td>GetActiveUniform</td>
<td>2.11.7</td>
</tr>
<tr>
<td>GetActiveUniformiv</td>
<td>$0 \times Z^+$</td>
<td>Size of active uniform</td>
<td>-</td>
<td>GetActiveUniform</td>
<td>2.11.7</td>
</tr>
<tr>
<td>GetActiveUniformiv</td>
<td>$0 \times Z$</td>
<td>Uniform block index</td>
<td>-</td>
<td>GetActiveUniform</td>
<td>2.11.7</td>
</tr>
<tr>
<td>GetActiveUniformiv</td>
<td>$0 \times Z$</td>
<td>Uniform buffer offset</td>
<td>-</td>
<td>GetActiveUniform</td>
<td>2.11.7</td>
</tr>
</tbody>
</table>

Table 6.35. Program Object State (cont.)
<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIFORM_ARRAY_STRIDE</td>
<td>0 × *Z</td>
<td>GetActiveUniformsiv</td>
<td>-</td>
<td>Uniform buffer array stride</td>
<td>2.11.7</td>
</tr>
<tr>
<td>UNIFORM_MATRIX_ROW_MAJOR</td>
<td>0 × *Z</td>
<td>GetActiveUniformsiv</td>
<td>-</td>
<td>Uniform buffer intra-matrix stride</td>
<td>2.11.7</td>
</tr>
<tr>
<td>UNIFORM_MATRIX_ROW_MAJOR</td>
<td>0 × *Z</td>
<td>GetActiveUniformsiv</td>
<td>-</td>
<td>Whether uniform is a row-major matrix</td>
<td>2.11.7</td>
</tr>
<tr>
<td>UNIFORM_MATRIX_ROW_MAJOR</td>
<td>0 × *B</td>
<td>GetActiveUniformsiv</td>
<td>-</td>
<td>Uniform buffer binding points associated with the specified uniform block</td>
<td>2.11.7</td>
</tr>
<tr>
<td>UNIFORM_MATRIX_ROW_MAJOR</td>
<td>0 × *B</td>
<td>GetActiveUniformsiv</td>
<td>-</td>
<td>Size of the storage needed to hold this uniform block’s data</td>
<td>2.11.7</td>
</tr>
<tr>
<td>UNIFORM_MATRIX_ROW_MAJOR</td>
<td>0 × *B</td>
<td>GetActiveUniformsiv</td>
<td>-</td>
<td>Count of active uniforms in the specified uniform block</td>
<td>2.11.7</td>
</tr>
<tr>
<td>UNIFORM_MATRIX_ROW_MAJOR</td>
<td>0 × *B</td>
<td>GetActiveUniformsiv</td>
<td>-</td>
<td>Array of active uniform indices of the specified uniform block</td>
<td>2.11.7</td>
</tr>
<tr>
<td>UNIFORM_MATRIX_ROW_MAJOR</td>
<td>0 × *B</td>
<td>GetActiveUniformsiv</td>
<td>-</td>
<td>True if uniform block is actively referenced by the vertex stage</td>
<td>2.11.7</td>
</tr>
<tr>
<td>UNIFORM_MATRIX_ROW_MAJOR</td>
<td>0 × *B</td>
<td>GetActiveUniformsiv</td>
<td>-</td>
<td>True if uniform block is actively referenced by tess. control stage</td>
<td>2.11.7</td>
</tr>
<tr>
<td>UNIFORM_MATRIX_ROW_MAJOR</td>
<td>0 × *B</td>
<td>GetActiveUniformsiv</td>
<td>-</td>
<td>True if uniform block is actively referenced by tess evaluation stage</td>
<td>2.11.7</td>
</tr>
<tr>
<td>UNIFORM_MATRIX_ROW_MAJOR</td>
<td>0 × *B</td>
<td>GetActiveUniformsiv</td>
<td>-</td>
<td>True if uniform block is actively referenced by the geometry stage</td>
<td>2.11.7</td>
</tr>
<tr>
<td>UNIFORM_MATRIX_ROW_MAJOR</td>
<td>0 × *B</td>
<td>GetActiveUniformsiv</td>
<td>-</td>
<td>True if uniform block is actively referenced by the fragment stage</td>
<td>2.11.7</td>
</tr>
</tbody>
</table>
### Table 6.37. Program Object State (cont.)

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TESS,CONTROL,OUTPUT,VERTICES</td>
<td>( Z^+ )</td>
<td>GetProgramiv</td>
<td>0</td>
<td>Output patch size for tess. control shader</td>
<td>2.12.1</td>
</tr>
<tr>
<td>TESS,GEN,MODE</td>
<td>( Z_3 )</td>
<td>GetProgramiv</td>
<td>QUADS</td>
<td>Base primitive type for tess. prim. generator</td>
<td>2.12.2</td>
</tr>
<tr>
<td>TESS,GEN,SPACING</td>
<td>( Z_3 )</td>
<td>GetProgramiv</td>
<td>EQUAL</td>
<td>Spacing of tess. prim. generator edge subdivision</td>
<td>2.12.2</td>
</tr>
<tr>
<td>TESS,GEN,VERTEX,ORDER</td>
<td>( Z_2 )</td>
<td>GetProgramiv</td>
<td>CCW</td>
<td>Order of vertices in primitives generated by tess. prim generator</td>
<td>2.12.2</td>
</tr>
<tr>
<td>TESS,GEN,POINT,MODE</td>
<td>( B )</td>
<td>GetProgramiv</td>
<td>FALSE</td>
<td>Tess. prim. generator emits points?</td>
<td>2.12.2</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Initial Value</td>
<td>Description</td>
<td>Sec.</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------------</td>
<td>---------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS</td>
<td>$5 \times Z^+$</td>
<td>GetProgramStageiv</td>
<td>0</td>
<td>Number of subroutine unif. locations in the shader</td>
<td>2.11.8</td>
</tr>
<tr>
<td>ACTIVE_SUBROUTINE_UNIFORMS</td>
<td>$5 \times Z^+$</td>
<td>GetProgramStageiv</td>
<td>0</td>
<td>Number of subroutine unif. variables in the shader</td>
<td>2.11.8</td>
</tr>
<tr>
<td>ACTIVE_SUBROUTINES</td>
<td>$5 \times Z^+$</td>
<td>GetProgramStageiv</td>
<td>0</td>
<td>Number of subroutine functions in the shader</td>
<td>2.11.8</td>
</tr>
<tr>
<td>ACTIVE_SUBROUTINE_UNIFORM_MAX_LENGTH</td>
<td>$5 \times Z^+$</td>
<td>GetProgramStageiv</td>
<td>0</td>
<td>Maximum subroutine uniform name length</td>
<td>2.11.8</td>
</tr>
<tr>
<td>ACTIVE_SUBROUTINE_MAX_LENGTH</td>
<td>$5 \times Z^+$</td>
<td>GetProgramStageiv</td>
<td>0</td>
<td>Maximum subroutine name length</td>
<td>2.11.8</td>
</tr>
<tr>
<td>NUM_COMPATIBLE_SUBROUTINES</td>
<td>$5 \times 0 \times Z^+$</td>
<td>GetActive-SubroutineUniformiv</td>
<td>-</td>
<td>Number of subroutines compatible with a sub. uniform</td>
<td>2.11.8</td>
</tr>
<tr>
<td>COMPATIBLE_SUBROUTINES</td>
<td>$5 \times 0 \times 0 \times Z^+$</td>
<td>GetActive-SubroutineUniformiv</td>
<td>-</td>
<td>List of subroutines compatible with a sub. uniform</td>
<td>2.11.8</td>
</tr>
<tr>
<td>UNIFORM_SIZE</td>
<td>$5 \times 0 \times Z^+$</td>
<td>GetActive-SubroutineUniformiv</td>
<td>-</td>
<td>Number of elements in sub. uniform array</td>
<td>2.11.8</td>
</tr>
<tr>
<td>UNIFORM_NAME_LENGTH</td>
<td>$5 \times 0 \times Z^+$</td>
<td>GetActive-SubroutineUniformiv</td>
<td>-</td>
<td>Length of sub. uniform name</td>
<td>2.11.8</td>
</tr>
<tr>
<td>-</td>
<td>$5 \times 0 \times S$</td>
<td>GetActive-SubroutineUniformName</td>
<td>-</td>
<td>Sub. uniform name string</td>
<td>2.11.8</td>
</tr>
<tr>
<td>-</td>
<td>$5 \times 0 \times S$</td>
<td>GetActive-SubroutineName</td>
<td>-</td>
<td>Length of subroutine name</td>
<td>2.11.8</td>
</tr>
<tr>
<td>-</td>
<td>$5 \times 0 \times S$</td>
<td>GetActive-SubroutineName</td>
<td>-</td>
<td>Subroutine name string</td>
<td>2.11.8</td>
</tr>
</tbody>
</table>
### Table 6.39. Program Object State (cont.)

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVE_ATOMIC_COUNTER_BUFFERS</td>
<td>$Z^+$</td>
<td>GetProgramiv</td>
<td>0</td>
<td>Number of active atomic counter buffers (AACBs) used by a program</td>
<td>2.11.7</td>
</tr>
<tr>
<td>ATOMIC_COUNTER_BUFFER_BINDING</td>
<td>$n \times Z^+$</td>
<td>GetActiveAtomicCounterBufferiv</td>
<td>-</td>
<td>Binding point associated with an AACB</td>
<td>2.11.7</td>
</tr>
<tr>
<td>ATOMIC_COUNTER_BUFFER_DATA_SIZE</td>
<td>$n \times Z^+$</td>
<td>GetActiveAtomicCounterBufferiv</td>
<td>-</td>
<td>Minimum size required by an AACB</td>
<td>2.11.7</td>
</tr>
<tr>
<td>ATOMIC_COUNTER_BUFFER_ACTIVE_ATOMIC_COUNTERS</td>
<td>$n \times Z^+$</td>
<td>GetActiveAtomicCounterBufferiv</td>
<td>-</td>
<td>Number of active atomic counters in an AACB</td>
<td>2.11.7</td>
</tr>
<tr>
<td>ATOMIC_COUNTER_BUFFER_ACTIVE_ATOMIC_COUNTERS_indices</td>
<td>$m \times n \times Z^+$</td>
<td>GetActiveAtomicCounterBufferiv</td>
<td>-</td>
<td>List of active atomic counters in an AACB</td>
<td>2.11.7</td>
</tr>
<tr>
<td>ATOMIC_COUNTER_BUFFER_REFERENCED_BY_VERTEX_SHADER</td>
<td>$n \times B$</td>
<td>GetActiveAtomicCounterBufferiv</td>
<td>FALSE</td>
<td>AACB has a counter used by vertex shaders</td>
<td>2.11.7</td>
</tr>
<tr>
<td>ATOMIC_COUNTER_BUFFER_REFERENCED_BY_TESS_CONTROL_SHADER</td>
<td>$n \times B$</td>
<td>GetActiveAtomicCounterBufferiv</td>
<td>FALSE</td>
<td>AACB has a counter used by tess. control shaders</td>
<td>2.11.7</td>
</tr>
<tr>
<td>ATOMIC_COUNTER_BUFFER_REFERENCED_BY_TESS_EVALUATION_SHADER</td>
<td>$n \times B$</td>
<td>GetActiveAtomicCounterBufferiv</td>
<td>FALSE</td>
<td>AACB has a counter used by tess. evaluation shaders</td>
<td>2.11.7</td>
</tr>
<tr>
<td>ATOMIC_COUNTER_BUFFER_REFERENCED_BY_GEOMETRY_SHADER</td>
<td>$n \times B$</td>
<td>GetActiveAtomicCounterBufferiv</td>
<td>FALSE</td>
<td>AACB has a counter used by geometry shaders</td>
<td>2.11.7</td>
</tr>
<tr>
<td>ATOMIC_COUNTER_BUFFER_REFERENCED_BY_FRAGMENT_SHADER</td>
<td>$n \times B$</td>
<td>GetActiveAtomicCounterBufferiv</td>
<td>FALSE</td>
<td>AACB has a counter used by fragment shaders</td>
<td>2.11.7</td>
</tr>
<tr>
<td>UNIFORM_ATOMIC_COUNTER_BUFFER_INDEX</td>
<td>$n \times Z^+$</td>
<td>GetActiveUniformsiv</td>
<td>-</td>
<td>AACB associated with an active uniform</td>
<td>2.11.7</td>
</tr>
</tbody>
</table>
### Table 6.40. Vertex and Geometry Shader State

<table>
<thead>
<tr>
<th>Get value</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT_VERTEX_ATTRIB</td>
<td></td>
<td>Current generic vertex attribute values</td>
<td>2.7</td>
</tr>
<tr>
<td>PROGRAM_POINT_SIZE</td>
<td>FALSE</td>
<td>Point size mode</td>
<td>3.4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetVertexAttrib</td>
<td>$16 \times R^4$</td>
<td>$B$</td>
</tr>
</tbody>
</table>
### Table 6.41. Query Object State

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetQueryObjectuiv</td>
<td>0 or FALSE</td>
<td>Query object result</td>
<td>6.1.7</td>
</tr>
<tr>
<td>GetQueryObjectiv</td>
<td>FALSE</td>
<td>Is the query object result available?</td>
<td>6.1.7</td>
</tr>
</tbody>
</table>

- **Table 6.41. Query Object State**
<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMAGE_BINDING_NAME</td>
<td>$8 \times Z^+$</td>
<td>GetIntegeri,v</td>
<td>0</td>
<td>Name of bound texture object</td>
<td>3.9.20</td>
</tr>
<tr>
<td>IMAGE_BINDING_LEVEL</td>
<td>$8 \times Z^+$</td>
<td>GetIntegeri,v</td>
<td>0</td>
<td>Level of bound texture object</td>
<td>3.9.20</td>
</tr>
<tr>
<td>IMAGE_BINDING_LAYERED</td>
<td>$8 \times B$</td>
<td>GetBooleani,v</td>
<td>FALSE</td>
<td>Texture object bound with multiple layers</td>
<td>3.9.20</td>
</tr>
<tr>
<td>IMAGE_BINDING_LAYER</td>
<td>$8 \times Z^+$</td>
<td>GetIntegeri,v</td>
<td>0</td>
<td>Layer of bound texture, if not layered</td>
<td>3.9.20</td>
</tr>
<tr>
<td>IMAGE_BINDING_ACCESS</td>
<td>$8 \times Z_3$</td>
<td>GetIntegeri,v</td>
<td>READ_ONLY</td>
<td>Read and/or write access for bound texture</td>
<td>3.9.20</td>
</tr>
<tr>
<td>IMAGE_BINDING_FORMAT</td>
<td>$8 \times Z^+$</td>
<td>GetIntegeri,v</td>
<td>R8</td>
<td>Format used for accesses to bound texture</td>
<td>3.9.20</td>
</tr>
</tbody>
</table>
### Table 6.43. Transform Feedback State

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSFORM_FEEDBACK_BUFFER_BINDING</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Buffer object bound to generic bind point for transform feedback</td>
<td>6.1.9</td>
</tr>
<tr>
<td>TRANSFORM_FEEDBACK_BUFFER_BINDING</td>
<td>$n \times Z^+$</td>
<td>GetIntegeri_v</td>
<td>0</td>
<td>Buffer object bound to each transform feedback attribute stream</td>
<td>6.1.9</td>
</tr>
<tr>
<td>TRANSFORM_FEEDBACK_BUFFER_START</td>
<td>$n \times Z^+$</td>
<td>GetInteger64i_v</td>
<td>0</td>
<td>Start offset of binding range for each transform feedback attrib. stream</td>
<td>6.1.9</td>
</tr>
<tr>
<td>TRANSFORM_FEEDBACK_BUFFER_SIZE</td>
<td>$n \times Z^+$</td>
<td>GetInteger64i_v</td>
<td>0</td>
<td>Size of binding range for each transform feedback attrib. stream</td>
<td>6.1.9</td>
</tr>
<tr>
<td>TRANSFORM_FEEDBACK_PAUSED</td>
<td>$B$</td>
<td>GetBooleanv</td>
<td>FALSE</td>
<td>Is transform feedback paused on this object?</td>
<td>6.1.9</td>
</tr>
<tr>
<td>TRANSFORM_FEEDBACK_ACTIVE</td>
<td>$B$</td>
<td>GetBooleanv</td>
<td>FALSE</td>
<td>Is transform feedback active on this object?</td>
<td>6.1.9</td>
</tr>
</tbody>
</table>
### Table 6.44. Atomic Counter State

<table>
<thead>
<tr>
<th>Get Value</th>
<th>Get Command</th>
<th>Type</th>
<th>Description</th>
<th>Initial Value</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATOMIC-COUNTER-BUFFER-BINDING</td>
<td>GetIntegerv</td>
<td>$\mathbb{Z}^+$</td>
<td>Current value of generic atomic counter buffer binding</td>
<td>0</td>
<td>2.9.9</td>
</tr>
<tr>
<td>ATOMIC-COUNTER-BUFFER-BINDING</td>
<td>GetIntegeri.v</td>
<td>$\mathbb{Z}^+$</td>
<td>Buffer object bound to each atomic counter buffer binding point</td>
<td>0</td>
<td>2.9.9</td>
</tr>
<tr>
<td>ATOMIC-COUNTER-BUFFER-START</td>
<td>GetInteger64i.v</td>
<td>$\mathbb{Z}^+$</td>
<td>Start offset of binding range for each atomic counter buffer</td>
<td>0</td>
<td>2.9.9</td>
</tr>
<tr>
<td>ATOMIC-COUNTER-BUFFER-SIZE</td>
<td>GetInteger64i.v</td>
<td>$\mathbb{Z}^+$</td>
<td>Size of binding range for each atomic counter buffer</td>
<td>0</td>
<td>2.9.9</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Initial Value</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>-------------</td>
<td>---------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>OBJECT</td>
<td>Z</td>
<td>GetSynciv</td>
<td>SYNC_FENCE</td>
<td>Type of sync object</td>
<td></td>
</tr>
<tr>
<td>STATUS</td>
<td>Z</td>
<td>GetSynciv</td>
<td>UNSIGNALED</td>
<td>Sync object status</td>
<td></td>
</tr>
<tr>
<td>CONDITION</td>
<td>Z</td>
<td>GetSynciv</td>
<td>SYNC_GPU_COMMANDS_COMPLETE</td>
<td>Sync object condition</td>
<td></td>
</tr>
<tr>
<td>FLAGS</td>
<td>Z</td>
<td>GetSynciv</td>
<td>0</td>
<td>Sync object flags</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.45. Sync (state per sync object)
<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINE_SMOOTH_HINT</td>
<td>Z_3</td>
<td>GetIntegerv</td>
<td>DON'T CARE</td>
<td>Line smooth hint</td>
<td>5.4</td>
</tr>
<tr>
<td>POLYGON_SMOOTH_HINT</td>
<td>Z_3</td>
<td>GetIntegerv</td>
<td>DON'T CARE</td>
<td>Polygon smooth hint</td>
<td>5.4</td>
</tr>
<tr>
<td>TEXTURE_COMPRESSION_HINT</td>
<td>Z_3</td>
<td>GetIntegerv</td>
<td>DON'T CARE</td>
<td>Texture compression quality hint</td>
<td>5.4</td>
</tr>
<tr>
<td>FRAGMENT_SHADER_DERIVATIVE_HINT</td>
<td>Z_3</td>
<td>GetIntegerv</td>
<td>DON'T CARE</td>
<td>Fragment shader derivative accuracy hint</td>
<td>5.4</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Minimum Value</td>
<td>Description</td>
<td>Sec.</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>------</td>
<td>-------------</td>
<td>---------------</td>
<td>-------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>MAX_CLIP_distances</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>8</td>
<td>Maximum number of user clipping planes</td>
<td>2.20</td>
</tr>
<tr>
<td>SUBPIXEL_BITS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>4</td>
<td>Number of bits of subpixel precision in screen $x_w$ and $y_w$</td>
<td>3</td>
</tr>
<tr>
<td>MAX_3D_TEXTURE_SIZE</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>2048</td>
<td>Maximum 3D texture image dimension</td>
<td>3.9.3</td>
</tr>
<tr>
<td>MAX_TEXTURE_SIZE</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>16384</td>
<td>Maximum 2D/1D texture image dimension</td>
<td>3.9.3</td>
</tr>
<tr>
<td>MAX_ARRAY_TEXTURE_LAYERS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>2048</td>
<td>Maximum number of layers for texture arrays</td>
<td>3.9.3</td>
</tr>
<tr>
<td>MAX_TEXTURE_LOD_BIAS</td>
<td>$R^+$</td>
<td>GetFloatv</td>
<td>2.0</td>
<td>Maximum absolute texture level of detail bias</td>
<td>3.9.11</td>
</tr>
<tr>
<td>MAX_CUBE_MAP_TEXTURE_SIZE</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>16384</td>
<td>Maximum cube map texture image dimension</td>
<td>3.9.3</td>
</tr>
<tr>
<td>MAX_RENDERBUFFER_SIZE</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>16384</td>
<td>Maximum width and height of renderbuffers</td>
<td>4.4.2</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Minimum Value</td>
<td>Description</td>
<td>Sec.</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>-------------</td>
<td>---------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>MAX_VIEWPORT_DIMS</td>
<td>$2 \times Z^+$</td>
<td>GetFloatv</td>
<td>see 2.14.1</td>
<td>Maximum viewport dimensions</td>
<td>2.14.1</td>
</tr>
<tr>
<td>MAX_VIEWPORTS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>16</td>
<td>Maximum number of active viewports</td>
<td>2.14.1</td>
</tr>
<tr>
<td>VIEWPORT_SUBPIXEL_BITS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>No. of bits of sub-pixel precision for viewport bounds</td>
<td>2.14.1</td>
</tr>
<tr>
<td>VIEWPORT_BOUNDS_RANGE</td>
<td>$2 \times R$</td>
<td>GetFloatv</td>
<td></td>
<td>Viewport bounds range $[min, max]$ ($\geq [-32768, 32767]$)</td>
<td>2.14.1</td>
</tr>
<tr>
<td>LAYER_PROVOKING_VERTEX</td>
<td>$Z_4$</td>
<td>GetIntegerv</td>
<td>see 2.13.4</td>
<td>Vertex convention followed by gl_Layer</td>
<td>2.13.4</td>
</tr>
<tr>
<td>VIEWPORT_INDEX_PROVOKING_VERTEX</td>
<td>$Z_4$</td>
<td>GetIntegerv</td>
<td>see 2.13.4</td>
<td>Vertex convention followed by gl_ViewportIndex</td>
<td>2.13.4</td>
</tr>
<tr>
<td>POINT_SIZE_RANGE</td>
<td>$2 \times R^+$</td>
<td>GetFloatv</td>
<td>1,1</td>
<td>Range (lo to hi) of point sprite sizes</td>
<td>3.4</td>
</tr>
<tr>
<td>POINT_SIZE_GRANULARITY</td>
<td>$R^+$</td>
<td>GetFloatv</td>
<td>–</td>
<td>Point sprite size granularity</td>
<td>3.4</td>
</tr>
<tr>
<td>ALIASED_LINE_WIDTH_RANGE</td>
<td>$2 \times R^+$</td>
<td>GetFloatv</td>
<td>1,1</td>
<td>Range (lo to hi) of aliased line widths</td>
<td>3.5</td>
</tr>
<tr>
<td>SMOOTH_LINE_WIDTH_RANGE</td>
<td>$2 \times R^+$</td>
<td>GetFloatv</td>
<td>1,1</td>
<td>Range (lo to hi) of antialiased line widths</td>
<td>3.5</td>
</tr>
<tr>
<td>SMOOTH_LINE_WIDTH_GRANULARITY</td>
<td>$R^+$</td>
<td>GetFloatv</td>
<td>–</td>
<td>Antialiased line width granularity</td>
<td>3.5</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Minimum Value</td>
<td>Description</td>
<td>Sec.</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------</td>
<td>-------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>MAX_ELEMENTS_INDICES</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>–</td>
<td>Recommended max. number of DrawRangeElements indices</td>
<td>2.8</td>
</tr>
<tr>
<td>MAX_ELEMENTS_VERTICES</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>–</td>
<td>Recommended max. number of DrawRangeElements vertices</td>
<td>2.8</td>
</tr>
<tr>
<td>COMPRESSIONED_TEXTURE_FORMATS</td>
<td>$4 \times Z^+$</td>
<td>GetIntegerv</td>
<td>–</td>
<td>Enumerated compressed texture formats</td>
<td>3.9.5</td>
</tr>
<tr>
<td>NUM_COMPRESSED_TEXTURE_FORMATS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Number of compressed texture formats</td>
<td>3.9.5</td>
</tr>
<tr>
<td>MAX_TEXTURE_BUFFER_SIZE</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>65536</td>
<td>No. of addressable texels for buffer textures</td>
<td>3.9.7</td>
</tr>
<tr>
<td>MAX_RECTANGLE_TEXTURE_SIZE</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>16384</td>
<td>Max. width &amp; height of rectangular textures</td>
<td>3.9.3</td>
</tr>
<tr>
<td>PROGRAM_BINARY_FORMATS</td>
<td>$0 \times Z^+$</td>
<td>GetIntegerv</td>
<td>N/A</td>
<td>Enumerated program binary formats</td>
<td>2.11.5</td>
</tr>
<tr>
<td>NUM_PROGRAM_BINARY_FORMATS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Number of program binary formats</td>
<td>2.11.5</td>
</tr>
<tr>
<td>SHADER_BINARY_FORMATS</td>
<td>$0 \times Z^+$</td>
<td>GetIntegerv</td>
<td>–</td>
<td>Enumerated shader binary formats</td>
<td>2.11.2</td>
</tr>
<tr>
<td>NUM_SHADER_BINARY_FORMATS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Number of shader binary formats</td>
<td>2.11.2</td>
</tr>
<tr>
<td>SHADER_COMPILER</td>
<td>$B$</td>
<td>GetBooleanv</td>
<td>–</td>
<td>Shader compiler supported</td>
<td>2.11</td>
</tr>
<tr>
<td>MIN_MAP_BUFFER_ALIGNMENT</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>64</td>
<td>Min. byte alignment of pointers returned by Map*Buffer</td>
<td>2.9.3</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Command</td>
<td>Get</td>
<td>Command</td>
<td>Command</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>---------</td>
<td>-----</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>EXTENSIONS</td>
<td>0+ S</td>
<td>GetString</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>EXTENSIONS</td>
<td>Z+</td>
<td>GetInteger</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>MAJOR VERSION</td>
<td>Z+</td>
<td>GetInteger</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>MINOR VERSION</td>
<td>Z+</td>
<td>GetInteger</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>CONTEXT FLAGS</td>
<td>Z+</td>
<td>GetInteger</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>RENDERER</td>
<td>S</td>
<td>GetString</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>SHADING LANGUAGE VERSION</td>
<td>S</td>
<td>GetString</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>VENDOR</td>
<td>S</td>
<td>GetString</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>VERSION</td>
<td>S</td>
<td>GetString</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 6.50. Implementation Dependent Version and Extension Support
### Table 6.51. Implementation Dependent Vertex Shader Limits

<table>
<thead>
<tr>
<th>Get value</th>
<th>Get Command</th>
<th>Type</th>
<th>Minimum Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_VERTEX_ATTRIBS</td>
<td>GetIntegerv</td>
<td>( Z^+ )</td>
<td>16</td>
<td>Number of active vertex attributes</td>
</tr>
<tr>
<td>MAX_VERTEX_UNIFORM_COMPONENTS</td>
<td>GetIntegerv</td>
<td>( Z^+ )</td>
<td>1024</td>
<td>Number of components for vertex shader uniform variables</td>
</tr>
<tr>
<td>MAX_VERTEX_UNIFORM_BLOCKS</td>
<td>GetIntegerv</td>
<td>( Z^+ )</td>
<td>256</td>
<td>Number of vertex shader uniform variables</td>
</tr>
<tr>
<td>MAX_VERTEX_OUTPUT_COMPONENTS</td>
<td>GetIntegerv</td>
<td>( Z^+ )</td>
<td>12</td>
<td>Max number of vertex uniform buffers per program</td>
</tr>
<tr>
<td>MAX_VERTEX_ATOMIC_COUNTER_BUFFERS</td>
<td>GetIntegerv</td>
<td>( Z^+ )</td>
<td>0</td>
<td>Number of atomic counter buffers accessed by a vertex shader</td>
</tr>
<tr>
<td>MAX_VERTEX_ATOMIC_COUNTERS</td>
<td>GetIntegerv</td>
<td>( Z^+ )</td>
<td>0</td>
<td>Number of atomic counters accessed by a vertex shader</td>
</tr>
</tbody>
</table>

Sec. | 2.7 | 2.11.7 | 2.11.7 | 2.11.11 | 2.11.12 | 2.11.12
<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Minimum Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX:TESS:GIN:LEVEL</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>64</td>
<td>Max. level supported by tess. primitive generator</td>
<td>2.12.2</td>
</tr>
<tr>
<td>MAX:PATCH:VERTICES</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>32</td>
<td>Max. patch size</td>
<td>2.6.1</td>
</tr>
<tr>
<td>MAX:TESS:CONTROL:UNIFORM:COMPONENTS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>1024</td>
<td>No. of words for tess. control shader (TCS) uniforms</td>
<td>2.12.1</td>
</tr>
<tr>
<td>MAX:TESS:EVALUATION:UNIFORM:COMPONENTS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>1024</td>
<td>No. of words for tess. evaluation shader (TES) uniforms</td>
<td>2.12.3</td>
</tr>
<tr>
<td>MAX:TESS:CONTROL:TEXTURE:IMAGE:UNITS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>16</td>
<td>No. of tex. image units for TCS</td>
<td>2.11.12</td>
</tr>
<tr>
<td>MAX:TESS:EVALUATION:TEXTURE:IMAGE:UNITS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>16</td>
<td>No. of tex. image units for TES</td>
<td>2.11.12</td>
</tr>
<tr>
<td>MAX:TESS:CONTROL:OUTPUT:COMPONENTS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>128</td>
<td>No. components for TCS per-vertex outputs</td>
<td>2.12.1</td>
</tr>
<tr>
<td>MAX:TESS:PATCH:COMPONENTS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>120</td>
<td>No. components for TCS per-patch outputs</td>
<td>2.12.1</td>
</tr>
<tr>
<td>MAX:TESS:CONTROL:TOTAL:OUTPUT:COMPONENTS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>4096</td>
<td>No. components for TCS per-patch outputs</td>
<td>2.12.1</td>
</tr>
<tr>
<td>MAX:TESS:CONTROL:INPUT:COMPONENTS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>128</td>
<td>No. components for TCS per-vertex inputs</td>
<td>2.12.1</td>
</tr>
<tr>
<td>MAX:TESS:EVALUATION:INPUT:COMPONENTS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>128</td>
<td>No. components for TES per-vertex inputs</td>
<td>2.12.3</td>
</tr>
<tr>
<td>MAX:TESS:CONTROL:UNIFORM:BLOCKS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>12</td>
<td>No. of supported uniform blocks for TCS</td>
<td>2.11.7</td>
</tr>
<tr>
<td>MAX:TESS:EVALUATION:UNIFORM:BLOCKS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>12</td>
<td>No. of supported uniform blocks for TES</td>
<td>2.11.7</td>
</tr>
<tr>
<td>MAX:TESS:CONTROL:ATOMIC:COUNTER:BUFFERS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>No. of atomic counter (AC) buffers accessed by a TCS</td>
<td>2.11.7</td>
</tr>
<tr>
<td>MAX:TESS:EVALUATION:ATOMIC:COUNTER:BUFFERS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>No. of AC buffers accessed by a TES</td>
<td>2.11.12</td>
</tr>
<tr>
<td>MAX:TESS:CONTROL:ATOMIC:COUNTERS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>No. of ACs accessed by a TCS</td>
<td>2.11.7</td>
</tr>
<tr>
<td>MAX:TESS:EVALUATION:ATOMIC:COUNTERS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>No. of ACs accessed by a TES</td>
<td>2.11.12</td>
</tr>
</tbody>
</table>

Table 6.52. Implementation Dependent Tessellation Shader Limits
<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Minimum Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_GEOMETRY_UNIFORM_COMPONENTS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>512</td>
<td>Number of components for geom. shader uniform variables</td>
<td>2.13.3</td>
</tr>
<tr>
<td>MAX_GEOMETRY_UNIFORM_BLOCKS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>12</td>
<td>Max number of geometry uniform buffers per program</td>
<td>2.11.7</td>
</tr>
<tr>
<td>MAX_GEOMETRY_INPUT_COMPONENTS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>64</td>
<td>Max number of components of inputs read by a geometry shader</td>
<td>2.13.4</td>
</tr>
<tr>
<td>MAX_GEOMETRY_OUTPUT_COMPONENTS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>128</td>
<td>Max number of components of outputs written by a geometry shader</td>
<td>2.13.4</td>
</tr>
<tr>
<td>MAX_GEOMETRY_OUTPUT_VERTICES</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>256</td>
<td>Maximum number of vertices that any geometry shader can emit</td>
<td>2.13.4</td>
</tr>
<tr>
<td>MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>1024</td>
<td>Maximum number of total components (all vertices) of active outputs that a geometry shader can emit</td>
<td>2.13.4</td>
</tr>
<tr>
<td>MAX_GEOMETRY_TEXTURE_IMAGE_UNITS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>16</td>
<td>Number of texture image units accessible by a geometry shader</td>
<td>2.13.4</td>
</tr>
<tr>
<td>MAX_GEOMETRY_SHADER_INVOCATIONS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>32</td>
<td>Max supported geom. shader invocation count</td>
<td>2.13.4</td>
</tr>
<tr>
<td>MAX_VERTEX_STREAMS</td>
<td>Z+</td>
<td>GetInteger</td>
<td>4</td>
<td>Total number of vertex streams</td>
<td>2.13.4</td>
</tr>
<tr>
<td>MAX_GEOMETRY_ATOMIC_COUNTER_BUFFERS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Number of atomic counter buffers accessed by a geometry shader</td>
<td>2.11.7</td>
</tr>
<tr>
<td>MAX_GEOMETRY_ATOMIC_COUNTERS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Number of atomic counters accessed by a geometry shader</td>
<td>2.11.12</td>
</tr>
</tbody>
</table>
### Table 6.54. Implementation Dependent Fragment Shader Limits

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Type</th>
<th>Minimum Value</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetIntegerv</td>
<td>Z+</td>
<td>1024</td>
<td>GetIntegerv</td>
<td>Number of components for frag. shader uniform variables</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>Z+</td>
<td>256</td>
<td>GetIntegerv</td>
<td>Number of vectors for frag. shader uniform variables</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>Z+</td>
<td>12</td>
<td>GetIntegerv</td>
<td>Max number of fragment uniform buffers per program</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>Z+</td>
<td>128</td>
<td>GetIntegerv</td>
<td>Max number of components of inputs read by a fragment shader</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>Z+</td>
<td>16</td>
<td>GetIntegerv</td>
<td>Number of texture image units accessible by a fragment shader</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>Z+</td>
<td>-8</td>
<td>GetIntegerv</td>
<td>Min. level offset for textureGather</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>Z+</td>
<td>7</td>
<td>GetIntegerv</td>
<td>Max. level offset for textureGather</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>Z+</td>
<td>1</td>
<td>GetIntegerv</td>
<td>Number of atomic counter buffers accessed by a fragment shader</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>Z+</td>
<td>8</td>
<td>GetIntegerv</td>
<td>Number of atomic counters accessed by a fragment shader</td>
</tr>
</tbody>
</table>

- **MAX_FRAGMENT_UNIFORM_COMPONENTS**: Number of components for frag. shader uniform variables
- **MAX_FRAGMENT_UNIFORM_VARIABLES**: Number of vectors for frag. shader uniform variables
- **MAX_FRAGMENT_UNIFORM_BLOCKS**: Max number of fragment uniform buffers per program
- **MAX_FRAGMENT_INPUT_COMPONENTS**: Max number of components of inputs read by a fragment shader
- **MAX_TEXTURE_IMAGE_UNITS**: Number of texture image units accessible by a fragment shader
- **MIN_PROGRAM_TEXTURE_GATHER_OFFSET**: Min. level offset for textureGather
- **MAX_PROGRAM_TEXTURE_GATHER_OFFSET**: Max. level offset for textureGather
- **MAX_ATOMIC_COUNTER_BUFFERS**: Number of atomic counter buffers accessed by a fragment shader
- **MAX_ATOMIC_COUNTERS**: Number of atomic counters accessed by a fragment shader
### Table 6.55. Implementation Dependent Aggregate Shader Limits

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Minimum Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN_PROGRAM_TEXEL_OFFSET</td>
<td>Z</td>
<td>GetIntegerv</td>
<td>-8</td>
<td>Minimum texel offset allowed in lookup</td>
<td>2.11.12</td>
</tr>
<tr>
<td>MAX_PROGRAM_TEXEL_OFFSET</td>
<td>Z</td>
<td>GetIntegerv</td>
<td>7</td>
<td>Maximum texel offset allowed in lookup</td>
<td>2.11.12</td>
</tr>
<tr>
<td>MAX_UNIFORM_BUFFER_BINDINGS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>60</td>
<td>Max number of uniform buffer binding points on the context</td>
<td>2.11.7</td>
</tr>
<tr>
<td>MAX_UNIFORM_BLOCK_SIZE</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>16384</td>
<td>Max size in basic machine units of a uniform block</td>
<td>2.11.7</td>
</tr>
<tr>
<td>UNIFORM_BUFFER_OFFSET_ALIGNMENT</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>1</td>
<td>Minimum required alignment for uniform buffer sizes and offsets</td>
<td>2.11.7</td>
</tr>
<tr>
<td>MAX_COMBINED_UNIFORM_BLOCKS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>60</td>
<td>Max number of uniform buffers per program</td>
<td>2.11.7</td>
</tr>
<tr>
<td>MAX_VARYING_COMPONENTS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>60</td>
<td>Number of components for output variables</td>
<td>2.11.11</td>
</tr>
<tr>
<td>MAX_VARYING_VECTORS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>15</td>
<td>Number of vectors for output variables</td>
<td>2.11.11</td>
</tr>
<tr>
<td>MAX_COMBINED_TEXTURE_IMAGE_UNITS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>80</td>
<td>Total number of texture units accessible by the GL</td>
<td>2.11.12</td>
</tr>
<tr>
<td>MAX_SUBROUTINES</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>256</td>
<td>Maximum number of subroutines per shader stage</td>
<td>2.11.8</td>
</tr>
<tr>
<td>MAX_SUBROUTINE_UNIFORM_LOCATIONS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>1024</td>
<td>Maximum number of subroutine uniform locations per stage</td>
<td>2.11.8</td>
</tr>
<tr>
<td>MAX_ATOMIC_COUNTER_BUFFER_BINDINGS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>1</td>
<td>Max. number of atomic counter buffer bindings</td>
<td>2.9.9</td>
</tr>
<tr>
<td>MAX_ATOMIC_COUNTER_BUFFER_SIZE</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>32</td>
<td>Maximum size in basic machine units of an atomic counter buffer</td>
<td>2.11.7</td>
</tr>
<tr>
<td>MAX_COMBINED_ATOMIC_COUNTERS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>1</td>
<td>Max. number of atomic counter buffers per program</td>
<td>2.11.7</td>
</tr>
<tr>
<td>MAX_COMBINED_ATOMIC_COUNTERS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>8</td>
<td>Max. number of atomic counter uniforms per program</td>
<td>2.11.12</td>
</tr>
</tbody>
</table>
### Table 6.56. Implementation Dependent Aggregate Shader Limits (cont.)

<table>
<thead>
<tr>
<th>Command</th>
<th>Type</th>
<th>Get Value</th>
<th>Minimum Value</th>
<th>Command</th>
<th>Type</th>
<th>Get Value</th>
<th>Minimum Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_IMAGE_UNITS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>8</td>
<td>MAX_IMAGE_UNITS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>8</td>
<td>Limit on active image units + fragment outputs for a texture level bound to an image unit.</td>
</tr>
<tr>
<td>MAX_IMAGE_SAMPLES</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>MAX_IMAGE_UNIFORMS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Number of image variables in vertex shaders.</td>
</tr>
<tr>
<td>MAX_FRAGMENT_IMAGE_UNIFORMS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>MAX_FRAGMENT_IMAGE_UNIFORMS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>8</td>
<td>Number of image variables in fragment shaders.</td>
</tr>
<tr>
<td>MAX_GEOMETRY_IMAGE_UNIFORMS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>MAX_GEOMETRY_IMAGE_UNIFORMS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>8</td>
<td>Number of image variables in all shaders.</td>
</tr>
<tr>
<td>MAX_TESS_CONTROL_IMAGE_UNIFORMS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>MAX_TESS_EVALUATION_IMAGE_UNIFORMS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Number of image variables in tessellation control shaders.</td>
</tr>
<tr>
<td>MAX_TESS_EVALUATION_IMAGE_UNIFORMS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>MAX_TESS_EVALUATION_IMAGE_UNIFORMS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Number of image variables in tessellation evaluation shaders.</td>
</tr>
</tbody>
</table>

Sec. 3.9.20
<table>
<thead>
<tr>
<th>Get Command</th>
<th>Type</th>
<th>Get value</th>
<th>Sec.</th>
<th>Description</th>
<th>Minimum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetIntegerv</td>
<td>Z+</td>
<td>MAX_COMBINED_VERTEX_UNIFORM_BLOCKS X MAX_UNIFORM_BLOCK_SIZE / 4 + MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS</td>
<td>2.11.7</td>
<td>No. of words for vertex shader uniform variables in all uniform blocks (including default)</td>
<td>†</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>Z+</td>
<td>MAX_COMBINED_GEOMETRY_UNIFORM_BLOCKS X MAX_UNIFORM_BLOCK_SIZE / 4 + MAX_COMBINED_GEOMETRY_UNIFORM_COMPONENTS</td>
<td>2.11.7</td>
<td>No. of words for geometry shader uniform variables in all uniform blocks (including default)</td>
<td>†</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>Z+</td>
<td>MAX_COMBINED_TESS_CONTROL_UNIFORM_BLOCKS X MAX_UNIFORM_BLOCK_SIZE / 4 + MAX_COMBINED_TESS_CONTROL_UNIFORM_COMPONENTS</td>
<td>2.12.1</td>
<td>No. of words for TCS uniform variables in all uniform blocks (including default)</td>
<td>†</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>Z+</td>
<td>MAX_COMBINED_TESS_EVALUATION_UNIFORM_BLOCKS X MAX_UNIFORM_BLOCK_SIZE / 4 + MAX_COMBINED_TESS_EVALUATION_UNIFORM_COMPONENTS</td>
<td>2.12.3</td>
<td>No. of words for TES uniform variables in all uniform blocks (including default)</td>
<td>†</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>Z+</td>
<td>MAX_COMBINED_FRAGMENT_UNIFORM_BLOCKS X MAX_UNIFORM_BLOCK_SIZE / 4 + MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS</td>
<td>2.11.7</td>
<td>No. of words for fragment shader uniform variables in all uniform blocks (including default)</td>
<td>†</td>
</tr>
</tbody>
</table>

† The minimum value for each stage is \(\text{MAX}_{\text{stage}}_{\text{UNIFORM BLOCKS}} \times \text{MAX}_{\text{UNIFORM BLOCK SIZE}} / 4 + \text{MAX}_{\text{stage}}_{\text{UNIFORM COMPONENTS}}\)
### 6.2. STATE TABLES

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Minimum Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetIntegerv</td>
<td>1</td>
<td>Maximum number of sample mask words</td>
<td>6.1.15</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>4</td>
<td>Max. number of samples supported for all non-integer formats</td>
<td>4.1.3</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>1</td>
<td>Max. number of samples supported for all color formats in a multisample texture</td>
<td>6.1.15</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>1</td>
<td>Max. number of samples supported for all depth/stencil formats in a multisample texture</td>
<td>6.1.15</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>1</td>
<td>Max. number of samples supported for all integer format multisample buffers</td>
<td>6.1.15</td>
</tr>
<tr>
<td>GetQueryiv</td>
<td>see 6.1.7</td>
<td>Asynchronous query counter bits</td>
<td>6.1.7</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>0</td>
<td>Maximum WaitSync timeout interval</td>
<td>5.3.1</td>
</tr>
</tbody>
</table>

#### Table 6.58. Implementation Dependent Values (cont.)
<table>
<thead>
<tr>
<th>Minimum Value</th>
<th>Get Command Type</th>
<th>Minimum Value</th>
<th>Get Command Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.5</td>
<td>GetFloatv</td>
<td>+0.5</td>
<td>GetFloatv</td>
</tr>
<tr>
<td>Furthest negative offset for interpolate-AtOffset</td>
<td>Furthest positive offset for interpolate-AtOffset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subpixel bits for interpolate-AtOffset</td>
<td>Maximum number of active draw buffers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of active draw buffers when using dual-source blending</td>
<td>Maximum number of FBO attachment points for color buffers</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.59. Implementation Dependent Values (cont.)
<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Minimum Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMPLES</td>
<td>0 × Z+</td>
<td>GetInternalformat</td>
<td>†</td>
<td>Supported sample counts † See section 6.1.15</td>
<td>6.1.15</td>
</tr>
<tr>
<td>NUM,SAMPLE,COUNTS</td>
<td>Z+</td>
<td>GetInternalformat</td>
<td>1</td>
<td>Number of supported sample counts</td>
<td>6.1.15</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Minimum Value</td>
<td>Description</td>
<td>Sec.</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------</td>
<td>-------------</td>
<td>---------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>MAX, TRANSFORM, FEEDBACK, INTERLEAVED, COMPONENTS</td>
<td>Z⁺</td>
<td>GetIntegerv</td>
<td>64</td>
<td>Max no. of components to write to a single buffer in interleaved mode</td>
<td>2.17</td>
</tr>
<tr>
<td>MAX, TRANSFORM, FEEDBACK, SEPARATE, ATTRIBS</td>
<td>Z⁺</td>
<td>GetIntegerv</td>
<td>4</td>
<td>Max no. of separate attributes or outputs that can be captured in transform feedback</td>
<td>2.17</td>
</tr>
<tr>
<td>MAX, TRANSFORM, FEEDBACK, SEPARATE, COMPONENTS</td>
<td>Z⁺</td>
<td>GetIntegerv</td>
<td>4</td>
<td>Max no. of components per attribute or output in separate mode</td>
<td>2.17</td>
</tr>
<tr>
<td>MAX, TRANSFORM, FEEDBACK, BUFFERS</td>
<td>Z⁺</td>
<td>GetIntegerv</td>
<td>4</td>
<td>Max no. of buffer objs to write with transform feedback</td>
<td>2.17</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Minimum Value</td>
<td>Description</td>
<td>Sec.</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>-------------</td>
<td>---------------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>DOUBLEBUFFER</td>
<td>B</td>
<td>GetBooleanv</td>
<td>–</td>
<td>True if front &amp; back buffers exist</td>
<td>4.2.1</td>
</tr>
<tr>
<td>STEREO</td>
<td>B</td>
<td>GetBooleanv</td>
<td>–</td>
<td>True if left &amp; right buffers exist</td>
<td>6</td>
</tr>
<tr>
<td>SAMPLE, BUFFERS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Number of multisample buffers</td>
<td>3.3.1</td>
</tr>
<tr>
<td>SAMPLES</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Coverage mask size</td>
<td>3.3.1</td>
</tr>
<tr>
<td>SAMPLE_POSITION</td>
<td>$n \times 2 \times R^{(0,1)}$</td>
<td>GetMultisamplefv</td>
<td>–</td>
<td>Explicit sample positions</td>
<td>3.3.1</td>
</tr>
<tr>
<td>IMPLEMENTATION, COLOR, READ, TYPE</td>
<td>$Z_1$</td>
<td>GetIntegerv</td>
<td>–</td>
<td>Implementation preferred pixel type</td>
<td>4.3.1</td>
</tr>
<tr>
<td>IMPLEMENTATION, COLOR, READ, FORMAT</td>
<td>$Z_1$</td>
<td>GetIntegerv</td>
<td>–</td>
<td>Implementation preferred pixel format</td>
<td>4.3.1</td>
</tr>
</tbody>
</table>
### 6.2. *STATE TABLES*

<table>
<thead>
<tr>
<th>Command</th>
<th>Type</th>
<th>Get value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetIntegerv</td>
<td>$n \times Z$</td>
<td>$Z^+$</td>
<td>Buffer object bound to copy buffer</td>
</tr>
<tr>
<td>GetQueryiv</td>
<td>$n \times B$</td>
<td>$Z^+$</td>
<td>Buffer object bound to copy buffer</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>$Z^+$</td>
<td>$Z^+$</td>
<td>Active query object names</td>
</tr>
<tr>
<td>GetQueryiv</td>
<td>$Z^+$</td>
<td>$Z^+$</td>
<td>True if there is a corresponding current error code(s)</td>
</tr>
<tr>
<td>GetError</td>
<td>$n \times Z$</td>
<td>$Z$</td>
<td>Current error code(s)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Texture Cube Map</td>
<td>FALSE</td>
<td>Seamless cube map filtering enable</td>
</tr>
</tbody>
</table>

Table 6.63. Miscellaneous
Appendix A

Invariance

The OpenGL specification is not pixel exact. It therefore does not guarantee an exact match between images produced by different GL implementations. However, the specification does specify exact matches, in some cases, for images produced by the same implementation. The purpose of this appendix is to identify and provide justification for those cases that require exact matches.

A.1 Repeatability

The obvious and most fundamental case is repeated issuance of a series of GL commands. For any given GL and framebuffer state vector, and for any GL command, the resulting GL and framebuffer state must be identical whenever the command is executed on that initial GL and framebuffer state. This repeatability requirement doesn’t apply when using shaders containing side effects (image stores, image atomic operations, and atomic counter operations), because these memory operations are not guaranteed to be processed in a defined order.

One purpose of repeatability is avoidance of visual artifacts when a double-buffered scene is redrawn. If rendering is not repeatable, swapping between two buffers rendered with the same command sequence may result in visible changes in the image. Such false motion is distracting to the viewer. Another reason for repeatability is testability.

Repeatability, while important, is a weak requirement. Given only repeatability as a requirement, two scenes rendered with one (small) polygon changed in position might differ at every pixel. Such a difference, while within the law of repeatability, is certainly not within its spirit. Additional invariance rules are desirable to ensure useful operation.
A.2 Multi-pass Algorithms

Invariance is necessary for a whole set of useful multi-pass algorithms. Such algorithms render multiple times, each time with a different GL mode vector, to eventually produce a result in the framebuffer. Examples of these algorithms include:

- "Erasing" a primitive from the framebuffer by redrawing it, either in a different color or using the XOR logical operation.
- Using stencil operations to compute capping planes.

On the other hand, invariance rules can greatly increase the complexity of high-performance implementations of the GL. Even the weak repeatability requirement significantly constrains a parallel implementation of the GL. Because GL implementations are required to implement ALL GL capabilities, not just a convenient subset, those that utilize hardware acceleration are expected to alternate between hardware and software modules based on the current GL mode vector. A strong invariance requirement forces the behavior of the hardware and software modules to be identical, something that may be very difficult to achieve (for example, if the hardware does floating-point operations with different precision than the software).

What is desired is a compromise that results in many compliant, high-performance implementations, and in many software vendors choosing to port to OpenGL.

A.3 Invariance Rules

For a given instantiation of an OpenGL rendering context:

**Rule 1** For any given GL and framebuffer state vector, and for any given GL command, the resulting GL and framebuffer state must be identical each time the command is executed on that initial GL and framebuffer state.

**Rule 2** Changes to the following state values have no side effects (the use of any other state value is not affected by the change):

**Required:**

- Framebuffer contents (all bitplanes)
- The color buffers enabled for writing
- Scissor parameters (other than enable)
A.3. INVARIANCE RULES

- Write masks (color, depth, stencil)
- Clear values (color, depth, stencil)

**Strongly suggested:**

- Stencil parameters (other than enable)
- Depth test parameters (other than enable)
- Blend parameters (other than enable)
- Logical operation parameters (other than enable)
- Pixel storage state
- Polygon offset parameters (other than enables, and except as they affect the depth values of fragments)

**Corollary 1** Fragment generation is invariant with respect to the state values marked with • in Rule 2.

**Rule 3** The arithmetic of each per-fragment operation is invariant except with respect to parameters that directly control it.

**Corollary 2** Images rendered into different color buffers sharing the same framebuffer, either simultaneously or separately using the same command sequence, are pixel identical.

**Rule 4** The same vertex or fragment shader will produce the same result when run multiple times with the same input. The wording ‘the same shader’ means a program object that is populated with the same source strings, which are compiled and then linked, possibly multiple times, and which program object is then executed using the same GL state vector. Invariance is relaxed for shaders with side effects, such as accessing atomic counters (see section A.5).

**Rule 5** All fragment shaders that either conditionally or unconditionally assign \( \text{gl}_\text{FragCoord}.z \) to \( \text{gl}_\text{FragDepth} \) are depth-invariant with respect to each other, for those fragments where the assignment to \( \text{gl}_\text{FragDepth} \) actually is done.

If a sequence of GL commands specifies primitives to be rendered with shaders containing side effects (image stores, image atomic operations, atomic counter operations), invariance rules are relaxed. In particular, Rule 1, Corollary 3, and Rule 4 do not apply in the presence of shader side effects.

The following weaker versions of Rule 1 and 4 apply to GL commands involving shader side effects:

OpenGL 4.2 (Core Profile) - April 27, 2012
Rule 6 For any given GL and framebuffer state vector, and for any given GL command, the contents of any framebuffer state not directly or indirectly affected by results of shader image stores, atomic operations, or atomic counter operations must be identical each time the command is executed on that initial GL and framebuffer state.

Rule 7 The same vertex or fragment shader will produce the same result when run multiple times with the same input as long as:

- shader invocations do not use image atomic operations or atomic counters;
- no framebuffer memory is written to more than once by image stores, unless all such stores write the same value; and
- no shader invocation, or other operation performed to process the sequence of commands, reads memory written to by an image store.

When any sequence of GL commands triggers shader invocations that perform image stores, atomic operations, or atomic counter operations, and subsequent GL commands read the memory written by those shader invocations, these operations must be explicitly synchronized. For more details, see section 2.11.13.

A.4 Tessellation Invariance

When using a program containing tessellation evaluation shaders, the fixed-function tessellation primitive generator consumes the input patch specified by an application and emits a new set of primitives. The following invariance rules are intended to provide repeatability guarantees. Additionally, they are intended to allow an application with a carefully crafted tessellation evaluation shader to ensure that the sets of triangles generated for two adjacent patches have identical vertices along shared patch edges, avoiding “cracks” caused by minor differences in the positions of vertices along shared edges.

Rule 1 When processing two patches with identical outer and inner tessellation levels, the tessellation primitive generator will emit an identical set of point, line, or triangle primitives as long as the active program used to process the patch primitives has tessellation evaluation shaders specifying the same tessellation mode, spacing, vertex order, and point mode input layout qualifiers. Two sets of primitives are considered identical if and only if they contain the same number and type of primitives and the generated tessellation coordinates for the vertex numbered \( m \) of the primitive numbered \( n \) are identical for all values of \( m \) and \( n \).
A.4. TESSELLATION INVARIANCE

Rule 2 The set of vertices generated along the outer edge of the subdivided primitive in triangle and quad tessellation, and the tessellation coordinates of each, depends only on the corresponding outer tessellation level and the spacing input layout qualifier in the tessellation evaluation shader of the active program.

Rule 3 The set of vertices generated when subdividing any outer primitive edge is always symmetric. For triangle tessellation, if the subdivision generates a vertex with tessellation coordinates of the form $(0, x, 1-x)$, $(x, 0, 1-x)$, or $(x, 1-x, 0)$, it will also generate a vertex with coordinates of exactly $(0, 1-x, x)$, $(1-x, 0, x)$, or $(1-x, x, 0)$, respectively. For quad tessellation, if the subdivision generates a vertex with coordinates of $(x, 0)$ or $(0, x)$, it will also generate a vertex with coordinates of exactly $(1-x, 0)$ or $(0, 1-x)$, respectively. For isoline tessellation, if it generates vertices at $(0, x)$ and $(1, x)$ where $x$ is not zero, it will also generate vertices at exactly $(0, 1-x)$ and $(1, 1-x)$, respectively.

Rule 4 The set of vertices generated when subdividing outer edges in triangular and quad tessellation must be independent of the specific edge subdivided, given identical outer tessellation levels and spacing. For example, if vertices at $(x, 1-x, 0)$ and $(1-x, x, 0)$ are generated when subdividing the $w = 0$ edge in triangular tessellation, vertices must be generated at $(x, 0, 1-x)$ and $(1-x, 0, x)$ when subdividing an otherwise identical $v = 0$ edge. For quad tessellation, if vertices at $(x, 0)$ and $(1-x, 0)$ are generated when subdividing the $v = 0$ edge, vertices must be generated at $(0, x)$ and $(0, 1-x)$ when subdividing an otherwise identical $u = 0$ edge.

Rule 5 When processing two patches that are identical in all respects enumerated in rule 1 except for vertex order, the set of triangles generated for triangle and quad tessellation must be identical except for vertex and triangle order. For each triangle $n_1$ produced by processing the first patch, there must be a triangle $n_2$ produced when processing the second patch each of whose vertices has the same tessellation coordinates as one of the vertices in $n_1$.

Rule 6 When processing two patches that are identical in all respects enumerated in rule 1 other than matching outer tessellation levels and/or vertex order, the set of interior triangles generated for triangle and quad tessellation must be identical in all respects except for vertex and triangle order. For each interior triangle $n_1$ produced by processing the first patch, there must be a triangle $n_2$ produced when processing the second patch each of whose vertices has the same tessellation coordinates as one of the vertices in $n_1$. A triangle produced by the tessellator is considered an interior triangle if none of its vertices lie on an outer edge of the subdivided primitive.

OpenGL 4.2 (Core Profile) - April 27, 2012
Rule 7 For quad and triangle tessellation, the set of triangles connecting an inner and outer edge depends only on the inner and outer tessellation levels corresponding to that edge and the spacing input layout qualifier.

Rule 8 The value of all defined components of \texttt{gl_TessCoord} will be in the range \([0, 1]\). Additionally, for any defined component \(x\) of \texttt{gl_TessCoord}, the results of computing \(1.0 - x\) in a tessellation evaluation shader will be exact. Some floating-point values in the range \([0, 1]\) may fail to satisfy this property, but such values may never be used as tessellation coordinate components.

### A.5 Atomic Counter Invariance

When using a program containing atomic counters, the following invariance rules are intended to provide repeatability guarantees but within certain constraints.

**Rule 1** When a single shader type within a program accesses an atomic counter with only \texttt{atomicCounterIncrement}, any individual shader invocation is guaranteed to get a unique value returned.

**Corollary 1** Also holds true with \texttt{atomicCounterDecrement}.

**Corollary 2** This does not hold true for \texttt{atomicCounter}.

**Corollary 3** Repeatability is relaxed. While a unique value is returned to the shader, even given the same initial state vector and buffer contents, it is not guaranteed that the same unique value will be returned for each individual invocation of a shader (for example, on any single vertex, or any single fragment). It is wholly the shader writer’s responsibility to respect this constraint.

**Rule 2** When two or more shader types within a program access an atomic counter with only \texttt{atomicCounterIncrement}, there is no repeatability of the ordering of operations between stages. For example, some number of vertices may be processed, then some number of fragments may be processed.

**Corollary 4** This also holds true with \texttt{atomicCounterDecrement} and \texttt{atomicCounter}.
A.6 What All This Means

Hardware accelerated GL implementations are expected to default to software operation when some GL state vectors are encountered. Even the weak repeatability requirement means, for example, that OpenGL implementations cannot apply hysteresis to this swap, but must instead guarantee that a given mode vector implies that a subsequent command always is executed in either the hardware or the software machine.

The stronger invariance rules constrain when the switch from hardware to software rendering can occur, given that the software and hardware renderers are not pixel identical. For example, the switch can be made when blending is enabled or disabled, but it should not be made when a change is made to the blending parameters.

Because floating-point values may be represented using different formats in different renderers (hardware and software), many OpenGL state values may change subtly when renderers are swapped. This is the type of state value change that invariance rule 1 seeks to avoid.
Appendix B

Corollaries

The following observations are derived from the body and the other appendixes of the specification. Absence of an observation from this list in no way impugns its veracity.

1. The error semantics of upward compatible OpenGL revisions may change, and features deprecated in a previous revision may be removed. Otherwise, only additions can be made to upward compatible revisions.

2. GL query commands are not required to satisfy the semantics of the Flush or the Finish commands. All that is required is that the queried state be consistent with complete execution of all previously executed GL commands.

3. Application specified point size and line width must be returned as specified when queried. Implementation-dependent clamping affects the values only while they are in use.

4. The mask specified as the third argument to StencilFunc affects the operands of the stencil comparison function, but has no direct effect on the update of the stencil buffer. The mask specified by StencilMask has no effect on the stencil comparison function; it limits the effect of the update of the stencil buffer.

5. There is no atomicity requirement for OpenGL rendering commands, even at the fragment level.

6. Because rasterization of non-antialiased polygons is point sampled, polygons that have no area generate no fragments when they are rasterized in FILL mode, and the fragments generated by the rasterization of “narrow” polygons may not form a continuous array.
7. OpenGL does not force left- or right-handedness on any of its coordinates systems.

8. (No pixel dropouts or duplicates.) Let two polygons share an identical edge. That is, there exist vertices A and B of an edge of one polygon, and vertices C and D of an edge of the other polygon; the positions of vertex A and C are identical; and the positions of vertex B and D are identical. Vertex positions are identical if the \texttt{gl\_Position} values output by the vertex (or if active, geometry) shader are identical. Then, when the fragments produced by rasterization of both polygons are taken together, each fragment intersecting the interior of the shared edge is produced exactly once.

9. Dithering algorithms may be different for different components. In particular, alpha may be dithered differently from red, green, or blue, and an implementation may choose to not dither alpha at all.
Appendix C

Compressed Texture Image Formats

C.1 RGTC Compressed Texture Image Formats

Compressed texture images stored using the RGTC compressed image encodings are represented as a collection of $4 \times 4$ texel blocks, where each block contains 64 or 128 bits of texel data. The image is encoded as a normal 2D raster image in which each $4 \times 4$ block is treated as a single pixel. If an RGTC image has a width or height that is not a multiple of four, the data corresponding to texels outside the image are irrelevant and undefined.

When an RGTC image with a width of $w$, height of $h$, and block size of block-size (8 or 16 bytes) is decoded, the corresponding image size (in bytes) is:

$$\left\lceil \frac{w}{4} \right\rceil \times \left\lceil \frac{h}{4} \right\rceil \times \text{blocksize}.$$  

When decoding an RGTC image, the block containing the texel at offset $(x, y)$ begins at an offset (in bytes) relative to the base of the image of:

$$\text{blocksize} \times \left( \left\lceil \frac{w}{4} \right\rceil \times \left\lceil \frac{y}{4} \right\rceil + \left\lfloor \frac{x}{4} \right\rfloor \right).$$

The data corresponding to a specific texel $(x, y)$ are extracted from a $4 \times 4$ texel block using a relative $(x, y)$ value of

$$(x \mod 4, y \mod 4).$$

There are four distinct RGTC image formats:
Each block of texels consists of 64 bits of unsigned red image data. Each red image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

\[\text{red}_0, \text{red}_1, \text{bits}_0, \text{bits}_1, \text{bits}_2, \text{bits}_3, \text{bits}_4, \text{bits}_5\]

The 6 bits\(_s\) bytes of the block are decoded into a 48-bit bit vector:

\[\text{bits} = \text{bits}_0 + 256 \times (\text{bits}_1 + 256 \times (\text{bits}_2 + 256 \times (\text{bits}_3 + 256 \times (\text{bits}_4 + 256 \times \text{bits}_5))))\]

\text{red}_0\ and\ \text{red}_1\ are\ 8-bit\ unsigned\ integers\ that\ are\ unpacked\ to\ red\ values\ \text{RED}_0\ and\ \text{RED}_1\ 
\text{bits}\ is\ a\ 48-bit\ unsigned\ integer,\ from\ which\ a\ three-bit\ control\ code\ is\ extracted\ for\ a\ texel\ at\ location\ (x, y)\ in\ the\ block\ using:\

\[\text{code}(x, y) = \text{bits} \cdot [3 \times (4 \times y + x) + 2 \ldots 3 \times (4 \times y + x) + 0]\]

where bit 47 is the most significant and bit 0 is the least significant bit. The red value \( R \) for a texel at location \((x, y)\) in the block is given by:

\[
\begin{align*}
R &= \begin{cases} 
\text{RED}_0, & \text{red}_0 > \text{red}_1, \text{code}(x, y) = 0 \\
\text{RED}_1, & \text{red}_0 > \text{red}_1, \text{code}(x, y) = 1 \\
6\text{RED}_0+\text{RED}_1, & \text{red}_0 > \text{red}_1, \text{code}(x, y) = 2 \\
5\text{RED}_0+2\text{RED}_1, & \text{red}_0 > \text{red}_1, \text{code}(x, y) = 3 \\
4\text{RED}_0+3\text{RED}_1, & \text{red}_0 > \text{red}_1, \text{code}(x, y) = 4 \\
3\text{RED}_0+4\text{RED}_1, & \text{red}_0 > \text{red}_1, \text{code}(x, y) = 5 \\
2\text{RED}_0+5\text{RED}_1, & \text{red}_0 > \text{red}_1, \text{code}(x, y) = 6 \\
\text{RED}_0+6\text{RED}_1, & \text{red}_0 > \text{red}_1, \text{code}(x, y) = 7 \\
\text{RED}_0, & \text{red}_0 \leq \text{red}_1, \text{code}(x, y) = 0 \\
\text{RED}_1, & \text{red}_0 \leq \text{red}_1, \text{code}(x, y) = 1 \\
4\text{RED}_0+\text{RED}_1, & \text{red}_0 \leq \text{red}_1, \text{code}(x, y) = 2 \\
3\text{RED}_0+2\text{RED}_1, & \text{red}_0 \leq \text{red}_1, \text{code}(x, y) = 3 \\
2\text{RED}_0+3\text{RED}_1, & \text{red}_0 \leq \text{red}_1, \text{code}(x, y) = 4 \\
\text{RED}_0+4\text{RED}_1, & \text{red}_0 \leq \text{red}_1, \text{code}(x, y) = 5 \\
\text{RED}_{\text{min}}, & \text{red}_0 \leq \text{red}_1, \text{code}(x, y) = 6 \\
\text{RED}_{\text{max}}, & \text{red}_0 \leq \text{red}_1, \text{code}(x, y) = 7
\end{cases}
\end{align*}
\]
C.1. RGTC COMPRESSED TEXTURE IMAGE FORMATS

\( RED_{\min} \) and \( RED_{\max} \) are 0.0 and 1.0 respectively.

Since the decoded texel has a red format, the resulting RGBA value for the
text is \((R, 0, 0, 1)\).

C.1.2 Format COMPRESSED_SIGNED_RED_RGTC1

Each \( 4 \times 4 \) block of texels consists of 64 bits of signed red image data. The red
values of a texel are extracted in the same way as COMPRESSED_RED_RGTC1 ex-
cept \( red_0, red_1, RED_0, RED_1, RED_{\min}, \) and \( RED_{\max} \) are defined as follows:

\[
\begin{align*}
red_0 \text{ and } red_1 & \text{ are 8-bit signed (twos complement) integers.} \\
RED_0 &= \begin{cases} 
\frac{red_0}{127}, & red_0 > -128 \\
-1.0, & red_0 = -128
\end{cases} \\
RED_1 &= \begin{cases} 
\frac{red_1}{127}, & red_1 > -128 \\
-1.0, & red_1 = -128
\end{cases}
\end{align*}
\]

\( RED_{\min} = -1.0 \)

\( RED_{\max} = 1.0 \)

CAVEAT for signed \( red_0 \) and \( red_1 \) values: the expressions \( red_0 > red_1 \) and
\( red_0 \leq red_1 \) above are considered undefined (read: may vary by implementation)
when \( red_0 = -127 \) and \( red_1 = -128 \). This is because if \( red_0 \) were remapped to
-127 prior to the comparison to reduce the latency of a hardware decompressor, the
expressions would reverse their logic. Encoders for the signed red-green formats
should avoid encoding blocks where \( red_0 = -127 \) and \( red_1 = -128 \).

C.1.3 Format COMPRESSED_RG_RGTC2

Each \( 4 \times 4 \) block of texels consists of 64 bits of compressed unsigned red image
data followed by 64 bits of compressed unsigned green image data.

The first 64 bits of compressed red are decoded exactly like COMPRESSED_-RED_RGTC1 above.

The second 64 bits of compressed green are decoded exactly like COMPRESSED_RED_RGTC1 above except the decoded value \( R \) for this second block is considered the resulting green value \( G \).

Since the decoded texel has a red-green format, the resulting RGBA value for the
texel is \((R, G, 0, 1)\).

OpenGL 4.2 (Core Profile) - April 27, 2012
C.1.4 Format COMPRESSED_SIGNED_RG_RGTC2

Each $4 \times 4$ block of texels consists of 64 bits of compressed signed red image data followed by 64 bits of compressed signed green image data.

The first 64 bits of compressed red are decoded exactly like COMPRESSED_SIGNED_RED_RGTC1 above.

The second 64 bits of compressed green are decoded exactly like COMPRESSED_SIGNED_RED_RGTC1 above except the decoded value $R$ for this second block is considered the resulting green value $G$.

Since this image has a red-green format, the resulting RGBA value is $(R, G, 0, 1)$.

C.2 BPTC Compressed Texture Image Formats

Compressed texture images stored using the BPTC compressed image formats are represented as a collection of $4 \times 4$ texel blocks, where each block contains 128 bits of texel data. The image is encoded as a normal 2D raster image in which each $4 \times 4$ block is treated as a single pixel. If a BPTC image has a width or height that is not a multiple of four, the data corresponding to texels outside the image are irrelevant and undefined.

When a BPTC image with a width of $w$, height of $h$, and block size of $\text{blocksize}$ (16 bytes) is decoded, the corresponding image size (in bytes) is:

$$\left\lceil \frac{w}{4} \right\rceil \times \left\lceil \frac{h}{4} \right\rceil \times \text{blocksize}.$$  

When decoding a BPTC image, the block containing the texel at offset $(x, y)$ begins at an offset (in bytes) relative to the base of the image of:

$$\text{blocksize} \times \left( \left\lceil \frac{w}{4} \right\rceil \times \left\lceil \frac{y}{4} \right\rceil + \left\lfloor \frac{x}{4} \right\rfloor \right).$$

The data corresponding to a specific texel $(x, y)$ are extracted from a $4 \times 4$ texel block using a relative $(x, y)$ value of $(x \mod 4, y \mod 4)$.

There are two distinct BPTC image formats each of which has two variants. COMPRESSED_RGBA_BPTC_UNORM and COMPRESSED_SRGB_ALPHA_BPTC_UNORM compress 8-bit fixed-point data. COMPRESSED_RGB_BPTC_SIGNED_FLOAT and COMPRESSED_RGB_BPTC_UNSIGNED_FLOAT compress
C.2. BPTC COMPRESSED TEXTURE IMAGE FORMATS

high dynamic range floating-point values. The formats are similar, so the description of the float format will reference significant sections of the UNORM description.

C.2.1 Formats

COMPRESSED_RGBA_BPTC_UNORM and COMPRESSED_SRGB_ALPHA_BPTC_UNORM

Each 4 × 4 block of texels consists of 128 bits of RGBA or SRGB_ALPHA image data.

Each block contains enough information to select and decode a pair of colors called endpoints, interpolate between those endpoints in a variety of ways, then remap the result into the final output.

Each block can contain data in one of eight modes. The mode is identified by the lowest bits of the lowest byte. It is encoded as zero or more zeros followed by a one. For example, using x to indicate a bit not included in the mode number, mode 0 is encoded as xxxxxx1 in the low byte in binary, mode 5 is xx100000, and mode 7 is 10000000. Encoding the low byte as zero is reserved and should not be used when encoding a BPTC texture.

All further decoding is driven by the values derived from the mode listed in table C.1. The fields in the block are always in the same order for all modes. Starting at the lowest bit after the mode and going up, these fields are: partition number, rotation, index selection, color, alpha, per-endpoint P-bit, shared P-bit, primary indices, and secondary indices. The number of bits to be read in each field is determined directly from the table.

Each block can be divided into between 1 and 3 groups of pixels with independent compression parameters called subsets. A texel in a block with one subset is always considered to be in subset zero. Otherwise, a number determined by the number of partition bits is used to look up in the partition tables C.2 or C.3 for 2 and 3 subsets respectively. This partitioning is indexed by the X and Y within the block to generate the subset index.

Each block has two colors for each subset, stored first by endpoint, then by subset, then by color. For example, a format with two subsets and five color bits would have five bits of red for endpoint 0 of the first subset, then five bits of red for endpoint 1, then the two ends of the second subset, then green and blue stored similarly. If a block has non-zero alpha bits, the alpha data follows the color data with the same organization. If not, alpha is overridden to 1.0. These bits are treated as the high bits of a fixed-point value in a byte. If the format has a shared P-bit, there are two bits for endpoints 0 and 1 from low to high. If the format has a per-endpoint P-bits, then there are 2*subsets P-bits stored in the same order as color and alpha. Both kinds of P-bits are added as a bit below the color data stored in the
byte. So, for a format with 5 red bits, the P-bit ends up in bit 2. For final scaling, the top bits of the value are replicated into any remaining bits in the byte. For the preceding example, bits 6 and 7 would be written to bits 0 and 1.

The endpoint colors are interpolated using index values stored in the block. The index bits are stored in x-major order. Each index has the number of bits indicated by the mode except for one special index per subset called the anchor index. Since the ordering of the endpoints is unimportant, we can save one bit on one index per subset by ordering the endpoints such that the highest bit is guaranteed to be zero. In partition zero, the anchor index is always index zero. In other partitions, the anchor index is specified by tables C.4, C.5, and C.6. If secondary index bits are present, they are read in the same manner. The anchor index information is only used to determine the number of bits each index has when it’s read from the block data.

The endpoint color and alpha values used for final interpolation are the decoded values corresponding to the applicable subset as selected above. The index value for interpolating color comes from the secondary index for the texel if the format has an index selection bit and its value is one and from the primary index otherwise. The alpha index comes from the secondary index if the block has a secondary index and the block either doesn’t have an index selection bit or that bit is zero and the primary index otherwise.

Interpolation is always performed using a 6-bit interpolation factor. The effective interpolation factors for 2, 3, and 4 bit indices are given below:

2 0, 21, 43, 64
3 0, 9, 18, 27, 37, 46, 55, 64
4 0, 4, 9, 13, 17, 21, 26, 30, 34, 38, 43, 47, 51, 55, 60, 64

The interpolation results in an RGBA color. If rotation bits are present, this color is remapped according to:

0 no change
1 swap(a,r)
2 swap(a,g)
3 swap(a,b)

These 8-bit values show up in the shader interpreted as either RGBA8 or SRGB8_ALPHA8 for COMPRESSED_RGBA_BPTC_UNORM and COMPRESSED_-SRGB_ALPHA_BPTC_UNORM respectively.

OpenGL 4.2 (Core Profile) - April 27, 2012
Table C.1: Mode-dependent BPTC parameters. The full descriptions of each column are as follows:
Mode: As described previously
NS: Number of subsets in each partition
PB: Partition bits
RB: Rotation bits
ISB: Index selection bits
CB: Color bits
AB: Alpha bits
EPB: Endpoint P-bits
SPB: Shared P-bits
IB: Index bits per element
IB2: Secondary index bits per element

<table>
<thead>
<tr>
<th>Mode</th>
<th>NS</th>
<th>PB</th>
<th>RB</th>
<th>ISB</th>
<th>CB</th>
<th>AB</th>
<th>EPB</th>
<th>SPB</th>
<th>IB</th>
<th>IB2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

OpenGL 4.2 (Core Profile) - April 27, 2012
Table C.2: Partition table for 2 subset. Each row is one 4 × 4 block.
C.2. BPTC COMRESSED TEXTURE IMAGE FORMATS

Table C.3: Partition table for 3 subset. Each row is one 4 x 4 block.
C.2. **BPTC COMPRESSED TEXTURE IMAGE FORMATS**

<table>
<thead>
<tr>
<th>15</th>
<th>15</th>
<th>15</th>
<th>15</th>
<th>15</th>
<th>15</th>
<th>15</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>8</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>15</td>
<td>15</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>15</td>
<td>15</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>2</td>
<td>2</td>
<td>15</td>
</tr>
</tbody>
</table>

Table C.4: Anchor index values for the second subset of two-subset partitioning. Values run right, then down.

<table>
<thead>
<tr>
<th>3</th>
<th>3</th>
<th>15</th>
<th>15</th>
<th>8</th>
<th>3</th>
<th>15</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>8</td>
<td>15</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>5</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>10</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>15</td>
<td>5</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>8</td>
<td>13</td>
<td>15</td>
<td>12</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Table C.5: Anchor index values for the second subset of three-subset partitioning. Values run right, then down.

<table>
<thead>
<tr>
<th>15</th>
<th>8</th>
<th>8</th>
<th>3</th>
<th>15</th>
<th>15</th>
<th>3</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>8</td>
</tr>
<tr>
<td>15</td>
<td>8</td>
<td>15</td>
<td>3</td>
<td>15</td>
<td>8</td>
<td>15</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>15</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>15</td>
<td>10</td>
<td>10</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td>8</td>
<td>15</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>3</td>
<td>15</td>
<td>15</td>
<td>8</td>
</tr>
</tbody>
</table>

Table C.6: Anchor index values for the third subset of three-subset partitioning. Values run right, then down.
C.2. BPTC COMPRESSED TEXTURE IMAGE FORMATS

C.2.2 Formats

COMPRESSED_RGB_BPTC_SIGNED_FLOAT and
COMPRESSED_RGB_BPTC_UNSIGNED_FLOAT

Each $4 \times 4$ block of texels consists of 128 bits of RGB data. These formats are very similar and will be described together. In the description and pseudocode below, signed will be used as a condition which is true for the SIGNED format and false for the UNSIGNED format. Both formats only contain RGB data, so the returned alpha value is 1.0. If a block uses a reserved or invalid encoding, the return value is $(0, 0, 0, 1)$.

Each block can contain data in one of 14 modes. The mode number is encoded in either the low two bits or the low five bits. If the low two bits are less than two, that is the mode number, otherwise the low five bits the mode number. Mode numbers not listed in table C.7 are reserved (19, 23, 27, and 31).

The data for the compressed blocks is stored in a different format for each mode. The formats are specified in table C.8. The format strings are intended to be read from left to right with the LSB on the left. Each element is of the form $v[a : b]$. If $a \geq b$, this indicates extracting $b - a + 1$ bits from the block at that location and put them in the corresponding bits of the variable $v$. If $a < b$, then the bits are reversed. $v[a]$ is used as a shorthand for the one bit $v[a : a]$. As an example, $m[1 : 0], g2[4]$ would move the low two bits from the block into the low two bits of $m$ then the next bit of the block into bit 4 of $g2$. The variable names given in the table will be referred to in the language below.

Subsets and indices work in much the same way as described for the fixed-point formats above. If a float block has no partition bits, then it is a single-subset block. If it has partition bits, then it is a 2 subset block. The partition index references the first half of table C.2. Indices are read in the same way as the fixed-point formats including obeying the anchor values for index 0 and as needed by table C.4.

In a single-subset blocks, the two endpoints are contained in $r_0, g_0, b_0$ (hence $e_0$) and $r_1, g_1, b_1$ (hence $e_1$). In a two-subset block, the endpoints for the second subset are in $r_2, g_2, b_2$ and $r_3, g_3, b_3$. The value in $e_0$ is sign-extended if the format of the texture is signed. The values in $e_1$ (and $e_2$ and $e_3$ if the block is two-subset) are sign-extended if the format of the texture is signed or if the block mode has transformed endpoints. If the mode has transformed endpoints, the values from $e_0$ are used as a base to offset all other endpoints, wrapped at the number of endpoint bits. For example, $r_1 = (r_0 + r_1) \& ((1 << EPB) - 1)$.

Next, the endpoints are unquantized to maximize the usage of the bits and to ensure that the negative ranges are oriented properly to interpolate as a two’s complement value. The following pseudocode assumes the computation is being done using sufficiently large intermediate values to avoid overflow. For the unsigned float format, we unquantize a value $x$ to $\text{unq}$ by:

$$\text{unq}(x) = (1 + x) \approx x$$
C.2. BPTC COMPRESSED TEXTURE IMAGE FORMATS

if (EPB >= 15)
    unq = x;
else if (x == 0)
    unq = 0;
else if (x == ((1<<EPB)-1))
    unq = 0xFFFF;
else
    unq = ((x << 15) + 0x4000) >> (EPB-1);

The signed float unquantization is similar, but needs to worry about orienting
the negative range:

    s = 0;
    if (EPB >= 16)
        unq = x;
    else {
        if (x < 0) {
            s = 1;
            x = -x;
        }
        if (x == 0)
            unq = 0;
        else if (x >= ((1<<(EPB-1))-1))
            unq = 0x7FFF;
        else
            unq = ((x << 15) + 0x4000) >> (EPB-1);
    }
    if (s)
        unq = -unq;

After the endpoints are unquantized, interpolation proceeds as in the fixed-
point formats above including the interpolation weight table.

The interpolated values are passed through a final unquantization step. For
the unsigned format, this step simply multiplies by $\frac{31}{64}$. The signed format negates
negative components, multiplies by $\frac{31}{32}$, then ORs in the sign bit if the original value
was negative.

The resultant value should be a legal 16-bit half float which is then returned as
a float to the shader.

OpenGL 4.2 (Core Profile) - April 27, 2012
Table C.7: Endpoint and partition parameters for block modes

<table>
<thead>
<tr>
<th>Mode Number</th>
<th>Transformed Endpoints</th>
<th>Partition Bits (PB)</th>
<th>Endpoint Bits (EPB)</th>
<th>Delta Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>5, 5, 5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>6, 6, 6</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>5</td>
<td>11</td>
<td>5, 4, 4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>5</td>
<td>11</td>
<td>4, 5, 4</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>5</td>
<td>11</td>
<td>4, 4, 5</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>5, 5, 5</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>6, 5, 5</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>5, 6, 5</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>5, 5, 6</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>5</td>
<td>6</td>
<td>6, 6, 6</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>10, 10, 10</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>9, 9, 9</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>12</td>
<td>8, 8, 8</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>0</td>
<td>16</td>
<td>4, 4, 4</td>
</tr>
<tr>
<td>Mode Number</td>
<td>Block Format</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>--------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>m[1:0], g2[4], b2[4], b3[4], r0[9:0], g0[9:0], b0[9:0], r1[4:0], g3[4], g2[3:0], g1[4:0], b3[0], g3[3:0], b1[4:0], b3[1], b2[3:0], r2[4:0], b3[2], r3[4:0], b3[3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>m[1:0], g2[5], g3[4], g3[5], r0[6:0], b3[0], b3[1], b2[4], g0[6:0], b2[5], b3[2], g2[4], b0[6:0], b3[3], b3[5], b3[4], r1[5:0], g2[3:0], g1[5:0], g3[3:0], b1[5:0], b2[3:0], r2[5:0], r3[5:0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>m[4:0], r0[9:0], g0[9:0], b0[9:0], r1[4:0], r0[10], g2[3:0], g1[3:0], g0[10], b3[0], g3[3:0], b1[3:0], b0[10], b3[1], b2[3:0], r2[4:0], b3[2], r3[4:0], b3[3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>m[4:0], r0[9:0], g0[9:0], r1[3:0], r0[10], g3[4], g2[3:0], g1[4:0], g0[10], g3[3:0], b1[3:0], b0[10], b3[1], b2[3:0], r2[3:0], b3[0], b3[2], r3[3:0], g2[4], b3[3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>m[4:0], r0[9:0], r1[3:0], r0[10], b2[4], g2[3:0], g1[3:0], g0[10], b3[0], g3[3:0], b1[4:0], b0[10], b2[3:0], r2[3:0], b3[1], b3[2], r3[3:0], b3[4], b3[3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>m[4:0], r0[8:0], b2[4], g0[8:0], g2[4], b0[8:0], b3[4], r1[4:0], g3[4], g2[3:0], g1[4:0], b3[0], g3[3:0], b1[4:0], b3[1], b2[3:0], r2[4:0], b3[2], r3[4:0], b3[3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>m[4:0], r0[7:0], g3[4], b2[4], g0[7:0], b3[2], g2[4], b0[7:0], b3[4], r1[5:0], g2[3:0], g1[4:0], b3[0], g3[3:0], b1[4:0], b3[1], b2[3:0], r2[5:0], r3[5:0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>m[4:0], r0[7:0], b3[0], b2[4], g0[7:0], g2[5], g2[4], b0[7:0], g3[5], b3[4], r1[4:0], g3[4], g2[3:0], g1[5:0], g3[3:0], b1[4:0], b3[1], b2[3:0], r2[4:0], b3[2], r3[4:0], b3[3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>m[4:0], r0[7:0], b3[1], b2[4], g0[7:0], b2[5], g2[4], b0[7:0], b3[5], b3[4], r1[4:0], g3[4], g2[3:0], g1[5:0], g3[3:0], b1[4:0], b3[1], b2[3:0], r2[4:0], b3[2], r3[4:0], b3[3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>m[4:0], r0[5:0], g3[4], b3[0], b3[1], b2[4], g0[5:0], g2[5], b2[5], b3[2], g2[4], b0[5:0], g3[5], b3[3], b3[5], b3[4], r1[5:0], g2[3:0], g1[5:0], g3[3:0], b1[5:0], b2[3:0], r2[5:0], r3[5:0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>m[4:0], r0[9:0], g0[9:0], b0[9:0], r1[9:0], g1[9:0], b1[9:0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>m[4:0], r0[9:0], g0[9:0], b0[9:0], r1[8:0], r0[10], g1[8:0], g0[10], b1[8:0], b0[10]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>m[4:0], r0[9:0], g0[9:0], b0[9:0], r1[7:0], r0[10:11], g1[7:0], g0[10:11], b1[7:0], b0[10:11]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>m[4:0], r0[9:0], g0[9:0], b0[9:0], r1[3:0], r0[10:15], g1[3:0], g0[10:15], b1[3:0], b0[10:15]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix D

Shared Objects and Multiple Contexts

This appendix describes special considerations for objects shared between multiple OpenGL contexts, including deletion behavior and how changes to shared objects are propagated between contexts.

Objects that can be shared between contexts include buffer objects, program and shader objects, renderbuffer objects, sync objects, and texture objects (except for the texture objects named zero).

Objects which contain references to other objects include framebuffer, program pipeline, query, transform feedback, and vertex array objects. Such objects are called container objects and are not shared.

Implementations may allow sharing between contexts implementing different OpenGL versions or different profiles of the same OpenGL version (see appendix E). However, implementation-dependent behavior may result when aspects and/or behaviors of such shared objects do not apply to, and/or are not described by more than one version or profile.

D.1 Object Deletion Behavior

D.1.1 Side Effects of Shared Context Destruction

The share list is the group of all contexts which share objects. If a shared object is not explicitly deleted, then destruction of any individual context has no effect on that object unless it is the only remaining context in the share list. Once the last context on the share list is destroyed, all shared objects, and all other resources allocated for that context or share list, will be deleted and reclaimed by the imple-
D.1. OBJECT DELETION BEHAVIOR

mentation as soon as possible.

D.1.2 Automatic Unbinding of Deleted Objects

When a buffer, texture, or renderbuffer object is deleted, it is unbound from any bind points it is bound to in the current context, as described for DeleteBuffers, DeleteTextures, and DeleteRenderbuffers. If the object binding was established with other related state (such as a buffer range in BindBufferRange or selected level and layer information in FramebufferTexture or BindImageTexture), that state is not affected by the automatic unbind. Bind points in other contexts are not affected. Attachments to unbound container objects, such as deletion of a buffer attach to a vertex array object which is not bound to the context, are not affected and continue to act as references on the deleted object, as described in the following section.

D.1.3 Deleted Object and Object Name Lifetimes

When a buffer, texture, renderbuffer, query, transform feedback, or sync object is deleted, its name immediately becomes invalid (e.g. is marked unused), but the underlying object will not be deleted until it is no longer in use. A buffer, texture, or renderbuffer object is in use while it is attached to any container object or bound to a context bind point in any context. A sync object is in use while there is a corresponding fence command which has not yet completed and signaled the sync object, or while there are any GL clients and/or servers blocked on the sync object as a result of ClientWaitSync or WaitSync commands. Query and transform feedback objects are in use so long as they are active, as described in sections 2.15 and 2.17.1, respectively.

When a shader object or program object is deleted, it is flagged for deletion, but its name remains valid until the underlying object can be deleted because it is no longer in use. A shader object is in use while it is attached to any program object. A program object is in use while it is the current program in any context.

Caution should be taken when deleting an object attached to a container object (such as a buffer object attached to a vertex array object, or a renderbuffer or texture attached to a framebuffer object), or a shared object bound in multiple contexts. Following its deletion, the object’s name may be returned by Gen* commands, even though the underlying object state and data may still be referred to by container objects, or in use by contexts other than the one in which the object was deleted. Such a container or other context may continue using the object, and may still contain state identifying its name as being currently bound, until such time as the container object is deleted, the attachment point of the container object is
changed to refer to another object, or another attempt to bind or attach the name is made in that context. Since the name is marked unused, binding the name will create a new object with the same name, and attaching the name will generate an error. The underlying storage backing a deleted object will not be reclaimed by the GL until all references to the object from container object attachment points or context binding points are removed.

D.2 Sync Objects and Multiple Contexts

When multiple GL clients and/or servers are blocked on a single sync object and that sync object is signalled, all such blocks are released. The order in which blocks are released is implementation-dependent.

D.3 Propagating Changes to Objects

GL objects contain two types of information, data and state. Collectively these are referred to below as the contents of an object. For the purposes of propagating changes to object contents as described below, data and state are treated consistently.

Data is information the GL implementation does not have to inspect, and does not have an operational effect. Currently, data consists of:

- Pixels in the framebuffer.
- The contents of textures and renderbuffers.
- The contents of buffer objects.

State determines the configuration of the rendering pipeline and the driver does have to inspect.

In hardware-accelerated GL implementations, state typically lives in GPU registers, while data typically lives in GPU memory.

When the contents of an object \( T \) are changed, such changes are not always immediately visible, and do not always immediately affect GL operations involving that object. Changes may occur via any of the following means:

- State-setting commands, such as TexParameter.
- Data-setting commands, such as TexSubImage* or BufferSubData.
• Data-setting through rendering to attached renderbuffers or transform feedback operations.

• Commands that affect both state and data, such as `TexImage*` and `BufferData`.

• Changes to mapped buffer data followed by a command such as `UnmapBuffer` or `FlushMappedBufferRange`.

• Rendering commands that trigger shader invocations, where the shader performs image stores, atomic operations, or built-in atomic counter functions.

### D.3.1 Determining Completion of Changes to an object

The contents of an object $T$ are considered to have been changed once a command such as described in section D.3 has completed. Completion of a command may be determined either by calling `Finish`, or by calling `FenceSync` and executing a `WaitSync` command on the associated sync object. The second method does not require a round trip to the GL server and may be more efficient, particularly when changes to $T$ in one context must be known to have completed before executing commands dependent on those changes in another context.

### D.3.2 Definitions

In the remainder of this section, the following terminology is used:

• An object $T$ is *directly attached* to the current context if it has been bound to one of the context binding points. Examples include but are not limited to bound textures, bound framebuffers, bound vertex arrays, and current programs.

• $T$ is *indirectly attached* to the current context if it is attached to another object $C$, referred to as a *container object*, and $C$ is itself directly or indirectly attached. Examples include but are not limited to renderbuffers or textures attached to framebuffers; buffers attached to vertex arrays; and shaders attached to programs.

---

1The GL already specifies that a single context processes commands in the order they are received. This means that a change to an object in a context at time $t$ must be completed by the time a command issued in the same context at time $t + 1$ uses the result of that change.
D.3. PROPAGATING CHANGES TO OBJECTS

- An object \( T \) which is directly attached to the current context may be re-attached by re-binding \( T \) at the same bind point. An object \( T \) which is indirectly attached to the current context may be re-attached by re-attaching the container object \( C \) to which \( T \) is attached.

  Corollary: re-binding \( C \) to the current context re-attaches \( C \) and its hierarchy of contained objects.

D.3.3 Rules

The following rules must be obeyed by all GL implementations:

Rule 1 If the contents of an object \( T \) are changed in the current context while \( T \) is directly or indirectly attached, then all operations on \( T \) will use the new contents in the current context.

  Note: The intent of this rule is to address changes in a single context only. The multi-context case is handled by the other rules.

  Note: “Updates” via rendering or transform feedback are treated consistently with update via GL commands. Once \texttt{EndTransformFeedback} has been issued, any command in the same context that uses the results of the transform feedback operation will see the results. If a feedback loop is setup between rendering and transform feedback (see above), results will be undefined.

Rule 2 While a container object \( C \) is bound, any changes made to the contents of \( C \)’s attachments in the current context are guaranteed to be seen. To guarantee seeing changes made in another context to objects attached to \( C \), such changes must be completed in that other context (see section D.3.1) prior to \( C \) being bound. Changes made in another context but not determined to have completed as described in section D.3.1, or after \( C \) is bound in the current context, are not guaranteed to be seen.

Rule 3 Changes to the contents of shared objects are not automatically propagated between contexts. If the contents of a shared object \( T \) are changed in a context other than the current context, and \( T \) is already directly or indirectly attached to the current context, any operations on the current context involving \( T \) via those attachments are not guaranteed to use its new contents.

Rule 4 If the contents of an object \( T \) are changed in a context other than the current context, \( T \) must be attached or re-attached to at least one binding point in the current context, or at least one attachment point of a currently bound container object \( C \), in order to guarantee that the new contents of \( T \) are visible in the current context.
Note: “Attached or re-attached” means either attaching an object to a binding point it wasn’t already attached to, or attaching an object again to a binding point it was already attached.

Note: To be sure that a data update resulting from a transform-feedback operation in another context is visible in the current context, the app needs to make sure that the command EndTransformFeedback has completed (see section D.3.1).

Example: If a texture image is bound to multiple texture bind points and the texture is changed in another context, re-binding the texture at any one of the texture bind points is sufficient to cause the changes to be visible at all texture bind points.
Appendix E

Profiles and the Deprecation Model

OpenGL 3.0 introduces a deprecation model in which certain features may be marked as deprecated. Deprecated features are expected to be completely removed from a future version of OpenGL. Deprecated features are summarized in section E.2.

To aid developers in writing applications which will run on such future versions, it is possible to create an OpenGL 3.0 context which does not support deprecated features. Such a context is called a forward compatible context, while a context supporting all OpenGL 3.0 features is called a full context. Forward compatible contexts cannot restore deprecated functionality through extensions, but they may support additional, non-deprecated functionality through extensions.

Profiles define subsets of OpenGL functionality targeted to specific application domains. OpenGL 3.2 defines two profiles (see below), and future versions may introduce additional profiles addressing embedded systems or other domains. OpenGL 3.2 implementations are not required to support all defined profiles, but must support the core profile described below.

To enable application control of deprecation and profiles, new context creation APIs have been defined as extensions to GLX and WGL. These APIs allow specifying a particular version, profile, and full or forward compatible status, and will either create a context compatible with the request, or fail (if, for example, requesting an OpenGL version or profile not supported by the implementation).

Only the ARB may define OpenGL profiles and deprecated features.
E.1 Core and Compatibility Profiles

OpenGL 3.2 is the first version of OpenGL to define multiple profiles. The core profile builds on OpenGL 3.1 by adding features described in section H.1. The compatibility profile builds on the combination of OpenGL 3.1 with the special GL_ARB_compatibility extension defined together with OpenGL 3.1, adding the same new features and in some cases extending their definition to interact with existing features of OpenGL 3.1 only found in GL_ARB_compatibility.

It is not possible to implement both core and compatibility profiles in a single GL context, since the core profile mandates functional restrictions not present in the compatibility profile. Refer to the WGL_ARB_create_context_profile and GLX_ARB_create_context_profile extensions (see appendix M.3.68) for information on creating a context implementing a specific profile.

E.2 Deprecated and Removed Features

OpenGL 3.0 defined a set of deprecated features. OpenGL 3.1 removed most of the deprecated features and moved them into the optional GL_ARB_compatibility extension. The OpenGL 3.2 core profile removes the same features as OpenGL 3.1, while the optional compatibility profile supports all those features.

Deprecated and removed features are summarized below in two groups: features which are marked deprecated by the core profile, but have not yet been removed, and features actually removed from the core profile of the current version of OpenGL (no features have been removed from or deprecated in the compatibility profile).

Functions which have been removed will generate an INVALID_OPERATION error if called in the core profile or in a forward-compatible context. Functions which are partially removed (e.g. no longer accept some parameter values) will generate the errors appropriate for any other unrecognized value of that parameter when a removed parameter value is passed in the core profile or a forward-compatible context. Functions which are deprecated but have not yet been removed from the core profile continue to operate normally except in a forward-compatible context, where they are also removed.

E.2.1 Deprecated But Still Supported Features

The following features are deprecated, but still present in the core profile. They may be removed from a future version of OpenGL, and are removed in a forward-compatible context implementing the core profile.
E.2. DEPRECATED AND REMOVED FEATURES

- Wide lines - **LineWidth** values greater than 1.0 will generate an **INVALID_VALUE** error.

- Global component limit query - the implementation-dependent values **MAX_VARYING_COMPONENTS** and **MAX_VARYING_FLOATS**.

E.2.2 Removed Features

- Application-generated object names - the names of all object types, such as buffer, query, and texture objects, must be generated using the corresponding **Gen*** commands. Trying to bind an object name not returned by a **Gen*** command will result in an **INVALID_OPERATION** error. This behavior is already the case for framebuffer, renderbuffer, and vertex array objects. Object types which have default objects (objects named zero), such as vertex array, framebuffer, and texture objects, may also bind the default object, even though it is not returned by **Gen***.

- Color index mode - No color index visuals are supplied by the window system-binding APIs such as GLX and WGL, so the default framebuffer is always in RGBA mode. All language and state related to color index mode vertex, rasterization, and fragment processing behavior is removed. **COLOR_INDEX** formats are also deprecated.

- OpenGL Shading Language versions 1.10 and 1.20. These versions of the shading language depend on many API features that have also been deprecated.

- **Begin / End** primitive specification - **Begin**, **End**, and **EdgeFlag*; Color*, **FogCoord*, **Index*, **Normal3*, **SecondaryColor3*, **TexCoord*, **Vertex*; and all associated state. Vertex arrays and array drawing commands must be used to draw primitives. However, **VertexAttrib* and the current vertex attribute state are retained in order to provide default attribute values for disabled attribute arrays.

E.2. DEPRECATED AND REMOVED FEATURES

and NORMALIZE; TexGen*, and Enable/Disable targets TEXTURE_-
GEN_*; Material*, Light*, LightModel*, and ColorMaterial, Shade-
Model, and Enable/Disable targets LIGHTING, VERTEX_PROGRAM_TWO_-
SIDE, LIGHT, and COLOR_MATERIAL; ClipPlane; and all associated
fixed-function vertex array, multitexture, matrix and matrix stack, normal
and texture coordinate, lighting, and clipping state. A vertex shader must be
defined in order to draw primitives.

Language referring to edge flags in the current specification is modified as
though all edge flags are TRUE.

Note that the FrontFace and ClampColor commands are not deprecated,
as they still affect other non-deprecated functionality; however, the Clamp-
Color targets CLAMP_VERTEX_COLOR and CLAMP_FRAGMENT_COLOR are
deprecated.

- Client vertex and index arrays - all vertex array attribute and element array
index pointers must refer to buffer objects. The default vertex array object
(the name zero) is also deprecated. Calling VertextAttribPointer when no
buffer object or no vertex array object is bound will generate an INVALID_-
OPERATION error, as will calling any array drawing command when no ver-
tex array object is bound.

- Rectangles - Rect*.

- Current raster position - RasterPos* and WindowPos*, and all associated
state.

- Two-sided color selection - Enable target VERTEX_PROGRAM_TWO_-
SIDE; OpenGL Shading Language built-ins gl_BackColor and gl_-
BackSecondaryColor; and all associated state.

- Non-sprite points - Enable/Disable targets POINT_SMOOTH and POINT_-
SPRITE, and all associated state. Point rasterization is always performed as
though POINT_SPRITE were enabled.

- Wide lines and line stipple - LineWidth is not deprecated, but values greater
than 1.0 will generate an INVALID_VALUE error; LineStipple and En-
able/Disable target LINE_STIPPLE, and all associated state.

- Quadrilateral and polygon primitives - vertex array drawing modes
POLYGON, QUADS, and QUAD_STRIP, related descriptions of rasterization
of non-triangle polygons, and all associated state.

OpenGL 4.2 (Core Profile) - April 27, 2012
E.2. DEPRECATED AND REMOVED FEATURES

- Separate polygon draw mode - **PolygonMode** face values of `FRONT` and `BACK`; polygons are always drawn in the same mode, no matter which face is being rasterized.

- Polygon Stipple - **PolygonStipple** and Enable/Disable target `POLYGON_ STIPPLE`, and all associated state.

- Pixel transfer modes and operations - all pixel transfer modes, including pixel maps, shift and bias, color table lookup, color matrix, and convolution commands and state, and all associated state and commands defining that state.

- Pixel drawing - **DrawPixels** and **PixelZoom**. However, the language describing pixel rectangles in section 3.7 is retained as it is required for **TexImage** and **ReadPixels**.

- Bitmaps - **Bitmap** and the BITMAP external format.

- Legacy OpenGL 1.0 pixel formats - the values 1, 2, 3, and 4 are no longer accepted as internal formats by **TexImage** or any other command taking an internal format argument. The initial internal format of a texel array is RGBA instead of 1. TEXTURE_COMPONENTS is deprecated; always use TEXTURE_INTERNAL_FORMAT.

- Legacy pixel formats - all **ALPHA**, **LUMINANCE**, **LUMINANCE_ALPHA**, and **INTENSITY** external and internal formats, including compressed, floating-point, and integer variants; all references to luminance and intensity formats elsewhere in the specification, including conversion to and from those formats; and all associated state. including state describing the allocation or format of luminance and intensity texture or framebuffer components.

- Depth texture mode - **DEPTH_TEXTURE_MODE**. Section 3.9.17 is to be changed so that \( r \) is returned to texture samplers directly, and the OpenGL Shading Language 1.30 Specification is to be changed so that \( (r,0,0,1) \) is always returned from depth texture samplers in this case.

- Texture wrap mode **CLAMP** - **CLAMP** is no longer accepted as a value of texture parameters **TEXTURE_WRAP_S**, **TEXTURE_WRAP_T**, or **TEXTURE_WRAP_R**.

- Texture borders - the `border` value to **TexImage** must always be zero, or an **INVALID_VALUE** error is generated (section 3.9.3); all language in section 3.9 referring to nonzero border widths during texture image specification and texture sampling; and all associated state.
E.2. DEPRECATED AND REMOVED FEATURES

- Automatic mipmap generation - TexParameter* target GENERATE_-MIPMAP, and all associated state.

- Fixed-function fragment processing - AreTexturesResident, PrioritizeTextures, and TexParameter target TEXTURE_PRIORITY; TexEnv target TEXTURE_ENV, and all associated parameters; TexEnv target TEXTURE_FILTER_CONTROL, and parameter name TEXTURE_LOD_BIAS; Enable targets of all dimensionalities (TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, and TEXTURE_CUBE_MAP); Enable target COLOR_SUM; Enable target FOG, Fog, and all associated parameters; the implementation-dependent values MAX_TEXTURE_UNITS and MAX_TEXTURE_COORDS; and all associated state.

- Alpha test - AlphaFunc and Enable/Disable target ALPHA_TEST, and all associated state.

- Accumulation buffers - ClearAccum, and ACCUM_BUFFER_BIT is not valid as a bit in the argument to Clear (section 4.2.3); Accum; the ACCUM_*_BITS framebuffer state describing the size of accumulation buffer components; and all associated state.

Window system-binding APIs such as GLX and WGL may choose to either not expose window configs containing accumulation buffers, or to ignore accumulation buffers when the default framebuffer bound to a GL context contains them.

- Pixel copying - CopyPixels (the comments also applying to CopyTexImage will be moved to section 3.9.4).

- Auxiliary color buffers, including AUXi targets of the default framebuffer.

- Context framebuffer size queries - RED_BITS, GREEN_BITS, BLUE_BITS, ALPHA_BITS, DEPTH_BITS, and STENCIL_BITS.

- Evaluators - Map*, EvalCoord*, MapGrid*, EvalMesh*, EvalPoint*, and all evaluator map enables, and all associated state.

- Selection and feedback modes - RenderMode, InitNames, PopName, PushName, LoadName, and SelectBuffer; FeedbackBuffer and PassThrough; and all associated state.

- Display lists - NewList, EndList, CallList, CallLists, ListBase, GenLists, IsList, and DeleteLists; all references to display lists and behavior when
compiling commands into display lists elsewhere in the specification; and all associated state.

- **Hints** - the `PERSPECTIVE_CORRECTION_HINT`, `POINT_SMOOTH_HINT`, `FOG_HINT`, and `GENERATE_MIPMAP_HINT` targets to `Hint` (section 5.4).

- **Attribute stacks** - `PushAttrib`, `PushClientAttrib`, `PopAttrib`, `PopClientAttrib`, the `MAX_ATTRIB_STACK_DEPTH`, `MAX_CLIENT_ATTRIB_STACK_DEPTH`, `ATTRIB_STACK_DEPTH`, and `CLIENT_ATTRIB_STACK_DEPTH` state, the client and server attribute stacks, and the values `ALL_ATTRIB_BITS` and `CLIENT_ALL_ATTRIB_BITS`.

- **Unified extension string** - `EXTENSIONS` target to `GetString`.

- **Token names and queries** - all token names and queries not otherwise mentioned above for deprecated state, as well as all query entry points where all valid targets of that query are deprecated state (chapter 6 and the state tables).
Appendix F

Version 3.0 and Before

OpenGL version 3.0, released on August 11, 2008, is the eighth revision since the original version 1.0. When using a full 3.0 context, OpenGL 3.0 is upward compatible with earlier versions, meaning that any program that runs with a 2.1 or earlier GL implementation will also run unchanged with a 3.0 GL implementation. OpenGL 3.0 context creation is done using a window system binding API, and on most platforms a new command, defined by extensions introduced along with OpenGL 3.0, must be called to create a 3.0 context. Calling the older context creation commands will return an OpenGL 2.1 context. When using a forward compatible context, many OpenGL 2.1 features are not supported.

Following are brief descriptions of changes and additions to OpenGL 3.0. Descriptions of changes and additions in earlier versions of OpenGL (versions 1.1, 1.2, 1.2.1, 1.3, 1.4, 1.5, 2.0, and 2.1) are omitted in this specification, but may be found in the OpenGL 3.0 Specification, available on the World Wide Web at URL http://www.opengl.org/registry/

F.1 New Features

New features in OpenGL 3.0, including the extension or extensions if any on which they were based, include:

- API support for the new texture lookup, texture format, and integer and unsigned integer capabilities of the OpenGL Shading Language 1.30 specification (GL_EXT_gpu_shader4).
- Conditional rendering (GL_NV_conditional_render).
F.2. DEPRECATION MODEL

- Fine control over mapping buffer subranges into client space and flushing modified data (GL_APPLE_flush_buffer_range).

- Floating-point color and depth internal formats for textures and renderbuffers (GL_ARB_color_buffer_float, GL_NV_depth_buffer_float, GL_ARB_texture_float, GL_EXT_packed_float, and GL_EXT_texture_shared_exponent).

- Framebuffer objects (GL_EXT_framebuffer_object).

- Half-float (16-bit) vertex array and pixel data formats (GL_NV_half_float and GL_ARB_half_float_pixel).

- Multisample stretch blit functionality (GL_EXT_framebuffer_multisample and GL_EXT_framebuffer_blit).

- Non-normalized integer color internal formats for textures and renderbuffers (GL_EXT_texture_integer).

- One- and two-dimensional layered texture targets (GL_EXT_texture_array).

- Packed depth/stencil internal formats for combined depth+stencil textures and renderbuffers (GL_EXT_packed_depthStencil).

- Per-color-attachment blend enables and color writemasks (GL_EXT_draw_buffers2).

- RGTC specific internal compressed formats (GL_EXT_texture_compression_rgtc).

- Single- and double-channel (R and RG) internal formats for textures and renderbuffers.

- Transform feedback (GL_EXT_transform_feedback).

- Vertex array objects (GL_APPLE_vertex_array_object).

- sRGB framebuffer mode (GL_EXT_framebuffer_sRGB)

F.2 Deprecation Model

OpenGL 3.0 introduces a deprecation model in which certain features may be marked as deprecated. The deprecation model is described in detail in appendix E, together with a summary of features deprecated in OpenGL 3.0.
F.3  CHANGED TOKENS

<table>
<thead>
<tr>
<th>New Token Name</th>
<th>Old Token Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPARE_REF_TO_TEXTURE</td>
<td>COMPARE_R_TO_TEXTURE</td>
</tr>
<tr>
<td>MAX_VARYING_COMPONENTS</td>
<td>MAX_VARYING_FLOATS</td>
</tr>
<tr>
<td>MAX_CLIP_DISTANCES</td>
<td>MAX_CLIP_PLANES</td>
</tr>
<tr>
<td>CLIP_DISTANCE&lt;i&gt;</td>
<td>CLIP_PLANE&lt;i&gt;</td>
</tr>
</tbody>
</table>

Table F.1: New token names and the old names they replace.

F.3 Changed Tokens

New token names are introduced to be used in place of old, inconsistent names. However, the old token names continue to be supported, for backwards compatibility with code written for previous versions of OpenGL. The new names, and the old names they replace, are shown in table F.1.

F.4 Change Log

Minor corrections to the OpenGL 3.0 Specification were made after its initial release.

Changes in the draft of September 23, 2008:

• Changed ClearBuffer* in section 4.2.3 to use DEPTH and STENCIL buffer names. Changed GetFramebufferAttachmentParameteriv in section 6.1.13 to accept only DEPTH and STENCIL to identify default framebuffer depth and stencil buffers, and only DEPTH_ATTACHMENT and STENCIL_ATTACHMENT to identify framebuffer object depth and stencil buffers (bug 3744).

Changes in the draft of September 18, 2008:

• Added missing close-brace to ArrayElement pseudocode in section 2.8 (bug 3897).

• Noted in section 2.15 that BeginQuery will generate an INVALID_OPERATION error when called with an existing query object name whose type does not match the specified target (bug 3712).

• Add description of gl_ClipDistance to shader outputs in section 2.11.12 and note that only one of gl_ClipVertex and gl_ClipDistance should be written by a shader (bug 3898).

OpenGL 4.2 (Core Profile) - April 27, 2012
• Changed `ClearBuffer*` in section 4.2.3 to indirect through the draw buffer state by specifying the buffer type and draw buffer number, rather than the attachment name; also changed to accept `DEPTH_BUFFER / DEPTH_ATTACHMENT` and `STENCIL_BUFFER / STENCIL_ATTACHMENT` interchangeably, to reduce inconsistency between clearing the default framebuffer and framebuffer objects. Likewise changed `GetFramebufferAttachmentParameteriv` in section 6.1.13 to accept `DEPTH_BUFFER / DEPTH_ATTACHMENT` and `STENCIL_BUFFER / STENCIL_ATTACHMENT` interchangeably (bug 3744).

• Add proper type suffix to query commands in tables 6.5 and 6.40 (Mark Kilgard).

• Update deprecation list in section E.2 to itemize deprecated state for two-sided color selection and include per-texture-unit LOD bias (bug 3735).

Changes in the draft of August 28, 2008:

• Sections 2.9, 2.9.3; tables 2.9, 2.10, and 6.8 - move buffer map/unmap calls into their own subsection and rewrite `MapBuffer` in terms of `MapBufferRange`. Add buffer state `BUFFER_ACCESS_FLAGS`, `BUFFER_MAP_OFFSET`, `BUFFER_MAP_LENGTH`. Make `MapBuffer` and `MapBufferRange` errors consistent (bug 3601).

• Section 2.10 - Extend `INVALID_OPERATION` error to any array pointer-setting command called to specify a client array while a vertex array object is bound, not just `VertexAttrib*Pointer` (bug 3696).

• Sections 2.14.1, 4.1.2, 4.2.1, and 4.3.3 - define initial state when a context is bound with no default framebuffer - null viewport and scissor region, draw buffer = read buffer = `NONE`, max viewport dims = `max(display size - if any, max renderbuffer size)`. Viewport/scissor language added to the GLX and WGL create context extension specs as well (bug 2941).

• Section 2.17 - define “word-aligned” to be a multiple of 4 (e.g. 32 bits) (bug 3624).

• Section 6.1.9 - Moved `GetBufferParameteriv` query from section 6.1.3 and changed formal argument specifying the parameter name from `value` to `pname` (side effect of bug 3697).

• Section 6.1.13 - Moved `GetFramebufferAttachmentiv` query from section 6.1.3. Querying framebuffer attachment parameters other than object

OpenGL 4.2 (Core Profile) - April 27, 2012
CREDITS AND ACKNOWLEDGEMENTS

OpenGL 3.0 is the result of the contributions of many people and companies. Members of the Khronos OpenGL ARB Working Group during the development of OpenGL 3.0, including the company that they represented at the time of their contributions, follow. Some major contributions made by individuals are listed together with their name, including specific functionality developed in the form of new ARB extensions together with OpenGL 3.0. In addition, many people participated in developing earlier vendor and EXT extensions on which the OpenGL 3.0 functionality is based in part; those individuals are listed in the respective extension specifications in the OpenGL Extension Registry.

Aaftab Munshi, Apple
Alain Bouchard, Matrox
Alexis Mather, AMD (Chair, ARB Marketing TSG)
Andreas Wolf, AMD
Avi Shapira, Graphic Remedy

OpenGL 4.2 (Core Profile) - April 27, 2012
Barthold Lichtenbelt, NVIDIA (Chair, Khronos OpenGL ARB Working Group)
Benjamin Lipchak, AMD
Benji Bowman, Imagination Technologies
Bill Licea-Kane, AMD (Chair, ARB Shading Language TSG)
Bob Beretta, Apple
Brent Insko, Intel
Brian Paul, Tungsten Graphics
Bruce Merry, ARM (Detailed specification review)
Cass Everitt, NVIDIA
Chris Dodd, NVIDIA
Daniel Horowitz, NVIDIA
Daniel Koch, TransGaming (Framebuffer objects, half float vertex formats, and
  instanced rendering)
Daniel Omachi, Apple
Dave Shreiner, ARM
Eric Boumaour, AMD
Eskil Steenberg, Obsession
Evan Hart, NVIDIA
Folker Schamel, Spinor GMBH
Gavriel State, TransGaming
Geoff Stahl, Apple
Georg Kolling, Imagination Technologies
Gregory Prisament, NVIDIA
Guillaume Portier, HI Corp
Ian Romanick, IBM / Intel (Vertex array objects; GLX protocol)
James Helferty, TransGaming (Instanced rendering)
James Jones, NVIDIA
Jamie Gennis, NVIDIA
Jason Green, TransGaming
Jeff Bolz, NVIDIA
Jeff Juliano, NVIDIA
Jeremy Sandmel, Apple (Chair, ARB Nextgen (OpenGL 3.0) TSG)
John Kessenich, Intel (OpenGL Shading Language Specification Editor; depre-
  cation model)
John Rosasco, Apple
Jon Leech, Independent (Chair, ARB Ecosystem TSG; OpenGL API Specifi-
  cation Editor; R/RG image formats and new context creation APIs)
Marc Olano, U. Maryland
Mark Callow, HI Corp
Mark Kilgard, NVIDIA (Many extensions on which OpenGL 3.0 features were

OpenGL 4.2 (Core Profile) - April 27, 2012
Matti Paavola, Nokia
Michael Gold, NVIDIA (Framebuffer objects and instanced rendering)
Neil Trevett, NVIDIA (President, Khronos Group)
Nick Burns, Apple
Nick Haemel, AMD
Pat Brown, NVIDIA (Many extensions on which OpenGL 3.0 features were based; detailed specification review)
Paul Martz, SimAuthor
Paul Ramsey, Sun
Pierre Boudier, AMD (Floating-point depth buffers)
Rob Barris, Blizzard (Framebuffer object and map buffer range)
Robert Palmer, Symbian
Robert Simpson, AMD
Steve Demlow, Vital Images
Thomas Roell, NVIDIA
Timo Suoranta, Futuremark
Tom Longo, AMD
Tom Olson, TI (Chair, Khronos OpenGL ES Working Group)
Travis Bryson, Sun
Yaki Tebeka, Graphic Remedy
Yanjun Zhang, S3 Graphics
Zack Rusin, Tungsten Graphics

The ARB gratefully acknowledges administrative support by the members of Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, and Michelle Clark, and technical support from James Riordon, webmaster of Khronos.org and OpenGL.org.
Appendix G

Version 3.1

OpenGL version 3.1, released on March 24, 2009, is the ninth revision since the original version 1.0.

Unlike earlier versions of OpenGL, OpenGL 3.1 is not upward compatible with earlier versions. The commands and interfaces identified as deprecated in OpenGL 3.0 (see appendix F) have been removed from OpenGL 3.1 entirely, with the following exception:

- Wide lines have not been removed, and calling `LineWidth` with values greater than 1.0 is not an error.

Implementations may restore such removed features using the GL_ARB_compatibility extension discussed in section G.2.

Following are brief descriptions of changes and additions to OpenGL 3.1.

G.1 New Features

New features in OpenGL 3.1, including the extension or extensions if any on which they were based, include:

- Support for OpenGL Shading Language 1.30 and 1.40.
- Instanced rendering with a per-instance counter accessible to vertex shaders (GL_ARB_draw_instanced).
- Data copying between buffer objects (GL_ARB_copy_buffer).
- Primitive restart (GL_NV_primitive_restart). Because client enable/disable no longer exists in OpenGL 3.1, the PRIMITIVE_RESTART
state has become server state, unlike the NV extension where it is client state. As a result, the numeric values assigned to PRIMITIVE_RESTART and PRIMITIVE_RESTART_INDEX differ from the NV versions of those tokens.

- At least 16 texture image units must be accessible to vertex shaders, in addition to the 16 already guaranteed to be accessible to fragment shaders.

- Texture buffer objects (GL_ARB_texture_buffer_object).

- Rectangular textures (GL_ARB_texture_rectangle).

- Uniform buffer objects (GL_ARB_uniform_buffer_object).

- Signed normalized texture component formats.

G.2 Deprecation Model

The features marked as deprecated in OpenGL 3.0 (see section E) have been removed from OpenGL 3.1 (with the exception of line widths greater than one, which are retained).

As described by the deprecation model, features removed from OpenGL 3.0 have been moved into the new extension GL_ARB_compatibility. If an implementation chooses to provide this extension, it restores all features deprecated by OpenGL 3.0 and removed from OpenGL 3.1. This extension may only be provided in an OpenGL 3.1 or later context version.

Because of the complexity of describing this extension relative to the OpenGL 3.1 core specification, it is not written up as a separate document, unlike other extensions in the extension registry. Instead, an alternate version of this specification document has been generated with the deprecated material still present, but marked in a distinct color.

No additional features are deprecated in OpenGL 3.1.

G.3 Change Log

Changes in the specification update of May 28, 2009:

- Update MAX_CLIP_DISTANCES from 6 to 8 in section 2.20 and table 6.47, to match GLSL (bug 4803).

- Accept NULL pointers in CompressedTexImage* (section 3.9.5) and treat them the same as for the corresponding TexImage* commands (bug 4863).
Relax error conditions when specifying RGTC format texture images (section 3.9.4) and subimages (section 3.9.5) so that non-power-of-two RGTC images may be specified (also see section C.1), and edits to partial tiles at the edge of such an image made (bug 4856).

Relaxed texture magnification switch-over point calculation in section 3.9.12 (bug 4392).

Clarify initial value of stencil value masks in section 4.1.4 and table 6.20 (bug 4378).

Change FramebufferTextureLayer in section 4.4.2 to generate INVALID_VALUE for negative layer only if texture is non-zero (bug 4084).

Clarify RenderbufferStorageMultisample language in section 4.4.2 to allow, but not require creation of multisampled integer renderbuffers with more one sample (bug 4396).

Added language to section 6.1.4 disallowing data-type format mismatches between internal and external texture formats in GetTexImage (bug 4163).

Change initial value of FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE in table 6.25 to NONE (bug 4407).

Brought extension list in appendix M.3 up to date and correctly described extensions introduced along with OpenGL 3.0 and OpenGL 3.1 which implement subsets of new functionality in those versions to enable older hardware.

Added missing contributors to the OpenGL 3.1 contributor list.

G.4 Credits and Acknowledgements

OpenGL 3.1 is the result of the contributions of many people and companies. Members of the Khronos OpenGL ARB Working Group during the development of OpenGL 3.1, including the company that they represented at the time of their contributions, follow. Some major contributions made by individuals are listed together with their name, including specific functionality developed in the form of new ARB extensions together with OpenGL 3.1. In addition, many people participated in developing earlier vendor and EXT extensions on which the OpenGL 3.1 functionality is based in part; those individuals are listed in the respective extension specifications in the OpenGL Extension Registry.

OpenGL 4.2 (Core Profile) - April 27, 2012
Alexis Mather, AMD (Chair, ARB Marketing TSG)
Avi Shapira, Graphic Remedy
Barthold Lichtenbelt, NVIDIA (Chair, Khronos OpenGL ARB Working Group)
Benjamin Lipchak, Apple (Uniform buffer objects)
Bill Licea-Kane, AMD (Chair, ARB Shading Language TSG; signed normalized texture formats)
Brent Insko, Intel
Brian Paul, Tungsten Graphics
Bruce Merry, ARM (Detailed specification review)
Christopher Webb, NVIDIA
Daniel Koch, TransGaming
Daniel Omachi, Apple
Eric Werness, NVIDIA
Gavriel State, TransGaming
Geoff Stahl, Apple
Gregory Roth, NVIDIA
Ian Romanick, Intel
James Helferty, TransGaming
James Jones, NVIDIA
Jeff Bolz, NVIDIA (Buffer to buffer copies)
Jeremy Sandmel, Apple (Chair, ARB Nextgen (OpenGL 3.1) TSG; uniform buffer objects)
John Kessenich, Intel (OpenGL Shading Language Specification Editor)
John Rosasco, Apple (Uniform buffer objects)
Jon Leech, Independent (OpenGL API Specification Editor)
Mark Callow, HI Corp
Mark Kilgard, NVIDIA (Many extensions on which OpenGL 3.0 features were based)
Matt Craighead, NVIDIA
Michael Gold, NVIDIA
Neil Trevett, NVIDIA (President, Khronos Group)
Nick Haemel, AMD
Pat Brown, NVIDIA (Many extensions on which OpenGL 3.0 features were based; detailed specification review)
Paul Martz, SimAuthor
Pierre Boudier, AMD
Rob Barris, Blizzard
Tom Olson, TI (Chair, Khronos OpenGL ES Working Group)
Yaki Tebeka, Graphic Remedy
Yanjun Zhang, S3 Graphics

OpenGL 4.2 (Core Profile) - April 27, 2012
The ARB gratefully acknowledges administrative support by the members of Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, and Michelle Clark, and technical support from James Riordon, webmaster of Khronos.org and OpenGL.org.
Appendix H

Version 3.2

OpenGL version 3.2, released on August 3, 2009, is the tenth revision since the original version 1.0.

Separate versions of the OpenGL 3.2 Specification exist for the core and compatibility profiles described in appendix E, respectively subtitled the “Core Profile” and the “Compatibility Profile”. This document describes the Core Profile. An OpenGL 3.2 implementation must be able to create a context supporting the core profile, and may also be able to create a context supporting the compatibility profile.

Material specific to the compatibility profile specification is marked in a distinct color to clearly call out differences between the two profiles.

The OpenGL 3.2 core profile is upward compatible with OpenGL 3.1, but not with earlier versions (see appendices G and F).

The OpenGL 3.2 compatibility profile is upward compatible with the combination of OpenGL 3.1 and the GL_ARB_compatibility extension, as well as with all earlier versions of OpenGL.

Following are brief descriptions of changes and additions to OpenGL 3.2.

H.1 New Features

New features in OpenGL 3.2, including the extension or extensions if any on which they were based, include:

- Introduction of core and compatibility profiles, superseding the GL_ARB_compatibility extension introduced with OpenGL 3.1.

- Support for OpenGL Shading Language 1.50.
H.2. DEPRECIATION MODEL

- **BGRA vertex component ordering** (GL_ARB_vertex_array_bgra).
- Drawing commands allowing modification of the base vertex index (GL_ARB_draw_elements_base_vertex).
- Shader fragment coordinate convention control (GL_ARB_fragment_coord_conventions).
- Provoking vertex control (GL_ARB_provoking_vertex).
- Seamless cube map filtering (GL_ARB_seamless_cube_map).
- Multisampled textures and texture samplers for specific sample locations (GL_ARB_texture_multisample).
- Fragment depth clamping (GL_ARB_depth_clamp).
- Geometry shaders (GL_ARB_geometry_shader4).
- Fence sync objects (GL_ARB_sync).

H.2 Deprecation Model

In addition to restoring features removed from OpenGL 3.1 core, the compatibility profile defines some additional interactions between those features and new features introduced by OpenGL 3.2.

The following features are newly deprecated by the OpenGL 3.2 core profile:

- Global component limit query - the implementation-dependent values MAX_VARYING_COMPONENTS and MAX_VARYING_FLOATS.

No features are deprecated by the OpenGL 3.2 compatibility profile.

H.3 Changed Tokens

New token names are introduced to be used in place of old, less general names. However, the old token names continue to be supported, for backwards compatibility with code written for previous versions of OpenGL. The new names, and the old names they replace, are shown in table H.1.
H.4 Change Log

Minor corrections to the OpenGL 3.2 Specification were made after its initial release in the update of December 7, 2009:

- Clean up description of GL command syntax in section 2.3, generalize the list of object types in the introduction to section 2.5 instead of enumerating them all redundantly, fix `BindBuffer` (section 2.9.1), `BeginQuery` (section 2.15), and `BindTexture` (section 3.9.1) to only generate errors for user generated names in the core profile, increase minimum number of clip half-spaces to 8 in section 2.20, improve formatting and correct column heading in tables 6.14 and 6.15, and a variety of minor editorial corrections not enumerated here (Bug 5761).

- Remove “just as if they were unused” from description of names generated by `GenVertexArrays`, `GenFramebuffers`, and `GenRenderbuffers` in sections 2.10, 4.4.1, and 4.4.2 (Bug 5201).

- Moved error language resulting from trying to put client pointers into VAOs from section 2.10 to section 2.8 (Bug 3975). This results in a forward reference, unfortunately.

- Remove reference to borders from texel fetch language in section 2.11.12 of the core specification (Bug 5343).

- Remove `INVALID_VALUE` error when passing a program object to `TransformFeedbackVaryings` in section 2.11.11 (Bug 5661).

- Changed number of query types from two to three in section 2.15 (Bug 5624).

- Change flat-shading source value description from “generic attribute” to “varying” in sections 3.5.1 and 3.6.1 (Bug 5359).

- Remove leftover references in core spec sections 3.9.5 and 6.1.3 to deprecated texture border state (Bug 5579). Still need to fix gl3.h accordingly.

OpenGL 4.2 (Core Profile) - April 27, 2012
• Fix typo in second paragraph of section 3.9.8 (Bug 5625).

• Simplify and clean up equations in the coordinate wrapping and mipmapping calculations of section 3.9.11, especially in the core profile where wrap mode CLAMP does not exist (Bug 5615).

• Fix computation of $u(x, y)$ and $v(x, y)$ in scale factor calculations of section 3.9.11 for rectangular textures (Bug 5700).

• Restructure definition of texture completeness in section 3.9.14 to separate mipmap consistency from filter requirements and cover new texture target types, and simplify how completeness applies to texture fetches (section 2.11.12) and lookups (sections 2.11.12 and 3.10.2) (Bugs 4264, 5749).

• Update sampling language in sections 3.9.14, 2.11.12, and 3.10.2 to not require texture completeness when non-mipmapped access to the texture is being done (Bug 4264, based on ES bugs 4282 and 3499).

• Add fixed sample location state for multisample textures to section 3.9.15 (Bug 5454).

• Don’t use the sign of the input component in the description of dithering in section 4.1.9 (Bug 5594).

• Change error condition for certain invalid buffers to DrawBuffers in section 4.2.1 from INVALID_OPERATION to INVALID_ENUM (Bug 5576).

• Clarify error conditions in section 4.2.3 when the clear mask is zero or contains invalid bits (Bug 5567).

• Change BlitFramebuffer in section 4.3.2 so format conversion is supported only within the three equivalence classes of fixed-point and floating-point buffers, unsigned integer buffers, and signed integer buffers (Bug 5577).

• Remove a reference to unreachable INVALID_OPERATION errors from the core profile only in section 6.1.2 (Bug 5365).

• Specify that compressed texture component type queries in section 6.1.3 return how components are interpreted after decompression (Bug 5453).

• Increase value of MAX_UNIFORM_BUFFER_BINDINGS and MAX_COMBINED_UNIFORM_BLOCKS in table 6.55 from 24 to 36 to reflect addition of geometry shaders (Bug 5607).

• Update sharing rule 2 in appendix D.3.3 to read sensibly (Bug 5397).

OpenGL 4.2 (Core Profile) - April 27, 2012
H.5. CREDITS AND ACKNOWLEDGEMENTS

- Update sharing rule 4 in appendix D.3.3 to cover the case where an object is only attached or bound in a single place, clarify comments about transform feedback, and state that reattachment is required to guarantee seeing changes made in other contexts, but does not preclude implementations from making changes visible without reattachment (Bugs 5546, 5777).

Changes in the specification for public release on August 3, 2009:

- Public release of OpenGL 3.2.

H.5 Credits and Acknowledgements

OpenGL 3.2 is the result of the contributions of many people and companies. Members of the Khronos OpenGL ARB Working Group during the development of OpenGL 3.2, including the company that they represented at the time of their contributions, follow. Some major contributions made by individuals are listed together with their name, including specific functionality developed in the form of new ARB extensions together with OpenGL 3.1. In addition, many people participated in developing earlier vendor and EXT extensions on which the OpenGL 3.1 functionality is based in part; those individuals are listed in the respective extension specifications in the OpenGL Extension Registry.

Aaftab Munshi, Apple
Acorn Pooley, NVIDIA
Barthold Lichtenbelt, NVIDIA (Chair, Khronos OpenGL ARB Working Group)
Benjamin Lipchak, Apple
Bill Licea-Kane, AMD (Chair, ARB Shading Language TSG)
Bruce Merry, ARM (Detailed specification review)
Cynthia Allison, NVIDIA
Daniel Koch, TransGaming (base vertex offset drawing, fragment coordinate conventions, provoking vertex control, BGRA attribute component ordering)
Dave Shreiner, ARM
David Garcia, AMD
Gavriel State, TransGaming
Geoff Stahl, Apple
Graham Sellers, AMD (seamless cube maps)
Gregory Roth, NVIDIA
Henri Verbeet, CodeWeavers
Ian Romanick, Intel
Jason Green, TransGaming

OpenGL 4.2 (Core Profile) - April 27, 2012
Jeff Bolz, NVIDIA (multisample textures)
Jeff Juliano, NVIDIA
Jeremy Sandmel, Apple (Chair, ARB Nextgen (OpenGL 3.2) TSG)
John Kessenich, Intel (OpenGL Shading Language Specification Editor)
Jon Leech, Independent (OpenGL API Specification Editor, fence sync objects)
Marcus Steyer, NVIDIA
Mark Callow, HI Corp
Mark Kilgard, NVIDIA (Many extensions on which OpenGL 3.2 features were based, including depth clamp, fragment coordinate conventions, provoking vertex control, and BGRA attribute component ordering)
Mark Krenek, Aspyr
Michael Gold, NVIDIA
Neil Trevett, NVIDIA (President, Khronos Group)
Nicholas Vining, Destineer
Nick Haemel, AMD
Pat Brown, NVIDIA (Many extensions on which OpenGL 3.0 features were based; detailed specification review)
Patrick Doane, Blizzard
Paul Martz, Skew Matrix
Pierre Boudier, AMD
Rob Barris, Blizzard
Ryan Gordon, Destineer
Stefan Dosinger, CodeWeavers
Yanjun Zhang, S3 Graphics

The ARB gratefully acknowledges administrative support by the members of Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, and Michelle Clark, and technical support from James Riordon, webmaster of Khronos.org and OpenGL.org.
Appendix I

Version 3.3

OpenGL version 3.3, released on March 11, 2010 is the eleventh revision since the original version 1.0.

Separate versions of the OpenGL 3.3 Specification exist for the core and compatibility profiles described in appendix E, respectively subtitled the “Core Profile” and the “Compatibility Profile”. This document describes the Core Profile. An OpenGL 3.3 implementation must be able to create a context supporting the core profile, and may also be able to create a context supporting the compatibility profile.

Material specific to the compatibility profile specification is marked in a distinct color to clearly call out differences between the two profiles.

The OpenGL 3.3 compatibility and core profiles are upward compatible with the OpenGL 3.2 compatibility and core profiles, respectively (see appendix H).

Following are brief descriptions of changes and additions to OpenGL 3.3.

I.1 New Features

New features in OpenGL 3.3, including the extension or extensions if any on which they were based, include:

- Support for OpenGL Shading Language 3.30, including built-in functions for getting and setting the bit encoding for floating-point values (GL_ARB_shader_bit_encoding - this extension only affects the shading language, not the API).

- New blending functions whereby a fragment shader may output two colors, one of which is treated as the source color, and the other used as a blend-
I.2. DEPRECATION MODEL

No new features are deprecated by OpenGL 3.3. Features deprecated by OpenGL 3.2 remain deprecated, but have not yet been removed.
I.3 Change Log

I.4 Credits and Acknowledgements

OpenGL 3.3 is the result of the contributions of many people and companies. Members of the Khronos OpenGL ARB Working Group during the development of OpenGL 3.3, including the company that they represented at the time of their contributions, follow. Some major contributions made by individuals are listed together with their name, including specific functionality developed in the form of new ARB extensions together with OpenGL 3.2. In addition, many people participated in developing earlier vendor and EXT extensions on which the OpenGL 3.3 functionality is based in part; those individuals are listed in the respective extension specifications in the OpenGL Extension Registry.

Aaftab Munshi, Apple
Alex Chalfin, AMD
Aske Simon Christensen, ARM
Axel Mamode, Sony
Barthold Lichtenbelt, NVIDIA (Chair, Khronos OpenGL ARB Working Group)
Benj Lipchak, AMD
Benjamin Lipchak, Apple
Bill Licea-Kane, AMD (GL_ARB_occlusion_query2, GL_ARB_shader_bit_encoding)
Brian Harris, Id Software
Brian Paul, VMware
Bruce Merry, ARM (Detailed specification review)
Cass Everitt, Epic Games
Chris Dodd, NVIDIA
Daniel Koch, TransGaming Inc. (GL_ARB_instanced_arrays, GL_ARB_texture_rgb10_a2ui)
Dave Shreiner, ARM
Eric Boumaour, AMD
Eric Werness, NVIDIA
Eric Zolnowski, AMD
Evan Hart, AMD
Graham Sellers, AMD (GL_ARB_blend_func_extended, GL_ARB_sampler_objects, GL_ARB_vertex_type_2_10_10_10_rev)
Gregory Roth, NVIDIA
Ian Romanick, Intel (GL_ARB_explicit_attrib_location)
Ian Stewart, NVIDIA

OpenGL 4.2 (Core Profile) - April 27, 2012
I.4. CREDITS AND ACKNOWLEDGEMENTS

Ignacio Castano, NVIDIA
Jaakko Konttinen, AMD
James Helferty, TransGaming Inc. (GL_ARB_instanced_arrays)
James Jones, NVIDIA Corporation
Jason Green, TransGaming Inc.
Jeff Bolz, NVIDIA (GL_ARB_texture_swizzle)
Jeremy Sandmel, Apple (Chair, ARB Nextgen (OpenGL 4.0) TSG)
John Kessenich, Intel (OpenGL Shading Language Specification Editor)
John Rosasco, Apple
Jon Leech, Independent (OpenGL API Specification Editor)
Lijun Qu, AMD
Mais Alnasser, AMD
Mark Callow, HI Corp
Mark Young, AMD
Maurice Ribble, Qualcomm
Michael Gold, NVIDIA
Mike Strauss, NVIDIA
Mike Weiblen, Zebra Imaging
Murat Balci, AMD
Neil Trevett, NVIDIA (President, Khronos Group)
Nick Haemel, AMD (Pat Brown, NVIDIA
Patrick Doane, Blizzard
Pierre Boudier, AMD
Piers Daniell, NVIDIA (GL_ARB_timer_query)
Piotr Uminski, Intel
Remi Arnaud, Sony
Rob Barris
Robert Simpson, Qualcomm
Timothy Lamb, AMD
Tom Olson, ARM
Tom Olson, TI (Chair, Khronos OpenGL ES Working Group)
Yanjun Zhang, S3 Graphics
Yunjun Zhang, AMD

The ARB gratefully acknowledges administrative support by the members of
Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, and Michelle Clark, and technical support from James Riordon, webmaster of Khronos.org and OpenGL.org.

OpenGL 4.2 (Core Profile) - April 27, 2012
Appendix J

Version 4.0

OpenGL version 4.0, released on March 11, 2010, is the twelfth revision since the original version 1.0.

Separate versions of the OpenGL 4.0 Specification exist for the core and compatibility profiles described in appendix E, respectively subtitled the “Core Profile” and the “Compatibility Profile”. This document describes the Core Profile. An OpenGL 4.0 implementation must be able to create a context supporting the core profile, and may also be able to create a context supporting the compatibility profile.

Material specific to the compatibility profile specification is marked in a distinct color to clearly call out differences between the two profiles.

The OpenGL 4.0 compatibility and core profiles are upward compatible with the OpenGL 3.3 compatibility and core profiles, respectively (see appendix I).

Following are brief descriptions of changes and additions to OpenGL 4.0.

J.1 New Features

New features in OpenGL 4.0, including the extension or extensions if any on which they were based, include:

- Support for OpenGL Shading Language 4.00, including new fragment shader texture functions (textureLOD) that return the results of automatic level-of-detail computations that would be performed if a texture lookup were performed (GL_ARB_texture_query_lod - this extension only affects the shading language, not the API).

- Ability to set individual blend equations and blend functions for each color output (GL_ARB_draw_buffers_blend).
J.1. NEW FEATURES

• Mechanism for supplying the arguments to a DrawArraysInstanced or DrawElementsInstancedBaseVertex drawing command from buffer object memory (GL_ARB_draw_indirect).

• Many new features in OpenGL Shading Language 4.00 and related APIs to support capabilities of current generation GPUs (GL_ARB_gpu_shader5 - see that extension specification for a detailed summary of the features).

• Support for double-precision floating-point uniforms, including vectors and matrices, as well as double-precision floating-point data types in shaders (GL_ARB_gpu_shader_fp64).

• Ability to explicitly request that an implementation use a minimum number of unique set of fragment computation inputs when multisampling a pixel (GL_ARB_sample_shading).

• Support for “indirect subroutine calls”, where a single shader can include many subroutines and dynamically select through the API which subroutine is called from each call site (GL_ARB_shader_subroutine).

• New tessellation stages and two new corresponding shader types, tessellation control and tessellation evaluation shaders, operating on patches (fixed-sized collections of vertices) (GL_ARB_tessellation_shader).

• Support for three-component buffer texture formats RGB32F, RGB32I, and RGB32UI (GL_ARB_texture_buffer_object_rgb32).

• Cube map array textures, 2-dimensional array textures that may contain many cube map layers. Each cube map layer is a unique cube map image set (GL_ARB_texture_cube_map_array).

• New texture functions (textureGather) that determine the 2x2 footprint used for linear filtering in a texture lookup, and return a vector consisting of the first component from each of the four texels in the footprint (GL_ARB_texture_gather).

• Additional transform feedback functionality including
  
  – transform feedback objects which encapsulate transform feedback-related state;
  
  – the ability to pause and resume transform feedback operations; and
  
  – the ability to draw primitives captured in transform feedback mode without querying the captured primitive count.

OpenGL 4.2 (Core Profile) - April 27, 2012
J.2 DEPRECATION MODEL

(GL_ARB_transform_feedback2).

- Additional transform feedback functionality including increased flexibility in how vertex attributes can be written to buffer objects and new support for multiple separate vertex streams (GL_ARB_transform_feedback3).

J.2 Deprecation Model

No new features are deprecated by OpenGL 4.0. Features deprecated by OpenGL 3.3 remain deprecated, but have not yet been removed.

J.3 Change Log

J.4 Credits and Acknowledgements

OpenGL 4.0 is the result of the contributions of many people and companies. Members of the Khronos OpenGL ARB Working Group during the development of OpenGL 4.0, including the company that they represented at the time of their contributions, follow. Some major contributions made by individuals are listed together with their name, including specific functionality developed in the form of new ARB extensions together with OpenGL 3.2. In addition, many people participated in developing earlier vendor and EXT extensions on which the OpenGL 4.0 functionality is based in part; those individuals are listed in the respective extension specifications in the OpenGL Extension Registry.

Aaftab Munshi, Apple
Alex Chalfin, AMD
Aske Simon Christensen, ARM
Axel Mamode, Sony
Barthold Lichtenbelt, NVIDIA (Chair, Khronos OpenGL ARB Working Group)
Benj Lipchak, AMD
Benjamin Lipchak, Apple
Bill Licea-Kane, AMD (GL_ARB_texture_gather)
Brian Harris, Id Software
Brian Paul, VMware
Bruce Merry, ARM (Detailed specification review)
Cass Everitt, Epic Games
Chris Dodd, NVIDIA
Daniel Koch, TransGaming Inc.

OpenGL 4.2 (Core Profile) - April 27, 2012
J.4. CREDITS AND ACKNOWLEDGEMENTS

Dave Shreiner, ARM
Eric Boumaour, AMD
Eric Werness, NVIDIA (GL_ARB_sample_shading, GL_ARB_texture_gather, GL_ARB_texture_query_lod)
Eric Zolnowski, AMD (GL_ARB_transform_feedback2)
Evan Hart, AMD
Graham Sellers, AMD
Gregory Roth, NVIDIA
Ian Romanick, Intel
Ian Stewart, NVIDIA
Ignacio Castano, NVIDIA
Jaakko Konttinen, AMD
James Helferty, TransGaming Inc.
James Jones, NVIDIA Corporation
Jason Green, TransGaming Inc.
Jeff Bolz, NVIDIA (GL_ARB_draw_buffers_blend, GL_ARB_draw_indirect, GL_ARB_shader_subroutine, GL_ARB_texture_buffer_object_rgb32)
Jeremy Sandmel, Apple (Chair, ARB Nextgen (OpenGL 3.3) TSG)
John Kessenich, Intel (OpenGL Shading Language Specification Editor)
John Rosasco, Apple
Jon Leech, Independent (OpenGL API Specification Editor)
Lijun Qu, AMD
Mais Alnasser, AMD
Mark Callow, HI Corp
Mark Young, AMD (GL_ARB_draw_buffers_blend)
Maurice Ribble, Qualcomm
Michael Gold, NVIDIA
Mike Strauss, NVIDIA
Mike Weiblen, Zebra Imaging
Murat Balci, AMD
Neil Trevett, NVIDIA (President, Khronos Group)
Nick Haemel, AMD (GL_ARB_texture_cube_map_array)
Pat Brown, NVIDIA (GL_ARB_draw_indirect, GL_ARB_gpu_shader5, GL_ARB_gpu_shader_fp64, GL_ARB_sample_shading, GL_ARB_tessellation_shader, GL_ARB_texture_gather, GL_ARB_texture_query_lod, GL_ARB_transform_feedback2, GL_ARB_transform_feedback3)
Patrick Doane, Blizzard
Pierre Boudier, AMD

OpenGL 4.2 (Core Profile) - April 27, 2012
J.4. CREDITS AND ACKNOWLEDGEMENTS

Piers Daniell, NVIDIA
Piotr Uminski, Intel
Remi Arnaud, Sony
Rob Barris
Robert Simpson, Qualcomm
Timothy Lamb, AMD
Tom Olson, ARM
Tom Olson, TI (Chair, Khronos OpenGL ES Working Group)
Yanjun Zhang, S3 Graphics
Yunjun Zhang, AMD

The ARB gratefully acknowledges administrative support by the members of Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, and Michelle Clark, and technical support from James Riordon, webmaster of Khronos.org and OpenGL.org.
Appendix K

Version 4.1

OpenGL version 4.1, released on July 26, 2010, is the thirteenth revision since the original version 1.0.

Separate versions of the OpenGL 4.1 Specification exist for the core and compatibility profiles described in appendix E, respectively subtitled the “Core Profile” and the “Compatibility Profile”. This document describes the Core Profile. An OpenGL 4.1 implementation must be able to create a context supporting the core profile, and may also be able to create a context supporting the compatibility profile.

Material specific to the compatibility profile specification is marked in a distinct color to clearly call out differences between the two profiles.

The OpenGL 4.1 compatibility and core profiles are upward compatible with the OpenGL 4.0 compatibility and core profiles, respectively (see appendix J).

Following are brief descriptions of changes and additions to OpenGL 4.1.

K.1 New Features

New features in OpenGL 4.1, including the extension or extensions if any on which they were based, include:

- Improved OpenGL ES 2.0 compatibility by adding features previously found only in OpenGL ES 2.0 and not OpenGL 4.0 (GL_ARB_ES2_compatibility).

- Commands to retrieve and set the binary represtation of a program object (GL_ARB_get_program_binary).

- Increases in the required supported sizes for textures and renderbuffers.
K.2. DEPRECATION MODEL

- Ability to mix-and-match separately compiled shader objects defining different shader stages (GL_ARB_separate_shader_objects).
- Clarified restrictions on the precision requirements for shaders in the OpenGL Shading Language Specification (GL_ARB_shader_precision).
- OpenGL Shading Language support for vertex shader inputs with 64-bit floating-point components, and OpenGL API support for specifying the values of those inputs (GL_ARB_vertex_attrib_64bit).
- Expose multiple viewports for use with geometry shader outputs and multiple framebuffer attachments, and floating-point viewport bounds (GL_ARB_viewport_array).

K.2 Deprecation Model

No new features are deprecated by OpenGL 4.1. Features deprecated by OpenGL 4.0 remain deprecated, but have not yet been removed.

K.3 Changed Tokens

New token names are introduced to be used in place of old, less general names. However, the old token names continue to be supported, for backwards compatibility with code written for previous versions of OpenGL. The new names, and the old names they replace, are shown in table K.1.

<table>
<thead>
<tr>
<th>New Token Name</th>
<th>Old Token Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVE_PROGRAM</td>
<td>CURRENT_PROGRAM</td>
</tr>
</tbody>
</table>

Table K.1: New token names and the old names they replace.

K.4 Change Log

K.5 Credits and Acknowledgements

OpenGL 4.1 is the result of the contributions of many people and companies. Members of the Khronos OpenGL ARB Working Group during the development of OpenGL 4.1, including the company that they represented at the time of their
contributions, follow. Some major contributions made by individuals are listed togetherv
with their name, including specific functionality developed in the form of new ARB extensions together with OpenGL 4.1. In addition, many people participated in developing earlier vendor and EXT extensions on which the OpenGL 4.1 functionality is based in part; those individuals are listed in the respective extension specifications in the OpenGL Extension Registry.

Acorn Pooley, NVIDIA
Ahmet Oguz Akyuz, AMD
Alexis Mather, AMD
Andrew Lewycky, AMD
Anton Staaf, Google
Aske Simon Christensen, ARM
Avi Shapira, Graphic Remedy
Barthold Lichtenbelt, NVIDIA (Chair, Khronos OpenGL ARB Working Group)
Benji Bowman, Imagination Technologies
Benjamin Lipchak, Apple (GL_ARB_get_program_binary)
Bill Licea-Kane, AMD (Chair, ARB OpenGL Shading Language TSG)
Brian Paul, VMWare
Bruce Merry, ARM (Detailed specification review)
Chris Dodd, NVIDIA
Chris Marrin, Apple
Daniel Koch, TransGaming
David Garcia, AMD
Eric Werness, NVIDIA
Gavriel State, TransGaming
Georg Kolling
Graham Sellers, AMD (GL_ARB_shader_stencil_export, GL_ARB_vertex_attrib_64bit, GL_ARB_viewport_array)
Gregory Roth, NVIDIA (GL_ARB_get_program_binary, GL_ARB_separate_shader_objects)
Ian Romanick, Intel
Ian Stewart, NVIDIA
Jaakko Konttinen, AMD (GL_ARB_debug_output)
Jacob Ström, Ericsson AB
James Jones, NVIDIA
James Riordon, khronos.org
Jason Green, TransGaming
Jeff Bolz, NVIDIA (GL_ARB_ES2_compatibility)
Jeff Daniels

OpenGL 4.2 (Core Profile) - April 27, 2012
K.5. CREDITS AND ACKNOWLEDGEMENTS

Jeremy Sandmel, Apple (Chair, ARB Nextgen TSG)
Joey Blankenship
John Kessenich, Intel (OpenGL Shading Language Specification Editor, GL_ABI_ARB_vertex_attrib_64bit)
Jon Leech, Independent (OpenGL API Specification Editor, GL_ARB_cl_sync, GLX_ARB_create_context_robustness, WGL_ARB_create_context_robustness)
Kenneth Russell, Google
Kent Miller, Apple
Kevin Rogovin
Lingjun (Frank) Chen, Qualcomm
Mais Afnassser, AMD
Mark Callow, HI Corporation
Mark Kilgard, NVIDIA (GL_ARB_robustness, GL_ARB_separate_shader_objects)
Mark Young, AMD
Mike Weiblen, Zebra Imaging
Neil Trevett, NVIDIA
Nick Haemel, AMD
Pat Brown, NVIDIA (GL_ARB_separate_shader_objects, GL_ARB_vertex_attrib_64bit)
Patrick Doane, Blizzard
Pierre Boudier, AMD
Piers Daniell, NVIDIA (GL_ARB_get_program_binary, GL_ARB_vertex_attrib_64bit)
Piotr Uminski, Intel
Rob Barris
Robert Ohannessian
Robert Simpson, Qualcomm
Tom Olson, ARM (Chair, Khronos OpenGL ES Working Group)
Vladimir Vukicevic, Mozilla
Yaki Tebeka, Graphic Remedy
Yanjun Zhang, S3 Graphics

The ARB gratefully acknowledges administrative support by the members of Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, and Michelle Clark, and technical support from James Riordon, webmaster of Khronos.org and OpenGL.org.
Appendix L

Version 4.2

OpenGL version 4.2, released on August 8, 2011, is the fourteenth revision since the original version 1.0.

Separate versions of the OpenGL 4.2 Specification exist for the core and compatibility profiles described in appendix E, respectively subtitled the “Core Profile” and the “Compatibility Profile”. This document describes the Core Profile. An OpenGL 4.2 implementation must be able to create a context supporting the core profile, and may also be able to create a context supporting the compatibility profile.

Material specific to the compatibility profile specification is marked in a distinct color to clearly call out differences between the two profiles.

The OpenGL 4.2 compatibility and core profiles are upward compatible with the OpenGL 4.1 compatibility and core profiles, respectively (see appendix K).

Following are brief descriptions of changes and additions to OpenGL 4.2.

L.1 New Features

New features in OpenGL 4.2, including the extension or extensions if any on which they were based, include:

- Support for BPTC compressed textures (ARB_texture_compression_bptc).
- Allow pixel storage parameters to affect packing and unpacking of compressed textures (ARB_compressed_texture_pixel_storage).
- Shader atomic counters (ARB_shader_atomic_counters).
- Immutable texture images (ARB_texture_storage).
L.2. DEPRECATION MODEL

- Instanced transformed feedback drawing (ARB_transform_feedback_instanced).

- Allow the offset within buffer objects used for instanced rendering to be specified (ARB_base_instance).

- OpenGL Shading Language built-in functions allowing loads from and stores to texture images from any shader stage, and application control over the ordering of image load/store operations relative to other OpenGL pipeline operations accessing the same memory (ARB_shader_image_load_store).

- New OpenGL Shading Language features with no OpenGL API impact (ARB_conservative_depth and ARB_shading_language_420pack - see the OpenGL Shading Language Specification for details).

- Queries for sample counts available for a given internal format and usage (ARB_internalformat_query).

- More restrictive alignment constraints for mapped buffers (ARB_map_buffer_alignment).

L.2 Deprecation Model

The following features are newly deprecated by the OpenGL 4.2 core profile:

- The query targets NUM_COMPRESSED_TEXTURE_FORMATS and COMPRESSED_TEXTURE_FORMATS (see section 3.9.3).

Features deprecated by OpenGL 4.1 remain deprecated, but have not yet been removed.

L.3 Changed Tokens

New token names are introduced to be used in place of old, less general names. However, the old token names continue to be supported, for backwards compatibility with code written for previous versions of OpenGL. The new names, and the old names they replace, are shown in table L.1.

OpenGL 4.2 (Core Profile) - April 27, 2012
Table L.1: New token names and the old names they replace.

<table>
<thead>
<tr>
<th>New Token Name</th>
<th>Old Token Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>COPY_READ_BUFFER_BINDING</td>
<td>COPY_READ_BUFFER</td>
</tr>
<tr>
<td>COPY_WRITE_BUFFER_BINDING</td>
<td>COPY_WRITE_BUFFER</td>
</tr>
<tr>
<td>TRANSFORM_FEEDBACK_ACTIVE</td>
<td>TRANSFORM_FEEDBACK_BUFFER_ACTIVE</td>
</tr>
<tr>
<td>TRANSFORM_FEEDBACK_PAUSED</td>
<td>TRANSFORM_FEEDBACK_BUFFER_PAUSED</td>
</tr>
</tbody>
</table>

L.4 Change Log

Changes in the specification update of April 27, 2012:

- Make terminology of **NULL** for pointers and **null-terminated** for strings consistent (several places, not explicitly called out here) (Bug 7996).

- Numerous minor corrections of ”GL” to ”The GL” throughout the specification (Bug 7985).

- Replace use of `clampf` and `clampd` types with `float` and `double`, ensuring that clamping to [0, 1] is described explicitly for each such call. There will be a corresponding change to header files, .spec files, and man pages. This change affects the description of the types in section 2.3 as well as `DepthRange*` (section 2.14.1), `MinSampleShading` (section 3.3.1), `SampleCoverage` (section 4.1.3), `BlendColor` (section 4.1.7), and `ClearColor` and `ClearDepth*` (section 4.2.3) (Bug 7978).

- Use “clip volume” consistently in sections 2.4 and 2.5 replacing “viewing volume” (Bug 7921).

- Add an escape clause to the general language in section 2.5 so that commands such as `GetAttribLocation` can return non-zero values when errors are generated (Bug 7920).

- Clean up language describing `VertexAttrib*` commands in section 2.7, and note undefined behavior when specifying attributes with `VertexAttrib*Pointer` commands whose type is incompatible with the type of the attribute being specified in section 2.8 (Bug 8288).

- Change core profile definition of `DrawElementsOneInstance` in section 2.8.3 so that current generic attribute values are not affected by the command (Bugs 4260, 7989).
L.4. CHANGE LOG

- Update language for drawing commands in section 2.8.3 to properly describe instancing, and match language in OpenGL ES specs as much as possible (Bugs 7004, 8509).

- Added additional const-correctness to prototypes for 
  MultiDrawElements and MultiDrawElementsBaseVertex (section 2.8.3), 
  ShaderSource (section 2.11.1), 
  CreateShaderProgramv (section 2.11.3), 
  GetUniformIndices (section 2.11.7), and 
  TransformFeedbackVaryings (section 2.11.11) (Bug 7157).

- Add validation of the primcount parameter to pseudocode for several different drawing calls in section 2.8.3 (Bug 7923).

- Specify that deletion of names marked as used by Gen* but which have not yet acquired object state will mark those names as again unused, for each of 
  DeleteBuffers (section 2.9), 
  DeleteVertexArrays (section 2.10), 
  DeleteQueries (section 2.15), 
  DeleteTransformFeedbacks (section 2.17.1), 
  DeleteProgramPipelines (section 2.11.4), 
  DeleteTextures (section 3.9.1), 
  DeleteSamplers (section 3.9.2), 
  DeleteFramebuffers (section 4.4.1), and 
  DeleteRenderbuffers (section 4.4.2) (Bug 7972).

- Use new token names COPY_READ_BUFFER_BINDING and COPY_WRITE_BUFFER_BINDING in table 2.8 and section 2.9.5 and table 6.63, and 
  TRANSFORM_FEEDBACK_ACTIVE and TRANSFORM_FEEDBACK_PAUSED in table 6.43, as summarized in table L.1. Change GenFramebuffers ids formal parameter name to framebuffers in section 4.4.1 to match headers (Bug 8475).

- Add an INVALID_OPERATION error in section 2.9.3 for MapBufferRange when length is zero (and equivalently, for MapBuffer when the buffer size is zero) (Bug 7909).

- Define the name space for vertex array objects in section 2.10, as is done for other object types (Bug 7985).

- Describe global error behavior dependent on a program object’s link status under LinkProgram in section 2.11.3, and remove consequent redundant wording in descriptions of phrasing for queries 
  GetActiveAttrib (section 2.11.6); 
  GetUniformBlockIndex, GetActiveUniformBlockName, GetActiveUniformBlockiv, GetActiveAtomicCounterBufferiv, GetUniformIndices, GetActiveUniformName, GetActiveUniform, and GetActiveUniformsiv (section 2.11.7); and 
  GetTransformFeedbackVarying (section 2.11.11) (Bug 8501).
- Specify that ProgramBinary, as well as LinkProgram, installs new executable code into active shader state in sections 2.11.3 and 2.11.6 (Bugs 7927, 7928).

- Clarify that no specific program binary formats are defined by OpenGL itself, and update the language for the binary format queries to match the shader binary format queries in section 2.11.5 (Bug 7985).

- Clarify that unused attributes in a shader do not count as active attributes, rather than saying they “will not count against the limit”, in section 2.11.6 (Bug 7972).

- Reinstate language constraining the statement that uniform matrices consume no more than $4 \times \min(r, c)$ or $8 \times \min(r, c)$ uniform components to apply only in the default uniform block, and sync up the language describing combined limits for all shaders types in sections 2.11.7, 2.12.1, 2.12.3, 2.13.3, and 3.10.1 (Bug 5432).

- Update introduction to section 2.11.7 so that uniforms in UBOs are not described as part of project object state (Bug 7935).

- Drop redundant error condition from GetActiveUniformBlockiv in section 2.11.7 (Bug 7936).

- Specify that strings passed into GetUniformIndices must be null-terminated in section 2.11.7 (Bug 7938).

- Specify that uniform locations for sequential array indices are not required to be sequential in section 2.11.7 (Bug 8098).

- Fix GLSL type name uint in table 2.13 (Bug 8591).

- Update description of texture access from shadow samplers in section 2.11.12 to match that for fragment shaders (Bug 7962).

- Correct typos in sections 2.15 and 4.4.2 (Bug 7895).

- Define EndTransformFeedback in section 2.17.2 to do an implicit resume if transform feedback is active and not paused (Bug 7904).

- Add error condition to ResumeTransformFeedback in section 2.17.2 when the current program object has been re-linked since transform feedback became active for the current transform feedback object (Bug 8666).
L.4. CHANGE LOG

- Clarify pairing requirement on BeginTransformFeedback and EndTransformFeedback in section 2.17.2 (Bug 8664).

- Note at the end of the introduction to section 3 that rasterization never produces fragments for not corresponding to framebuffer pixels (Bug 7889).

- Add a floating-point column to the pixel types in table 3.2, and expand the generic language in section 3.7.2 so that floating-point type and INTEGER format arguments to commands specifying pixel rectangles always generate an INVALID_OPERATION error. Remove language describing this error for TexImage* commands in section 3.9.3 and ReadPixels in section 4.3.1. Note that there was previously an inconsistency between the generic language requiring a INVALID_ENUM error in this case and the command-specific language requiring a INVALID_OPERATION error. The latter error is now required of all commands specifying pixel rectangles (Bug 7981).

- Remove TEXTURE_WRAP_R from the constraints on setting sampler and texture parameter state in sections 3.9.2 and 3.9.8, and specify in section 3.9.2 that state which does not affect sampling for the type of texture being sampled also does not affect completeness (Bug 7942).

- Correct description of CopyTexImage2D to a forward reference to CopyPixels, rather than ReadPixels, in section 3.9.4 (Bug 7985).

- Add RGB565 format to table 3.12 (Bug 8530).

- Change description of internal texture format for the CompressedTexSubImage commands in section 3.9.5 so that it’s not referred to as a parameter of the commands (Bug 8876).

- Clean up description of TexImage*Multisample commands in section 3.9.6. Add an error when samples is zero, add clamping to closest supported number of samples, specify the initial state of relevant texture image parameters, and specify undefined results for GLSL queries of invalid sample indices (Bug 6509).

- Update caption for figure 3.7 to remove obsolete references to m and n (Bug 7988).

- Include texture swizzle state in summary of texture state in section 3.9.15 (Bug 8204).

- Treat the pixel unpack buffer as if it were zero during execution of TexStorage* commands in section 3.9.16 (Bug 8471).

OpenGL 4.2 (Core Profile) - April 27, 2012
L.4. CHANGE LOG

• Make results of OUT_OF_MEMORY errors during execution of TexStorage* commands undefined in section 3.9.16 (Bug 7815).

• Remove references to gl_FragColor and gl_FragData[n] from core profile in sections 3.10.2 and 4.2.1 (Bug 8549).

• Clean up underspecified alpha test behavior in section 4.1.3 to include cases where the draw framebuffer object has no color buffer, accommodate dual-source blending fragment color outputs, and require at least 8 bits of virtual alpha when there is no color buffer in the draw framebuffer (Bug 7419).

• Cleanup framebuffer language to clearly distinguish read and draw framebuffer objects in sections 4.1.11, 4.2.1, 4.3.1, 4.4.1, 4.4.2, 4.4.3, 4.4.4, and 4.4.5 (Bug 5578).

• Remove ALPHA pixel format from the core profile in two places it was inadvertently retained in the description of ReadPixels in section 4.3.1 (Bug 8465).

• Specify that writemasks are ignored by BlitFramebuffer in section 4.3.2 (Bug 7969).

• Remove description of read buffer as global state in section 4.3.3 in favor of the description of read buffer state under “Obtaining Pixels from the Framebuffer” in section 4.3.1 (Bug 5578).

• Specify that queries of properties of the read framebuffer are derived from the FBO bound to the read framebuffer, not the draw framebuffer, in section 4.4.1 and table 6.24 (Bug 8454).

• Add multisample texture types to the list of images that can be bound to framebuffer objects in section 4.4.2 (Bug 5578).

• Clarify framebuffer completeness language to allow for both renderbuffer and texture image attachments in section 4.4.4 (Bug 6837).

• Clean up description of QueryCounter in section 5.1 to specify the type of query object created, require that existing query objects be of the correct type, and allow creation of new query objects from non-marked names in the compatibility profile (Bugs 7662,8104).

• Clarify unblocking behavior of FenceSync in section 5.3 (Bug 8235).

• Specify in section 6.1.2 that bitmasks are treated as signed values when queried with GetIntegerv, to avoid truncation (Bug 8440).

OpenGL 4.2 (Core Profile) - April 27, 2012
L.4. CHANGE LOG

• Clarify in section 6.1.3 that queries of texture internal format return the format as specified at texture creation time (Bug 5275).

• Change minimum number of bits for the SAMPLES_PASSED query from a viewport-dependent calculation to 32 in section 6.1.7 (Bug 7795).

• Back out added const-correctness in prototypes for GetBufferPointerv (section 6.1.9) and GetVertexAttribPointerv (section 6.1.12) (Bug 7157).

• Add description of atomic counter buffer object binding queries to section 6.1.9, matching descriptions for uniform and transform feedback buffer object bindings. Fix misspelled query name to GetActiveAtomicCounterBufferiv in table 6.39 (Bug 7718).

• Moved state table entries for MAX_COLOR_ATTACHMENTS, MAX_DRAW_BUFFERS, and MAX_DUAL_SOURCE_DRAW_BUFFERS from framebuffer-dependent state to implementation-dependent state in table 6.59 (Bug 7990).

• Correct number of copies of CURRENT_QUERY state in table 6.63 and remove redundant unqueriable “Occlusion query active” state (Bug 7844).

• Note in appendix D.1.2 that automatic unbinding of deleted objects affects only the binding state, not any related state specified along with the bind (Bug 7730).

Changes in the specification update of January 19, 2012:

• Corrections to figure 2.1 (Bug 7997).

• Minor bugfixes and typos in sections 2.4, 2.8, 2.11, 2.11.6, 2.11.12, 2.15, 2.17.3, 3.5.2 (restored description of non-antialiased wide line rendering to the core profile since they are deprecated, but not yet removed), 3.9.2 (fixed prototypes for SamplerParameter commands), 3.10.2, 4.1.11 (specify that multisample buffer is only resolved at this time if the default framebuffer is bound), 4.4.2 (correct limits on layer for different types of attached textures), 4.4.4, 6.1.4 (remove redundant description by IsTexture that unbound object names created by GenTextures are not the names of texture objects), 6.2 (add GetInteger64v as a supported state query), appendix D, and tables 6.16 (correct for multiple instances of texture swizzle state), 6.31, 6.32, 6.49, and 6.61 (Bug 7895).

• Add missing automatic unbinding of previously bound buffer objects for BindBufferRange and BindBufferBase in section 2.9.1 (Bug 8196).
L.4. CHANGE LOG

• More clearly specify interface matching rules for shader inputs and outputs in section 2.11.4, for cases where both sides of an interface are found in the same program and where they are in different programs (Bug 7030).

• Clarify in section 2.11.6 that dvec3 and dvec4 vertex shader inputs consume only a single attribute location for the purpose of matching inputs to generic vertex attributes, but may consume two vectors for the purposes of determining if too many attribute vectors are used (Bug 7809). Also, add missing language describing the set of attributes consumed by matrix vertex attributes, with fixes to explicitly address dmat* types.

• Remove dangling references to nonexistent gl_VerticesOut in section 2.12.1 (Bug 8357).

• Fix names of cube map sampler type tokens in table 2.13 (Bug 8303).

• Fix behavior of DeleteTransformFeedbacks in section 2.17.1 to generate an error if any of the objects being deleted has transform feedback active (Bug 8323).

• Remove ambiguity in the order of operations and which vertices are appended by transform feedback when it is resumed in section 2.17.2 (Bug 8202).

• Updated description of errors resulting from specifying texture images of level 1 or greater which exceed implementation-dependent limits, in sections 3.9.3 and 3.9.14 (Bug 8210).

• Remove clamping of $D_t$ and $D_{ref}$ prior to depth texture comparison in section 3.9.17, since it doesn’t reflect hardware reality (Bug 7975).

• Update description of texture access from shadow samplers in section 3.10.2 to interact with texture swizzle (Bug 7962) and clarify that swizzling is not performed on the results of incomplete texture lookups (Bug 7917).

• Add buffer clearing to the list of operations affected by scissor rectangle zero in section 4.1.2 (Bug 8368).

• Remove error (from the core profile only) for querying CURRENT_VERTEX_ATTRIB for attribute zero with GetVertexAttrib* in section 6.1.12 (Bug 8352).

• Clarify that the initial state of SAMPLE_MASK_VALUE is for all bits to be set in table 6.13 (Bug 8441).

OpenGL 4.2 (Core Profile) - April 27, 2012
L.4. CHANGE LOG

- Add missing `PROGRAM_SEPARABLE` state to table 6.32 (Bug 8442).
- Numerous minor fixes to state table type fields and formatting (Bugs 8430, 8431).
- Clarified that automatic unbinding of deleted objects, as described in section D.1.2, does not affect attachments to unbound container objects the deleted objects are themselves attached to (Bug 8233).
- Add version in which several extensions were introduced to core GL in section M.3 (Bug 8418).

Changes in the specification update of August 22, 2011:

- More clearly specify interface matching rules for shader inputs and outputs in section 2.11.4, for cases where both sides of an interface are found in the same program and where they are in different programs (Bug 7030).
- Clarify in section 2.11.6 that `dvec3` and `dvec4` vertex shader inputs consume only a single attribute location for the purpose of matching inputs to generic vertex attributes, but may consume two vectors for the purposes of determining if too many attribute vectors are used (Bug 7809). Also, add missing language describing the set of attributes consumed by matrix vertex attributes, with fixes to explicitly address `dmat*` types.

Changes in the released specification of August 8, 2011:

- Update name of `MIN_MAP_BUFFER_ALIGNMENT` to follow GL conventions in section 2.9.3 and table 6.49 (Bug 7825).
- Change query object state description in section 2.15 so the initial state of the query result available flag agrees with the state table (Bug 7823).
- Minor cleanups to atomic counter language in section 2.11.7 and to atomic counter token names in tables 6.51, 6.52, 6.53, and 6.54 (Bug 7834).
- Clarify that completeness affects texture lookup and fetch operations in all shader stages in section 3.9.14 (Bug 7856).
- Change `BindImageTexture` parameter name from `index` to `unit` and fix minor language issues in section 3.9.20 (Bugs 7744, 7850, 7851).
- Fix typos in section 6.1.15 (Bug 7843).
• Fix minimum maximums for MAX_FRAGMENT_IMAGE_UNIFORMS and MAX_COMBINED_IMAGE_UNIFORMS in table 6.56 (Bug 7805).

• Change minimum maximum for MAX_ATOMIC_COUNTER_BUFFER_SIZE to 32 in table 6.55 (Bug 7855).

L.5 Credits and Acknowledgements

OpenGL 4.2 is the result of the contributions of many people and companies. Members of the Khronos OpenGL ARB Working Group during the development of OpenGL 4.2, including the company that they represented at the time of their contributions, follow. Some major contributions made by individuals are listed together with their name, including specific functionality developed in the form of new ARB extensions together with OpenGL 4.2. In addition, many people participated in developing earlier vendor and EXT extensions on which the OpenGL 4.2 functionality is based in part; those individuals are listed in the respective extension specifications in the OpenGL Extension Registry.

Acorn Pooley, NVIDIA
Barthold Lichtenbelt, NVIDIA (Chair, Khronos OpenGL ARB Working Group)
Benji Bowman, Imagination Technologies
Bill Licea-Kane (Chair, ARB OpenGL Shading Language TSG, ARB_shader_atomic_counters)
Bruce Merry, ARM (Detailed specification review, ARB_texture_storage)
Chris Dodd, NVIDIA
Christophe Riccio, Imagination Technologies
Daniel Koch (ARB_internalformat_query)
Eric Werness, NVIDIA (ARB_texture_compression_bptc)
Graham Sellers, AMD (ARB_base_instance, ARB_conservative_depth, ARB_transform_feedback_instanced)
Greg Roth, NVIDIA
Ian Romanick, Intel (ARB_texture_storage)
Jacob Ström, Ericsson AB
Jan-Harald Fredriksen (ARB_internalformat_query)
Jeannot Breton, NVIDIA
Jeff Bolz, NVIDIA Corporation (ARB_shader_image_load_store)
Jeremy Sandmel, Apple
John Kessenich, Independent (OpenGL Shading Language Specification Editor, ARB_shading_language_420pack)
Jon Leech, Independent (OpenGL API Specification Editor)
L.5. CREDITS AND ACKNOWLEDGEMENTS

Lingjun (Frank) Chen, Qualcomm
Mark Callow, HI Corporation
Maurice Ribble, Qualcomm
Nick Haemel, AMD
Pat Brown, NVIDIA Corporation (ARB_shader_image_load_store, ARB_shading_language_packing)
Patrick Doane, Blizzard
Pierre Boudier, AMD
Piers Daniell, NVIDIA Corporation (ARB_compressed_texture_pixel_storage, ARB_map_buffer_alignment)
Robert Simpson, Qualcomm
Tom Olson, ARM (Chair, Khronos OpenGL ES Working Group)

The ARB gratefully acknowledges administrative support by the members of Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, and Michelle Clark, and technical support from James Riordon, webmaster of Khronos.org and OpenGL.org.
Appendix M

Extension Registry, Header Files, and ARB Extensions

M.1 Extension Registry

Many extensions to the OpenGL API have been defined by vendors, groups of
vendors, and the OpenGL ARB. In order not to compromise the readability of the
OpenGL Specification, such extensions are not integrated into the core language;
instead, they are made available online in the OpenGL Extension Registry, together
with extensions to window system binding APIs, such as GLX and WGL, and with
specifications for OpenGL, GLX, and related APIs.

Extensions are documented as changes to a particular version of the Specifica-
tion. The Registry is available on the World Wide Web at URL

http://www.opengl.org/registry/

M.2 Header Files

Historically, C and C++ source code calling OpenGL was to #include a single
header file, <GL/gl.h>. In addition to the core OpenGL API, the APIs for all
extensions provided by an implementation were defined in this header.

When platforms became common where the OpenGL SDK (library and header
files) were not necessarily obtained from the same source as the OpenGL driver,
such as Microsoft Windows and Linux, <GL/gl.h> could not always be kept
in sync with new core API versions and extensions supported by drivers. At this
time the OpenGL ARB defined a new header, <GL/glext.h>, which could be
obtained directly from the OpenGL Extension Registry (see section M.1). The
combination of `<GL/gl.h>` and `<GL/glext.h>` always defines all APIs for all profiles of the latest OpenGL version, as well as for all extensions defined in the Registry.

`<GL3/gl3.h>` defines APIs for the core profile of OpenGL, together with ARB extensions compatible with the core profile. It does not include APIs for features only in the compatibility profile or for other extensions.

`<GL3/gl3ext.h>` defines APIs for additional ARB, EXT, and vendor extensions compatible with the core profile, but not defined in `<GL3/gl3.h>`. Most older extensions are not compatible with the core profile.

Applications using the compatibility profile (see appendices I and E) should include the traditional `<GL/gl.h>` and `<GL/glext.h>` headers.

Applications using the core profile should include the new `<GL3/gl3.h>` and `<GL3/gl3ext.h>` headers introduced with OpenGL 3.1.

By using `<GL3/gl3.h>` and `<GL3/gl3ext.h>`, instead of the legacy `<GL/gl.h>` and `<GL/glext.h>`, newly developed applications are given increased protection against accidentally using a “legacy” feature that has been removed from the core profile. This can assist in developing applications on a GL implementation that supports the compatibility profile when the application is also intended to run on other platforms supporting only the core profile.

Developers should always be able to download `<GL3/gl3.h>` and `<GL3/gl3ext.h>` from the Registry, with these headers replacing, or being used in place of older versions that may be provided by a platform SDK.

### M.3  ARB Extensions

OpenGL extensions that have been approved by the OpenGL Architectural Review Board (ARB) are summarized in this section. ARB extensions are not required to be supported by a conformant OpenGL implementation, but are expected to be widely available; they define functionality that is likely to move into the required feature set in a future revision of the specification.

#### M.3.1  Naming Conventions

To distinguish ARB extensions from core OpenGL features and from vendor-specific extensions, the following naming conventions are used:

- A unique name string of the form "GL_ARB_name" is associated with each extension. If the extension is supported by an implementation, this string will be among the EXTENSIONS strings returned by `GetStringi`, as described in section 6.1.5.

OpenGL 4.2 (Core Profile) - April 27, 2012
M.3. ARB EXTENSIONS

- All functions defined by the extension will have names of the form *FunctionARB*

- All enumerants defined by the extension will have names of the form *NAME_ARB*.

- In addition to OpenGL extensions, there are also ARB extensions to the related GLX and WGL APIs. Such extensions have name strings prefixed by "GLX_" and "WGL_" respectively. Not all GLX and WGL ARB extensions are described here, but all such extensions are included in the registry.

M.3.2 Promoting Extensions to Core Features

ARB extensions can be *promoted* to required core features in later revisions of OpenGL. When this occurs, the extension specifications are merged into the core specification. Functions and enumerants that are part of such promoted extensions will have the *ARB* affix removed.

Implementations of such later revisions should continue to export the name strings of promoted extensions in the *EXTENSIONS* strings and continue to support the *ARB*-affixed versions of functions and enumerants as a transition aid.

For descriptions of extensions promoted to core features in OpenGL 1.3 and beyond, see the corresponding version of the OpenGL specification, or the descriptions of that version in version-specific appendices to later versions of the specification.

M.3.3 Multitexture

The name string for multitexture is *GL_ARB_multitexture*. It was promoted to a core feature in OpenGL 1.3.

M.3.4 Transpose Matrix

The name string for transpose matrix is *GL_ARB_transpose_matrix*. It was promoted to a core feature in OpenGL 1.3.

M.3.5 Multisample

The name string for multisample is *GL_ARB_multisample*. It was promoted to a core feature in OpenGL 1.3.
M.3.6 Texture Add Environment Mode

The name string for texture add mode is GL_ARB_texture_env_add. It was promoted to a core feature in OpenGL 1.3.

M.3.7 Cube Map Textures

The name string for cube mapping is GL_ARB_texture_cube_map. It was promoted to a core feature in OpenGL 1.3.

M.3.8 Compressed Textures

The name string for compressed textures is GL_ARB_texture_compression. It was promoted to a core feature in OpenGL 1.3.

M.3.9 Texture Border Clamp

The name string for texture border clamp is GL_ARB_texture_border_clamp. It was promoted to a core feature in OpenGL 1.3.

M.3.10 Point Parameters

The name string for point parameters is GL_ARB_point_parameters. It was promoted to a core features in OpenGL 1.4.

M.3.11 Vertex Blend

Vertex blending replaces the single model-view transformation with multiple vertex units. Each unit has its own transform matrix and an associated current weight. Vertices are transformed by all the enabled units, scaled by their respective weights, and summed to create the eye-space vertex. Normals are similarly transformed by the inverse transpose of the model-view matrices.

The name string for vertex blend is GL_ARB_vertex_blend.

M.3.12 Matrix Palette

Matrix palette extends vertex blending to include a palette of model-view matrices. Each vertex may be transformed by a different set of matrices chosen from the palette.

The name string for matrix palette is GL_ARB_matrix_palette.
M.3.13 Texture Combine Environment Mode

The name string for texture combine mode is `GL_ARB_texture_env_combine`. It was promoted to a core feature in OpenGL 1.3.

M.3.14 Texture Crossbar Environment Mode

The name string for texture crossbar is `GL_ARB_texture_env_crossbar`. It was promoted to a core feature in OpenGL 1.4.

M.3.15 Texture Dot3 Environment Mode

The name string for DOT3 is `GL_ARB_texture_env_dot3`. It was promoted to a core feature in OpenGL 1.3.

M.3.16 Texture Mirrored Repeat

The name string for texture mirrored repeat is `GL_ARB_texture_mirrored_repeat`. It was promoted to a core feature in OpenGL 1.4.

M.3.17 Depth Texture

The name string for depth texture is `GL_ARB_depth_texture`. It was promoted to a core feature in OpenGL 1.4.

M.3.18 Shadow

The name string for shadow is `GL_ARB_shadow`. It was promoted to a core feature in OpenGL 1.4.

M.3.19 Shadow Ambient

Shadow ambient extends the basic image-based shadow functionality by allowing a texture value specified by the `TEXTURE_COMPARE_FAIL_VALUE_ARB` texture parameter to be returned when the texture comparison fails. This may be used for ambient lighting of shadowed fragments and other advanced lighting effects.

The name string for shadow ambient is `GL_ARB_shadow_ambient`.

M.3.20 Window Raster Position

The name string for window raster position is `GL_ARB_window_pos`. It was promoted to a core feature in OpenGL 1.4.
M.3.21 Low-Level Vertex Programming

Application-defined *vertex programs* may be specified in a new low-level programming language, replacing the standard fixed-function vertex transformation, lighting, and texture coordinate generation pipeline. Vertex programs enable many new effects and are an important first step towards future graphics pipelines that will be fully programmable in an unrestricted, high-level shading language.

The name string for low-level vertex programming is `GL_ARB_vertex_program`.

M.3.22 Low-Level Fragment Programming

Application-defined *fragment programs* may be specified in the same low-level language as `GL_ARB_vertex_program`, replacing the standard fixed-function vertex texturing, fog, and color sum operations.

The name string for low-level fragment programming is `GL_ARB_fragment_program`.

M.3.23 Buffer Objects

The name string for buffer objects is `GL_ARB_vertex_buffer_object`. It was promoted to a core feature in OpenGL 1.5.

M.3.24 Occlusion Queries

The name string for occlusion queries is `GL_ARB_occlusion_query`. It was promoted to a core feature in OpenGL 1.5.

M.3.25 Shader Objects

The name string for shader objects is `GL_ARB_shader_objects`. It was promoted to a core feature in OpenGL 2.0.

M.3.26 High-Level Vertex Programming

The name string for high-level vertex programming is `GL_ARB_vertex_shader`. It was promoted to a core feature in OpenGL 2.0.

M.3.27 High-Level Fragment Programming

The name string for high-level fragment programming is `GL_ARB_fragment_shader`. It was promoted to a core feature in OpenGL 2.0.
M.3.28 OpenGL Shading Language

The name string for the OpenGL Shading Language is GL_ARB_shading_language_100. The presence of this extension string indicates that programs written in version 1 of the Shading Language are accepted by OpenGL. It was promoted to a core feature in OpenGL 2.0.

M.3.29 Non-Power-Of-Two Textures

The name string for non-power-of-two textures is GL_ARB_texture_non-power_of_two. It was promoted to a core feature in OpenGL 2.0.

M.3.30 Point Sprites

The name string for point sprites is GL_ARB_point_sprite. It was promoted to a core feature in OpenGL 2.0.

M.3.31 Fragment Program Shadow

Fragment program shadow extends low-level fragment programs defined with GL_ARB_fragment_program to add shadow 1D, 2D, and 3D texture targets, and remove the interaction with GL_ARB_shadow.

The name string for fragment program shadow is GL_ARB_fragment_program_shadow.

M.3.32 Multiple Render Targets

The name string for multiple render targets is GL_ARB_draw_buffers. It was promoted to a core feature in OpenGL 2.0.

M.3.33 Rectangular Textures

Rectangular textures define a new texture target TEXTURE_RECTANGLE_ARB that supports 2D textures without requiring power-of-two dimensions. Rectangular textures are useful for storing video images that do not have power-of-two sizes (POTS). Resampling artifacts are avoided and less texture memory may be required. They are also useful for shadow maps and window-space texturing. These textures are accessed by dimension-dependent (aka non-normalized) texture coordinates.

Rectangular textures are a restricted version of non-power-of-two textures. The differences are that rectangular textures are supported only for 2D; they require a new texture target; and the new target uses non-normalized texture coordinates.
The name string for texture rectangles is GL_ARB_texture_rectangle. It was promoted to a core feature in OpenGL 3.1.

M.3.34 Floating-Point Color Buffers

Floating-point color buffers can represent values outside the normal \([0, 1]\) range of colors in the fixed-function OpenGL pipeline. This group of related extensions enables controlling clamping of vertex colors, fragment colors throughout the pipeline, and pixel data read back to client memory, and also includes WGL and GLX extensions for creating frame buffers with floating-point color components (referred to in GLX as framebuffer configurations, and in WGL as pixel formats).

The name strings for floating-point color buffers are GL_ARB_color_buffer_float, GLX_ARB_fbconfig_float, and WGL_ARB_pixel_format_float. GL_ARB_color_buffer_float was promoted to a core feature in OpenGL 3.0.

M.3.35 Half-Precision Floating Point

This extension defines the representation of a 16-bit floating-point data format, and a corresponding type argument which may be used to specify and read back pixel and texture images stored in this format in client memory. Half-precision floats are smaller than full precision floats, but provide a larger dynamic range than similarly sized (short) data types.

The name string for half-precision floating-point is GL_ARB_half_float_pixel. It was promoted to a core feature in OpenGL 3.0.

M.3.36 Floating-Point Textures

Floating-point textures stored in both 32- and 16-bit formats may be defined using new internalformat arguments to commands which specify and read back texture images.

The name string for floating-point textures is GL_ARB_texture_float. It was promoted to a core feature in OpenGL 3.0.

M.3.37 Pixel Buffer Objects

The buffer object interface is expanded by adding two new binding targets for buffer objects, the pixel pack and unpack buffers. This permits buffer objects to be used to store pixel data as well as vertex array data. Pixel-drawing and -reading commands using data in pixel buffer objects may operate at greatly improved performance compared to data in client memory.
The name string for pixel buffer objects is `GL_ARB_pixel_buffer_object`. It was promoted to a core feature in OpenGL 2.1.

**M.3.38 Floating-Point Depth Buffers**

The name string for floating-point depth buffers is `GL_ARB_depth_buffer_float`. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier `GL_NV_depth_buffer_float` extension, and is provided to enable this functionality in older drivers.

**M.3.39 Instanced Rendering**

The name string for instanced rendering is `GL_ARB_draw_instanced`. It was promoted to a core feature in OpenGL 3.1.

**M.3.40 Framebuffer Objects**

The name string for framebuffer objects is `GL_ARB_framebuffer_object`. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier `GL_EXT_framebuffer_object`, `GL_EXT_framebuffer_multisample`, and `GL_EXT_framebuffer_blit` extensions, and is provided to enable this functionality in older drivers.

**M.3.41 sRGB Framebuffers**

The name string for sRGB framebuffers is `GL_ARB_framebuffer_sRGB`. It was promoted to a core feature in OpenGL 3.0. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier `GL_EXT_framebuffer_sRGB` extension, and is provided to enable this functionality in older drivers.

To create sRGB format surfaces for use on display devices, an additional pixel format (config) attribute is required in the window system integration layer. The name strings for the GLX and WGL sRGB pixel format interfaces are `GLX_ARB_framebuffer_sRGB` and `WGL_ARB_framebuffer_sRGB` respectively.

**M.3.42 Geometry Shaders**

This extension defines a new shader type called a *geometry shader*. Geometry shaders are run after vertices are transformed, but prior to the remaining fixed-function vertex processing, and may generate new vertices for, or remove vertices from the primitive assembly process.
The name string for geometry shaders is GL_ARB_geometry_shader4. It was promoted to a core feature in OpenGL 3.2.

**M.3.43 Half-Precision Vertex Data**

The name string for half-precision vertex data GL_ARB_half_float_vertex. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier GL_NV_half_float extension, and is provided to enable this functionality in older drivers.

**M.3.44 Instanced Rendering**

This instanced rendering interface is a less-capable form of GL_ARB_draw_instanced which can be supported on older hardware. The name string for instanced rendering is GL_ARB_instanced_arrays. It was promoted to a core feature in OpenGL 3.3.

**M.3.45 Flexible Buffer Mapping**

The name string for flexible buffer mapping is GL_ARB_map_buffer_range. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier GL_APPLE_flush_buffer_range extension, and is provided to enable this functionality in older drivers.

**M.3.46 Texture Buffer Objects**

The name string for texture buffer objects is GL_ARB_texture_buffer_object. It was promoted to a core feature in OpenGL 3.1.

**M.3.47 RGTC Texture Compression Formats**

The name string for RGTC texture compression formats is GL_ARB_texture_compression_rgtc. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier GL_EXT_texture_compression_rgtc extension, and is provided to enable this functionality in older drivers.

It was promoted to a core feature in OpenGL 3.0.

**M.3.48 One- and Two-Component Texture Formats**

The name string for one- and two-component texture formats is GL_ARB_texture_rg. It was promoted to a core feature in OpenGL 3.0. This extension is
M.3. ARB EXTENSIONS

equivalent to new core functionality introduced in OpenGL 3.0, and is provided to enable this functionality in older drivers.

M.3.49 Vertex Array Objects

The name string for vertex array objects is GL_ARB_vertex_array_object. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier GL_APPLE_vertex_array_object extension, and is provided to enable this functionality in older drivers.

It was promoted to a core feature in OpenGL 3.0.

M.3.50 Versioned Context Creation

Starting with OpenGL 3.0, a new context creation interface is required in the window system integration layer. This interface specifies the context version required as well as other attributes of the context.

The name strings for the GLX and WGL context creation interfaces are GLX_ARB_create_context and WGL_ARB_create_context respectively.

M.3.51 Uniform Buffer Objects

The name string for uniform buffer objects is GL_ARB_uniform_buffer_object. This extension is equivalent to new core functionality introduced in OpenGL 3.1 and is provided to enable this functionality in older drivers.

M.3.52 Restoration of features removed from OpenGL 3.0

OpenGL 3.1 removes a large number of features that were marked deprecated in OpenGL 3.0 (see appendix G.2). GL implementations needing to maintain these features to support existing applications may do so, following the deprecation model, by exporting an extension string indicating those features are present. Applications written for OpenGL 3.1 should not depend on any of the features corresponding to this extension, since they will not be available on all platforms with 3.1 implementations.

The name string for restoration of features deprecated by OpenGL 3.0 is GL_ARB_compatibility.

The profile terminology introduced with OpenGL 3.2 eliminates the necessity for evolving this extension. Instead, interactions between features removed by OpenGL 3.1 and new features introduced in later OpenGL versions are defined by the compatibility profile corresponding to those versions.
M.3.53 Fast Buffer-to-Buffer Copies

The name string for fast buffer-to-buffer copies is GL_ARB_copy_buffer. This extension is equivalent to new core functionality introduced in OpenGL 3.1 and is provided to enable this functionality in older drivers.

M.3.54 Shader Texture Level of Detail Control

The name string for shader texture level of detail control is GL_ARB_shader_texture_lod. This extension is equivalent to new core functions introduced in OpenGL Shading Language 1.30 and is provided to enable this functionality in older versions of the shading language.

M.3.55 Depth Clamp Control

The name string for depth clamp control is GL_ARB_depth_clamp. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

M.3.56 Base Vertex Offset Drawing Commands

The name string for base vertex offset drawing commands is GL_ARB_draw_elements_base_vertex. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

M.3.57 Fragment Coordinate Convention Control

The name string for fragment coordinate convention control is GL_ARB_fragment_coord_conventions. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

M.3.58 Provoking Vertex Control

The name string for provoking vertex control is GL_ARB_provoking_vertex. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.
M.3.59 Seamless Cube Maps

The name string for seamless cube maps is `GL_ARB_seamless_cube_map`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

M.3.60 Fence Sync Objects

The name string for fence sync objects is `GL_ARB_sync`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

M.3.61 Multisample Textures

The name string for multisample textures is `GL_ARB_texture_multisample`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

M.3.62 BGRA Attribute Component Ordering

The name string for BGRA attribute component ordering is `GL_ARB_vertex_array_bgra`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

M.3.63 Per-Buffer Blend Control

The blending interface is extended to specify blend equation and blend function on a per-draw-buffer basis.

The name string for per-buffer blend control is `GL_ARB_draw_buffers_blend`. It was promoted to a core feature in OpenGL 4.0.

M.3.64 Sample Shading Control

Sample shading control adds the ability to request that an implementation use a minimum number of unique sets of fragment computation inputs when multisampling a pixel.

The name string for sample shading control is `GL_ARB_sample_shading`. It was promoted to a core feature in OpenGL 4.0.
M.3.65 Cube Map Array Textures

A cube map array texture is a two-dimensional array texture that may contain many cube map layers. Each cube map layer is a unique cube map image set.

The name string for cube map array textures is GL_ARB_texture_cube_map_array. It was promoted to a core feature in OpenGL 4.0.

M.3.66 Texture Gather

Texture gather adds a new set of texture functions (textureGather) to the OpenGL Shading Language that determine the $2 \times 2$ footprint used for linear filtering in a texture lookup, and return a vector consisting of the first component from each of the four texels in the footprint.

The name string for texture gather is GL_ARB_texture_gather. It was promoted to a core feature in OpenGL 4.0.

M.3.67 Texture Level-Of-Detail Queries

Texture level-of-detail queries adds a new set of fragment shader texture functions (textureLOD) to the OpenGL Shading Language that return the results of automatic level-of-detail computations that would be performed if a texture lookup were to be done.

The name string for texture level-of-detail queries is GL_ARB_texture_query_lod.

M.3.68 Profiled Context Creation

Starting with OpenGL 3.2, API profiles are defined. Profiled context creation extends the versioned context creation interface to specify a profile which must be implemented by the context.

The name strings for the GLX and WGL profiled context creation interfaces are GLX_ARB_create_context_profile and WGL_ARB_create_context_profile respectively.

M.3.69 Shading Language Include

Shading language include adds support for #include directives to shaders, and a named string API for defining the text corresponding to #include pathnames.

The name string for shading language include is GL_ARB_shading_language_include.

OpenGL 4.2 (Core Profile) - April 27, 2012
M.3.70  BPTC texture compression

BPTC texture compression provides new block compressed specific texture formats which can improve quality in images with sharp edges and strong chrominance transitions, and support high dynamic range floating-point formats.

The name string for bptc texture compression is GL_ARB_texture_compression_bptc.

M.3.71  Extended Blend Functions

The name string for extended blend functions is GL_ARB_blend_func_extended. This extension is equivalent to new core functionality introduced in OpenGL 3.3, and is provided to enable this functionality in older drivers.

M.3.72  Explicit Attribute Location

The name string for explicit attribute location is GL_ARB_explicit_attrib_location. This extension is equivalent to new core functionality introduced in OpenGL 3.3 and is provided to enable this functionality in older drivers.

M.3.73  Boolean Occlusion Queries

The name string for boolean occlusion queries is GL_ARB_occlusion_query2. This extension is equivalent to new core functionality introduced in OpenGL 3.3 and is provided to enable this functionality in older drivers.

M.3.74  Sampler Objects

The name string for sampler objects is GL_ARB_sampler_objects. This extension is equivalent to new core functionality introduced in OpenGL 3.3 and is provided to enable this functionality in older drivers.

M.3.75  Shader Bit Encoding

The name string for shader bit encoding is GL_ARB_shader_bit_encoding. This extension is equivalent to new core functionality introduced in OpenGL 3.3 and is provided to enable this functionality in older drivers.
M.3.76 RGB10A2 Integer Textures

The name string for RGB10A2 integer textures is GL_ARB_texture_rgb10_a2ui. This extension is equivalent to new core functionality introduced in OpenGL 3.3 and is provided to enable this functionality in older drivers.

M.3.77 Texture Swizzle

The name string for texture swizzle is GL_ARB_texture_swizzle. This extension is equivalent to new core functionality introduced in OpenGL 3.3 and is provided to enable this functionality in older drivers.

M.3.78 Timer Queries

The name string for timer queries is GL_ARB_timer_query. This extension is equivalent to new core functionality introduced in OpenGL 3.3 and is provided to enable this functionality in older drivers.

M.3.79 Packed 2.10.10.10 Vertex Formats

The name string for packed 2.10.10.10 vertex formats is GL_ARB_vertex_type_2_10_10_10_rev. This extension is equivalent to new core functionality introduced in OpenGL 3.3 and is provided to enable this functionality in older drivers.

M.3.80 Draw Indirect

The name string for draw indirect is GL_ARB_draw_indirect. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.

M.3.81 GPU Shader5 Miscellaneous Functionality

The name string for gpu shader5 miscellaneous functionality is GL_ARB_gpu_shader5. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.

M.3.82 Double-Precision Floating-Point Shader Support

The name string for double-precision floating-point shader support is GL_ARB_gpu_shader_fp64. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.
M.3.83 Shader Subroutines

The name string for shader subroutines is GL_ARB_shader_subroutine. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.

M.3.84 Tessellation Shaders

The name string for tessellation shaders is GL_ARB_tessellation_shader. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.

M.3.85 RGB32 Texture Buffer Objects

The name string for RGB32 texture buffer objects is GL_ARB_texture_buffer_object_rgb32. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.

M.3.86 Transform Feedback 2

The name string for transform feedback 2 is GL_ARB_transform_feedback2. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.

M.3.87 Transform Feedback 3

The name string for transform feedback 3 is GL_ARB_transform_feedback3. This extension is equivalent to new core functionality introduced in OpenGL 4.0 and is provided to enable this functionality in older drivers.

M.3.88 OpenGL ES 2.0 Compatibility

The name string for OpenGL ES 2.0 compatibility is GL_ARB_ES2_compatibility. This extension is equivalent to new core functionality introduced in OpenGL 4.1 and is provided to enable this functionality in older drivers.

M.3.89 Program Binary Support

The name string for program binary support is GL_ARB_get_program_binary. This extension is equivalent to new core functionality introduced in OpenGL 4.1 and is provided to enable this functionality in older drivers.
M.3.90 Separate Shader Objects

The name string for separate shader objects is GL_ARB_separate_shader_objects. This extension is equivalent to new core functionality introduced in OpenGL 4.1 and is provided to enable this functionality in older drivers.

M.3.91 Shader Precision Restrictions

The name string for shader precision restrictions is GL_ARB_shader_precision. This extension is equivalent to new core functionality introduced in OpenGL 4.1 and is provided to enable this functionality in older drivers.

M.3.92 Double Precision Vertex Shader Inputs

The name string for double precision vertex shader inputs is GL_ARB_vertex_attrib_64bit. This extension is equivalent to new core functionality introduced in OpenGL 4.1 and is provided to enable this functionality in older drivers.

M.3.93 Viewport Arrays

The name string for viewport arrays is GL_ARB_viewport_array. This extension is equivalent to new core functionality introduced in OpenGL 4.1 and is provided to enable this functionality in older drivers.

M.3.94 Robust Context Creation

Robust context creation allows creating an OpenGL context supporting robust buffer access behavior and a specified graphics reset notification behavior exposed through the GL_ARB_robustness extension (see section M.3.97).

The name strings for GLX and WGL robust context creation are GLX_ARB_create_context_robustness and WGL_ARB_create_context_robustness, respectively.

M.3.95 OpenCL Event Sharing

OpenCL event sharing allows creating OpenGL sync objects linked to OpenCL event objects, potentially improving efficiency of sharing images and buffers between the two APIs.

The name string for OpenCL event sharing is GL_ARB_cl_event.
M.3.96 Debug Output Notification

Debug output notification enables GL to inform the application when various events occur that may be useful during development and debugging.

The name string for debug output notification is GL_ARB_debug_output.

M.3.97 Context Robustness

Context robustness provides “safe” APIs that limit data written to application memory to a specified length, provides a mechanism to learn about graphics resets affecting the context, and defines guarantee that out-of-bounds buffer object accesses will have deterministic behavior precluding instability or termination. Some of these behaviors are controlled at context creation time via the companion GLX_ARB_create_context_robustness or WGL_ARB_create_context_robustness extensions (see section M.3.94).

The name string for context robustness is GL_ARB_robustness.

M.3.98 Shader Stencil Export

Sharder stencil export enables shaders to generate a stencil reference value, allowing stencil testing to be performed against per-shader-invocation values.

The name string for shader stencil export is GL_ARB_shader_stencil_export.

M.3.99 Base Instanced Rendering

The name string for base instanced rendering is GL_ARB_base_instance. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

M.3.100 OpenGL Shading Language 4.20 Feature Pack

The name string for the OpenGL Shading Language 4.20 feature pack is GL_ARB_shading_language_420pack. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

M.3.101 Instanced Transform Feedback

The name string for instanced transform feedback is GL_ARB_transform_feedback_instanced. This extension is equivalent to new core functionality
introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

M.3.102 Compressed Texture Pixel Storage

The name string for compressed texture pixel storage is GL_ARB_compressed(texture_pixel_storage). This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

M.3.103 Conservative Depth

The name string for conservative depth is GL_ARB_conservative_depth. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

M.3.104 Internal Format Query

The name string for internal format query is GL_ARB_internalformat_query. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

M.3.105 Map Buffer Alignment

The name string for map buffer alignment is GL_ARB_map_buffer_alignment. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

M.3.106 Shader Atomic Counters

The name string for shader atomic counters is GL_ARB_shader_atomic_counters. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

M.3.107 Shader Image Load/Store

The name string for shader image load/store is GL_ARB_shader_image_load_store. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.
M.3. ARB EXTENSIONS

M.3.108 Shading Language Packing

The name string for shading language packing is GL_ARB_shading_language_packing. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.

M.3.109 Texture Storage

The name string for texture storage is GL_ARB_texture_storage. This extension is equivalent to new core functionality introduced in OpenGL 4.2 and is provided to enable this functionality in older drivers.
Index

*BaseVertex, 34
*GetString, 362
*GetStringi, 362
*MapBuffer, 51
*MapBufferRange, 48
*Pointer, 31
*WaitSync, 365

Accum, 477
ACCUM_*_BITS, 477
ACCUM_BUFFER_BIT, 477
ACTIVE_ATOMIC_COUNTER_-BUFFERS, 371
ACTIVE_ATTRIBUTE_MAX_-LENGTH, 75, 370, 412
ACTIVE_ATTRIBUTES, 75, 370, 412
ACTIVE_PROGRAM, 372, 410, 507
ACTIVE_SUBROUTINE_MAX_-LENGTH, 102, 377, 417
ACTIVE_SUBROUTINE_UNIFORM_-LOCATIONS, 100, 103, 376, 377, 417
ACTIVE_SUBROUTINE_UNIFORM_MAX_LENGTH, 102, 377, 417
ACTIVE_SUBROUTINE_UNIFORMS, 101, 377, 417
ACTIVE_SUBROUTINES, 101–103, 377, 417
ACTIVE_TEXTURE, 209, 211, 356, 398
ACTIVE_UNIFORM_BLOCK_-MAX_NAME_LENGTH, 370, 414
ACTIVE_UNIFORM_BLOCKS, 80, 81, 370, 414
ACTIVE_UNIFORM_-MAX_LENGTH, 84, 85, 370, 412
ACTIVE_UNIFORMS, 84, 85, 370, 412
ACTIVE_ATOMIC_COUNTER_-BUFFERS, 371
ActiveShaderProgram, 67, 91
ActiveTexture, 103, 209
ALIASED_LINE_WIDTH_RANGE, 427
ALL_ATTRIB_BITS, 478
ALL_BARRIER_BITS, 121
ALL_SHADER_BITS, 67
ALPHA, 244, 260, 277, 300, 395, 396, 404, 476, 516
ALPHA_BITS, 477
ALPHA_TEST, 477
AlphaFunc, 477
ALREADY_SIGNED, 351
ALWAYS, 244, 266, 293, 294, 399
AND, 304
AND_INVERTED, 304
AND_REVERSE, 304
Antialiasing, 187
ANY_SAMPLES_PASSED, 158, 160, 161, 295, 363, 364
ARB_base_instance, 511, 520
INDEX

ARB_compressed_texture_pixel_storage, 510, 521
ARB_conservative_depth, 511, 520
ARB_internalformat_query, 511, 520
ARB_map_buffer_alignment_query, 511, 521
ARB_shader_atomic_counters, 510, 520
ARB_shader_image_load_store, 511, 520, 521
ARB_shading_language_420pack, 511, 520
ARB_shading_language_packing, 521
ARB_texture_compression_bptc, 510, 520
ARB_texture_storage, 510, 520
ARB_transform_feedback_instanced, 511, 520
AreTexturesResident, 477
ARRAY_BUFFER, 31, 44, 54
ARRAY_BUFFER_BINDING, 54, 386
ArrayElement, 481
ATOMIC_COUNTER.ARRAY.-.STRIDE, 99
ATOMIC_COUNTER.BARRIER_BIT, 121
ATOMIC-.COUNTER_BUFFER, 44, 45, 100
ATOMIC_COUNTER_BUFFER_ACTIVE_ATOMIC_COUNTER-INDICES, 83, 418
ATOMIC_COUNTER_BUFFER_ACTIVE_ATOMIC_COUNTERS, 82, 83, 418
ATOMIC_COUNTER-.BUFFER_BINDING, 82, 367, 418, 423
ATOMIC_COUNTER_BUFFER-.DATA_SIZE, 82, 100, 418
ATOMIC_COUNTER_BUFFER REFERENCED_BY_FRAGMENT_SHADER, 83, 418
ATOMIC_COUNTER_BUFFER REFERENCED_BY_GEOMETRY_SHADER, 83, 418
ATOMIC_COUNTER_BUFFER REFERENCED_BY_TESS_CONTROL_SHADER, 83, 418
ATOMIC_COUNTER_BUFFER REFERENCED_BY_TESS_EVALUATION_SHADER, 418
ATOMIC_COUNTER_BUFFER REFERENCED_BY_VERTEX_SHADER, 83, 418
ATOMIC_COUNTER_BUFFER_SIZE, 368, 423
ATOMIC_COUNTER_BUFFER-.START, 368, 423
atomic_uint, 89, 99
atomicCounter, 448
atomicCounterDecrement, 448
atomicCounterIncrement, 448
ATTACHED_SHADERS, 370, 373, 411
AttachShader, 61
ATTRIB_STACK_DEPTH, 478
AUX, 477
BACK, 189, 293, 296, 306–310, 312, 316, 323, 391, 476
BACK_LEFT, 307, 377
BACK_RIGHT, 307, 377
barrier, 130
Begin, 474
BeginConditionalRender, 160, 161
BeginQuery, 158, 159, 295, 347, 348, 481, 493
BeginQueryIndexed, 158, 159, 168, 169
BeginTransformFeedback, 163–166, 515
BGR, 198, 317, 319
BGR_INTEGER, 198

OpenGL 4.2 (Core Profile) - April 27, 2012
INDEX

BGRA, 31, 32, 35, 198, 201, 206, 317
BGRA_INTEGER, 198, 201
BindAttribLocation, 72, 76
BindBuffer, 43–45, 54, 55, 242, 493
BindBufferBase, 45, 46, 98, 100, 164, 167, 367, 368, 517
BindBufferRange, 45, 46, 98, 100, 164–167, 467, 517
BindFragDataLocation, 72, 283
BindFragDataLocationIndexed, 282, 283, 299
BindFramebuffer, 324–326, 341
BindImageTexture, 267, 269, 467, 519
BindProgramPipeline, 64, 66–68, 103, 116, 166, 372
BindRenderbuffer, 324–326, 341
BindSampler, 211–213
BindTexture, 103, 209, 210, 261, 493
BindTransformFeedback, 162, 163
BindVertexArray, 56
BITMAP, 476
Bitmap, 476
BLEND, 296, 301, 303, 400
BLEND_COLOR, 400
BLEND_DST_ALPHA, 400
BLEND_DST_RGB, 400
BLEND_EQUATION_ALPHA, 400
BLEND_EQUATION_RGB, 400
BLEND_SRC_ALPHA, 400
BLEND_SRC_RGB, 400
BlendColor, 298, 301, 512
BlendEquation, 296
BlendEquationi, 296
BlendEquationSeparate, 296
BlendEquationSeparatei, 296
BlendFunc, 298, 299
BlendFunci, 298, 299
BlendFuncSeparate, 298, 299
BlendFuncSeparatei, 298, 299
BlitFramebuffer, 313, 319, 322, 323, 338, 494, 516
BLUE, 198, 244, 260, 277, 317, 319, 395, 396, 404
BLUE_BITS, 477
BLUE_INTEGER, 198
BOOL, 86
bool, 86, 95
BOOL_VEC2, 86
BOOL_VEC3, 86
BOOL_VEC4, 86
Buffer*Data, 120
BUFFER_ACCESS, 45, 48, 50, 387
BUFFER_ACCESS_FLAGS, 45, 48, 50, 52, 387, 482
BUFFER_MAP_LENGTH, 45, 48, 50, 52, 387, 482
BUFFER_MAP_OFFSET, 45, 48, 50, 52, 387, 482
BUFFER_MAP_POINTER, 45, 48, 50, 52, 366, 387
BUFFER_MAPPED, 45, 48, 50, 387
BUFFER_SIZE, 45, 48, 50, 51, 387
BUFFER_UPDATE_BARRIER_BIT, 120
BUFFER_USAGE, 45, 48, 49, 387
BufferData, 46, 47, 78, 469
BufferSubData, 47, 78, 119, 122, 468
bvec2, 86, 92
bvec3, 86
bvec4, 86
BYTE, 30, 33, 197, 274, 320, 321
CallList, 477
CallLists, 477
CCW, 188, 371, 391, 416
ccw, 133
centroid, 276
centroid in, 276
CHANGED_ITEMS, 1–3, 15, 17, 19,
INDEX


CheckFramebufferStatus, 341, 342

CLAMP, 476, 494

CLAMP_FRAGMENT_COLOR, 475

CLAMP_READ_COLOR, 318, 389

CLAMP_TO_BORDER, 244, 246, 251, 260, 322

CLAMP_TO_EDGE, 244, 246, 251, 260, 322

CLAMP_VERTEX_COLOR, 475

ClampColor, 318, 475

CLEAR, 304

Clear, 160, 175, 311, 313, 345, 477

ClearAccum, 477

ClearBuffer, 313

ClearBuffer*, 160, 175, 345, 481, 482

ClearBuffer{if ui}v, 312, 313

ClearBufferfi, 312, 313

ClearBufferfv, 312, 313

ClearBufferiv, 312, 313

ClearBufferuiv, 312

ClearColor, 311, 312, 512

ClearDepth, 311–313

ClearDepthf, 311

ClearStencil, 311–313

CLIENT_ALL_ATTRIB_BITS, 478

CLIENT_ATTRIB_STACK_DEPTH, 478

ClientActiveTexture, 474

ClientWaitSync, 349–352, 467

CLIP_DISTANCEi, 171, 388, 481

CLIP_DISTANCE0, 171

CLIP_PLANEi, 481

ClipPlane, 475

coherent, 121

COLOR, 227, 312, 313

Color*, 474


COLOR_ATTACHMENTm, 306, 308

COLOR_ATTACHMENTn, 326

COLOR_ATTACHMENT0, 306, 309, 316, 326

COLOR_CLEAR_VALUE, 401

COLOR_INDEX, 474

COLOR_LOGIC_OP, 303, 400

COLOR_MATERIAL, 475

COLOR_LOGIC_OP, 303, 400

COLOR_MATERIAL, 475

COLOR_SUM, 477

COLOR_WRITEMASK, 309, 310, 401

ColorMask, 309, 310

ColorMaski, 309

ColorMaterial, 475

ColorPointer, 474

COMMAND_BARRIER_BIT, 120

COMPARE_R_TO_TEXTURE, 481

COMPARE_REF_TO_TEXTURE, 244, 266, 481

COMPATIBLE_SUBROUTINES, 101, 417

Compile_STATUS, 59, 65, 369, 409

CompileShader, 59
INDEX

COMPRESSED_RED, 222
COMPRESSED_RED_RGTC1, 221, 222, 453, 454
COMPRESSED_RG, 222
COMPRESSED_RG_RGTC2, 217, 222, 454
COMPRESSED_RGB, 222
COMPRESSED_RGB_BPTC_SIGNED_FLOAT, 222, 455, 462
COMPRESSED_RGB_BPTC_UNSIGNED_FLOAT, 222, 455, 462
COMPRESSED_RGBA, 222
COMPRESSED_RGBA_BPTC_UNSIGNED_FLOAT, 222, 455–457
COMPRESSED_SIGNED_RED_RGTC1, 217, 222, 454, 455
COMPRESSED_SIGNED_RG_RGTC2, 217, 222, 455
COMPRESSED_SRGB, 222, 266
COMPRESSED_SRGB_ALPHA, 222, 266
COMPRESSED_SRGB_ALPHA_BPTC_UNORM, 222, 455–457
COMPRESSED_TEXTURE_FORMATS, 216, 428, 511
CompressedTexImage, 120
CompressedTexImage, 237
CompressedTexImage1D, 233
CompressedTexImage*, 265, 341, 487
CompressedTexImage*D, 194
CompressedTexImage1D, 232, 234–237
CompressedTexImage2D, 232, 234–237
CompressedTexImage3D, 233–237, 360
CompressedTexSubImage, 515
CompressedTexSubImage1D, 237
CompressedTexSubImage2D, 236–238
CompressedTexSubImage3D, 236–238
CONDITION_SATISFIED, 351
CONSTANT_ALPHA, 300
CONSTANT_COLOR, 300
CONTEXT_COMPATIBILITY_PROFILE_BIT, 363
CONTEXT_CORE_PROFILE_BIT, 363
CONTEXT_FLAG_FORWARD_COMPATIBLE_BIT, 362
CONTEXT_FLAGS, 362, 429
CONTEXT_PROFILE_MASK, 362, 363
COPY, 303, 304, 400
COPY_INVERTED, 304
COPY_READ_BUFFER, 512
COPY_READ_BUFFER_BINDING, 44, 53, 442, 512, 513
COPY_WRITE_BUFFER, 512
COPY_WRITE_BUFFER_BINDING, 44, 53, 442, 512, 513
CopyBufferSubData, 53, 120
CopyPixels, 227, 477, 515
CopyTex*, 120
CopyTexImage, 343, 477
CopyTexImage*, 265, 332, 337, 338, 341
CopyTexImage1D, 228, 229, 232, 255
CopyTexImage2D, 227–229, 232, 255, 515
CopyTexImage3D, 229
CopyTexImageSubImage, 343
CopyTexImageSubImage*, 232, 238, 332
CopyTexImageSubImage1D, 229, 231, 232
CopyTexImageSubImage2D, 228–232
CopyTexImageSubImage3D, 228–232

OpenGL 4.2 (Core Profile) - April 27, 2012
INDEX

CreateProgram, 61
CreateShader, 58, 59
CreateShaderProgramv, 64, 65, 513
CULL_FACE, 189, 391
CULL_FACE_MODE, 391
CullFace, 188, 189, 192
CURRENT_PROGRAM, 411, 507
CURRENT_QUERY, 363, 442, 517
CURRENT_VERTEX_ATTRIB, 375, 419, 518
CW, 188, 371
cw, 133

DECR, 293
DECR_WRAP, 293
DELETE_STATUS, 59, 369, 370, 409, 411
DeleteBuffers, 43–45, 467, 513
DeleteFramebuffers, 325, 326, 513
DeleteLists, 477
DeleteProgram, 64
DeleteProgramPipelines, 66–68, 116, 372, 513
DeleteQueries, 158, 159, 348, 513
DeleteRenderbuffers, 328, 329, 341, 467, 513
DeleteSamplers, 212, 213, 513
DeleteShader, 59
DeleteSync, 350, 365
DeleteTextures, 210, 269, 341, 467, 513
DeleteTransformFeedbacks, 162, 163, 513, 518
DeleteVertexArrays, 56, 513
DEPTH, 227, 312, 313, 377, 396, 404, 481
DEPTH24_STENCIL8, 217, 221
DEPTH32F_STENCIL8, 217, 221
DEPTH_ATTACHMENT, 326, 332, 339, 481, 482
DEPTH_BITS, 477
DEPTH_BUFFER, 482
DEPTH_BUFFER_BIT, 311, 313, 321, 323
DEPTH_CLAMP, 171, 388
DEPTH_CLEAR_VALUE, 401
DEPTH_COMPONENT16, 217, 221
DEPTH_COMPONENT24, 217, 221
DEPTH_COMPONENT32, 221
DEPTH_COMPONENT32F, 217, 221
DEPTH_FUNC, 399
DEPTH_RANGE, 388
DEPTH_STENCIL_ATTACHMENT, 331, 332, 336, 337
DEPTH_TEST, 294, 399
DEPTH_TEXTURE_MODE, 476
DEPTH_WRITEMASK, 401
DepthFunc, 294
DepthMask, 310
DepthRange, 155, 355, 356
DepthRangeArrayv, 155
DepthRangef, 155
DepthRangeIndexed, 155
DetachShader, 61
dFdx, 353
dFdy, 353
DisableClientState, 474
Disablei, 289, 296
DisableVertexAttribArray, 33, 375
DITHER, 303, 400

OpenGL 4.2 (Core Profile) - April 27, 2012
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
<th>Related Functions/Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>dmat*</td>
<td>518, 519</td>
<td>DrawArrays, DrawBuffers</td>
</tr>
<tr>
<td>dmat2, dmat2x3, dmat3x2, dmat3x4, dmat4, dmat4x2, dmat4x3</td>
<td>73, 74, 86, 74, 75, 87, 74, 75, 86, 93, 74, 75, 87, 74, 75, 86, 74, 75, 87, 74, 75, 86, 74, 75, 87</td>
<td></td>
</tr>
<tr>
<td>DONT_CARE</td>
<td>353, 425</td>
<td>DrawArraysIndirect, DrawArraysInstanced, DrawArraysInstancedBaseInstance, DrawArraysInstancedBaseInstance, DrawArraysOneInstance</td>
</tr>
<tr>
<td>double</td>
<td>70, 74, 86, 95, 155</td>
<td></td>
</tr>
<tr>
<td>DOUBLE_MAT2</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>DOUBLE_MAT2x3, DOUBLE_MAT2x4</td>
<td>87, 87</td>
<td></td>
</tr>
<tr>
<td>DOUBLE_MAT3</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>DOUBLE_MAT3x2, DOUBLE_MAT3x4</td>
<td>87, 87</td>
<td></td>
</tr>
<tr>
<td>DOUBLE_MAT4</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>DOUBLE_MAT4x2, DOUBLE_MAT4x3</td>
<td>87, 87</td>
<td></td>
</tr>
<tr>
<td>DOUBLE_VEC2</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>DOUBLE_VEC3</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>DOUBLE_VEC4</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>DOUBLEBUFFER</td>
<td>441</td>
<td></td>
</tr>
<tr>
<td>DRAW_BUFFER</td>
<td>306, 309, 316</td>
<td></td>
</tr>
<tr>
<td>DRAW_BUFFER0, DRAW_BUFFERi</td>
<td>296, 309, 312, 403, 309</td>
<td></td>
</tr>
<tr>
<td>DRAW_BUFFERi</td>
<td>296, 299</td>
<td></td>
</tr>
<tr>
<td>DRAW_FRAMEBUFFER</td>
<td>324–326, 331–333, 342, 377, 402</td>
<td>DYNAMIC_COPY, DYNAMIC_DRAW, DYNAMIC_READ</td>
</tr>
<tr>
<td>DRAW_FRAMEBUFFER_BINDING</td>
<td>254, 305, 306, 322, 327, 342, 343, 402</td>
<td>DST_ALPHA, DST_COLOR</td>
</tr>
<tr>
<td>DRAW_INDIRECT_BUFFER</td>
<td>37, 42, 44, 55, 120</td>
<td>dst_alpha, dst_color, dvec2, dvec3, dvec4, dynamic_copy, dynamic_draw, dynamic_read</td>
</tr>
<tr>
<td>DRAW_INDIRECT_BUFFER_BINDING</td>
<td>386</td>
<td></td>
</tr>
<tr>
<td>Term</td>
<td>Page Numbers</td>
<td></td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>EdgeFlag*</td>
<td>474</td>
<td></td>
</tr>
<tr>
<td>EdgeFlagPointer</td>
<td>474</td>
<td></td>
</tr>
<tr>
<td>ELEMENT_ARRAY_BARRIER_BIT</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>ELEMENT_ARRAY_BUFFER, 44, 54, 120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELEMENT_ARRAY_BUFFER_BINDING,</td>
<td>385</td>
<td></td>
</tr>
<tr>
<td>EmitStreamVertex</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>EnableClientState</td>
<td>474</td>
<td></td>
</tr>
<tr>
<td>Enablei</td>
<td>289, 296</td>
<td></td>
</tr>
<tr>
<td>EnableVertexAttribArray</td>
<td>33, 56, 375</td>
<td></td>
</tr>
<tr>
<td>End</td>
<td>474</td>
<td></td>
</tr>
<tr>
<td>EndConditionalRender</td>
<td>160, 161</td>
<td></td>
</tr>
<tr>
<td>EndList</td>
<td>477</td>
<td></td>
</tr>
<tr>
<td>EndPrimitive</td>
<td>149, 345</td>
<td></td>
</tr>
<tr>
<td>EndQuery</td>
<td>158, 159, 295, 347, 348</td>
<td></td>
</tr>
<tr>
<td>EndQueryIndexed</td>
<td>159</td>
<td></td>
</tr>
<tr>
<td>EndStreamPrimitive</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>EndTransformFeedback</td>
<td>163, 166, 168, 470, 471, 514, 515</td>
<td></td>
</tr>
<tr>
<td>EQUAL,</td>
<td>139, 244, 266, 293, 294, 371, 416</td>
<td></td>
</tr>
<tr>
<td>equal_spacing</td>
<td>133</td>
<td></td>
</tr>
<tr>
<td>EQUIV,</td>
<td>304</td>
<td></td>
</tr>
<tr>
<td>EvalCoord*</td>
<td>477</td>
<td></td>
</tr>
<tr>
<td>EvalMesh*</td>
<td>477</td>
<td></td>
</tr>
<tr>
<td>EvalPoint*</td>
<td>477</td>
<td></td>
</tr>
<tr>
<td>EXTENSIONS</td>
<td>363, 429, 478, 523, 524</td>
<td></td>
</tr>
<tr>
<td>FASTEST</td>
<td>353</td>
<td></td>
</tr>
<tr>
<td>FeedbackBuffer</td>
<td>477</td>
<td></td>
</tr>
<tr>
<td>FenceSync</td>
<td>349, 350, 353, 469, 516</td>
<td></td>
</tr>
<tr>
<td>FILL,</td>
<td>191–193, 391, 450</td>
<td></td>
</tr>
<tr>
<td>Finish</td>
<td>348, 349, 450, 469</td>
<td></td>
</tr>
<tr>
<td>FIRST_VERTEX_CONVENTION</td>
<td>153, 169</td>
<td></td>
</tr>
<tr>
<td>FIXED,</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>FIXED_ONLY</td>
<td>318, 323, 389</td>
<td></td>
</tr>
<tr>
<td>flat,</td>
<td>148, 169</td>
<td></td>
</tr>
<tr>
<td>float,</td>
<td>70, 74, 86, 95</td>
<td></td>
</tr>
<tr>
<td>FLOAT_32_UNSIGNED_INT_24_8</td>
<td>24, 8, 195–197, 199, 201, 205, 315, 320, 321</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT2</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT2x3</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT2x4</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT3</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT3x2</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT3x4</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT4</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT4x2</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT4x3</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>FLOAT_VEC2</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>FLOAT_VEC3</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>FLOAT_VEC4</td>
<td>86</td>
<td></td>
</tr>
<tr>
<td>Flush,</td>
<td>348, 352, 450</td>
<td></td>
</tr>
<tr>
<td>FlushMappedBufferRange</td>
<td>50–52, 469</td>
<td></td>
</tr>
<tr>
<td>FOG,</td>
<td>477</td>
<td></td>
</tr>
<tr>
<td>Fog,</td>
<td>477</td>
<td></td>
</tr>
<tr>
<td>FOG_HINT</td>
<td>478</td>
<td></td>
</tr>
<tr>
<td>FogCoord*</td>
<td>474</td>
<td></td>
</tr>
<tr>
<td>FogCoordPointer</td>
<td>474</td>
<td></td>
</tr>
<tr>
<td>FRACTIONAL_EVEN</td>
<td>371</td>
<td></td>
</tr>
<tr>
<td>fractional_even_spacing</td>
<td>133</td>
<td></td>
</tr>
</tbody>
</table>
INDEX

551

INDEX

FRACTIONAL_ODD, 371
fractional_odd_spacing, 133
FRAGMENT_INTERPOLATION_OFFSET_BITS, 276, 438
FRAGMENT_SHADER, 275, 369, 372, 374, 410
FRAGMENT_SHADER_BIT, 67
FRAGMENT_SHADER_DERIVATIVE_HINT, 353, 425
FRAMEBUFFER, 325, 331–333, 342, 377
FRAMEBUFFER_ATTACHMENT_x_SIZE, 404
FRAMEBUFFER_ATTACHMENT_ALPHA_SIZE, 378
FRAMEBUFFER_ATTACHMENT_BLUE_SIZE, 378
FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING, 297, 302, 303, 378, 404
FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE, 378, 404
FRAMEBUFFER_ATTACHMENT_DEPTH_SIZE, 378
FRAMEBUFFER_ATTACHMENT_GREEN_SIZE, 378
FRAMEBUFFER_ATTACHMENT_LAYERED, 335, 379, 404
FRAMEBUFFER_ATTACHMENT_OBJECT_NAME, 331, 335, 339, 378, 379, 404
FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE, 331, 335, 339, 343, 378, 379, 404
FRAMEBUFFER_ATTACHMENT_RED_SIZE, 378
FRAMEBUFFER_ATTACHMENT_STENCIL_SIZE, 378
FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE, 335, 379, 404, 488
FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER, 335, 339, 344, 379, 404
FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL, 254, 335, 379, 404
FRAMEBUFFER_BARRIER_BIT, 121
FRAMEBUFFER_BINDING, 327
FRAMEBUFFER_COMPLETE, 342
FRAMEBUFFER_DEFAULT, 378
FRAMEBUFFER_INCOMPLETE_ATTACHMENT, 340
FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS, 341
FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT, 340
FRAMEBUFFER_INCOMPLETE_MULTISAMPLE, 340
FramebufferRenderbuffer, 331, 332, 335, 467
FramebufferTexture, 332, 335, 467
FramebufferTexture*, 335, 336, 441
FramebufferTexture1D, 333, 334
FramebufferTexture2D, 333–335
FramebufferTexture3D, 333–335
FramebufferTextureLayer, 334, 335, 488
FRONT, 189, 293, 296, 306–310, 312, 316, 323, 476

OpenGL 4.2 (Core Profile) - April 27, 2012
INDEX

FRONT_AND_BACK, 189, 191, 293, 296, 307–310, 312, 316
FRONT_FACE, 391
FRONT_LEFT, 307, 377
FRONT_RIGHT, 307, 377
FrontFace, 188, 280, 475
Frustum, 474
FUNC_ADD, 296, 298, 301, 400
FUNC_REVERSE_SUBTRACT, 296, 298
FUNC_SUBTRACT, 296, 298
fwidth, 353
Gen*, 467, 474
GenBuffers, 43, 44
GENERATE_MIPMAP, 477
GENERATE_MIPMAP_HINT, 478
GenerateMipmap, 256
GenFramebuffers, 324–327, 493, 513
GenLists, 477
GenProgramPipelines, 66–68, 116, 372
GenQueries, 158, 159, 348
GenRenderbuffers, 328, 329, 493
GenSamplers, 211–213, 361
GenTextures, 209, 210, 361, 517
GenTransformFeedbacks, 162, 163
GenVertexArrays, 55, 56, 493
GEOMETRY_INPUT_TYPE, 146, 370, 371, 413
GEOMETRY_OUTPUT_TYPE, 147, 371, 413
GEOMETRY_SHADER, 145, 369, 372, 410
GEOMETRY_SHADER_BIT, 67
GEOMETRY_SHADER_INVOCATIONS, 371, 413
GEOMETRY_VERTICES_OUT, 147, 151, 370, 371, 413
GEQUAL, 244, 266, 293, 294
Get, 354, 355
GetActiveSubroutineUniformName, 417
GetActiveSubroutineName, 417
GetActiveSubroutineUniformiv, 417
GetActiveUniformBlockiv, 415
GetActiveAtomicCounterBufferiv, 418
GetActiveAtomicCounterBufferv, 82, 91, 513, 517
GetActiveAttrib, 75, 76, 412, 513
GetActiveSubroutineName, 102
GetActiveSubroutineUniformiv, 101
GetActiveSubroutineUniformName, 101, 102
GetActiveUniform, 79, 85, 86, 89, 93, 412, 513
GetActiveUniformBlockiv, 80, 513, 514
GetActiveUniformBlockName, 80, 513
GetActiveUniformName, 84, 513
GetActiveUniformsiv, 85, 89, 90, 414, 415, 418, 513
GetAttachedShaders, 372, 411
GetAttribLocation, 76, 412, 512
GetBooleani_v, 309, 354, 401, 421
GetBooleanv, 292, 310, 354, 355, 381, 392, 401, 407, 408, 422, 428, 441
GetBufferParameteri64v, 366, 387
GetBufferParameteriv, 366, 387, 482
GetBufferPointerv, 366, 387, 517
GetBufferSubData, 120, 366, 387
GetCompressedTexImage, 235–238, 315, 353, 358, 360
GetDoublei_v, 355, 388
GetDoublev, 354, 355, 381
GetError, 19, 442
GetFloati_v, 355, 388
GetFloatv, 14, 157, 179, 292, 354, 355, 381, 383, 390–392, 400, 401, 426, 427, 438
GetFragDataIndex, 283, 284

OpenGL 4.2 (Core Profile) - April 27, 2012
INDEX

GetFragDataLocation, 283, 284
GetFramebufferAttachment-Parameteriv, 404
GetFramebufferAttachmentiv, 482
GetFramebufferAttachmentParameteriv, 343, 377, 378, 481, 482
GetInteger, 301, 432
GetInteger64i_v, 355, 367, 368, 414, 422, 423
GetInteger64v, 348, 351, 354, 355, 381, 437, 517
GetIntegeri_v, 292, 301, 354, 367, 392, 399, 400, 414, 421–423
GetInternalformativ, 240, 329, 380, 439
GetMultisamplefv, 111, 178, 441
GetProgramStageiv, 417
GetProgramBinary, 71–73, 411
GetProgramInfoLog, 372, 373, 410
GetProgramPipelineInfoLog, 373
GetProgramPipelineiv, 116, 372, 373, 410
GetProgramPipelineInfoLog, 410
GetProgramStageiv, 102, 377
GetQueryIndexediv, 364
GetQueryiv, 363, 437, 442
GetQueryObject*, 365
GetQueryObjecti64v, 364
GetQueryObjectiv, 364, 420
GetQueryObjectui64v, 364
GetQueryObjectuiv, 364, 420
GetRenderbufferParameteriv, 406
GetRenderbufferParameteriv, 343, 380, 483
GetSamplerParameter, 361, 397
GetSamplerParameter*, 211, 361
GetSamplerParameterfv, 397
GetSamplerParameterI{i ui}v, 361
GetSamplerParameterfv, 361
GetSamplerParameterIuiv, 361
GetSamplerParameteriv, 397
GetShaderInfoLog, 59, 373, 409
GetShaderiv, 59, 369, 373, 374, 409
GetShaderPrecisionFormat, 59, 374
GetShaderSource, 373, 409
GetSynciv, 349, 365, 424
GetTexImage, 120, 261, 271, 273, 315, 358–360, 394, 488
GetTexLevelParameter, 356, 357, 396
GetTexParameter, 261, 273, 343, 356, 395
GetTexParameterfv, 261, 395
GetTexParameterI, 356
GetTexParameterIuiv, 356
GetTexParameteriv, 261, 395
GetTransformFeedbackVarying, 413
GetUniform, 412
GetUniform*, 376
GetUniformBlockIndex, 79, 80, 513
GetUniformfv, 376
GetUniformIndices, 83, 84, 513, 514

OpenGL 4.2 (Core Profile) - April 27, 2012
<table>
<thead>
<tr>
<th>Function/Extension</th>
<th>Index 1</th>
<th>Index 2</th>
<th>Index 3</th>
<th>Index 4</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_ES2_compatibility, 506, 508, 538</td>
</tr>
<tr>
<td>GetUniformLocation</td>
<td>79, 100, 103, 104, 412</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_explicit_attrib_location, 498, 499, 536</td>
</tr>
<tr>
<td>GetUniformSubroutineiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_fragment_coord_conventions, 492, 533</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_fragment_program, 527, 528</td>
</tr>
<tr>
<td>GetUniformLocation</td>
<td>79, 100, 103, 104, 412</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_fragment_program_shadow, 528</td>
</tr>
<tr>
<td>GetUniformSubroutineiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_framebuffer_object, 530</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_framebuffer_sRGB, 530</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_geometry_shader4, 531</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_geometry_shader4, 492</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_get_program_binary, 506, 508, 509, 538</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_gu_shader5, 502, 504, 537</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_gu_shader_fp64, 502, 504, 504, 537</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_half_float_pixel, 480, 529</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_half_float_vertex, 531</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_instanced_arrays, 498–500, 531</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_internalformat_query, 541</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_map_buffer_alignment, 541</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_map_buffer_range, 531</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_matrix_palette, 525</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_multisample, 524</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_multitexture, 524</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_occlusion_query, 527</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_occlusion_query2, 498, 499, 536</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_pixel_buffer_object, 530</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_point_parameters, 525</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_point_sprite, 528</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_provoking_vertex, 533</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARB_provoking_vertex, 492</td>
</tr>
<tr>
<td>GetUniformiv</td>
<td>376</td>
<td></td>
<td></td>
<td></td>
<td>GL_ARBrobustness, 509, 539, 540</td>
</tr>
</tbody>
</table>

OpenGL 4.2 (Core Profile) - April 27, 2012
<table>
<thead>
<tr>
<th>Function/Feature</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_ARB_sample_shading</td>
<td>502, 504, 534</td>
</tr>
<tr>
<td>GL_ARB_sampler_objects</td>
<td>498, 499, 536</td>
</tr>
<tr>
<td>GL_ARB_seamless_cube_map</td>
<td>534</td>
</tr>
<tr>
<td>GL_ARB_seamless_cube_map</td>
<td>492</td>
</tr>
<tr>
<td>GL_ARB_separate_shader_objects</td>
<td>507–509, 539</td>
</tr>
<tr>
<td>GL_ARB_shader_atomic_counters</td>
<td>541</td>
</tr>
<tr>
<td>GL_ARB_shader_bit_encoding</td>
<td>497, 499, 536</td>
</tr>
<tr>
<td>GL_ARB_shader_image_load_store</td>
<td>541</td>
</tr>
<tr>
<td>GL_ARB_shader_objects</td>
<td>527</td>
</tr>
<tr>
<td>GL_ARB_shader_precision</td>
<td>507, 539</td>
</tr>
<tr>
<td>GL_ARB_shader_stencil_export</td>
<td>508, 540</td>
</tr>
<tr>
<td>GL_ARB_shader_subroutine</td>
<td>502, 504, 538</td>
</tr>
<tr>
<td>GL_ARB_shader_texture_lod</td>
<td>533</td>
</tr>
<tr>
<td>GL_ARB_shading_language_100</td>
<td>528</td>
</tr>
<tr>
<td>GL_ARB_shading_language_420pack</td>
<td>540</td>
</tr>
<tr>
<td>GL_ARB_shading_language_include</td>
<td>535</td>
</tr>
<tr>
<td>GL_ARB_shading_language_packing</td>
<td>542</td>
</tr>
<tr>
<td>GL_ARB_shadow</td>
<td>526, 528</td>
</tr>
<tr>
<td>GL_ARB_shadow_ambient</td>
<td>526</td>
</tr>
<tr>
<td>GL_ARB_sync</td>
<td>492, 534</td>
</tr>
<tr>
<td>GL_ARB_tessellation_shader</td>
<td>502, 504, 538</td>
</tr>
<tr>
<td>GL_ARB_texture_border_clamp</td>
<td>525</td>
</tr>
<tr>
<td>GL_ARB_texture_buffer_object</td>
<td>487, 531</td>
</tr>
<tr>
<td>GL_ARB_texture_buffer_object_rgb32</td>
<td>502, 504, 538</td>
</tr>
<tr>
<td>GL_ARB_texture_compression</td>
<td>525</td>
</tr>
<tr>
<td>GL_ARB_texture_compression_bptc</td>
<td>536</td>
</tr>
<tr>
<td>GL_ARB_texture_compression_rgtc</td>
<td>531</td>
</tr>
<tr>
<td>GL_ARB_texture_cube_map</td>
<td>525</td>
</tr>
<tr>
<td>GL_ARB_texture_cube_map_array</td>
<td>502, 504, 535</td>
</tr>
<tr>
<td>GL_ARB_texture_env_add</td>
<td>525</td>
</tr>
<tr>
<td>GL_ARB_texture_env_combine</td>
<td>526</td>
</tr>
<tr>
<td>GL_ARB_texture_env_crossbar</td>
<td>526</td>
</tr>
<tr>
<td>GL_ARB_texture_env_dot3</td>
<td>526</td>
</tr>
<tr>
<td>GL_ARB_texture_float</td>
<td>480, 529</td>
</tr>
<tr>
<td>GL_ARB_texture_gather</td>
<td>502–504, 535</td>
</tr>
<tr>
<td>GL_ARB_texture_mirrored_repeat</td>
<td>526</td>
</tr>
<tr>
<td>GL_ARB_texture_multisample</td>
<td>534</td>
</tr>
<tr>
<td>GL_ARB_texture_non_power_of_two</td>
<td>528</td>
</tr>
<tr>
<td>GL_ARB_texture_query_lod</td>
<td>501, 504, 535</td>
</tr>
<tr>
<td>GL_ARB_texture_rectangle</td>
<td>487, 529</td>
</tr>
<tr>
<td>GL_ARB_texture_rg</td>
<td>531</td>
</tr>
<tr>
<td>GL_ARB_texture_rgb10_a2ui</td>
<td>498, 499, 537</td>
</tr>
<tr>
<td>GL_ARB_texture_storage</td>
<td>542</td>
</tr>
<tr>
<td>GL_ARB_texture_swizzle</td>
<td>498, 500, 537</td>
</tr>
<tr>
<td>GL_ARB_texture_swizzle</td>
<td>537</td>
</tr>
<tr>
<td>GL_ARB_texture_multisample</td>
<td>492</td>
</tr>
<tr>
<td>GL_ARB_timer_query</td>
<td>498, 500, 537</td>
</tr>
<tr>
<td>GL_ARB_transform_feedback2</td>
<td>503, 504, 538</td>
</tr>
<tr>
<td>GL_ARB_transform_feedback3</td>
<td>503, 504, 538</td>
</tr>
<tr>
<td>GL_ARB_transform_feedback_instantced</td>
<td>540</td>
</tr>
<tr>
<td>GL_ARB_transpose_matrix</td>
<td>524</td>
</tr>
<tr>
<td>GL_ARB_uniform_buffer_object</td>
<td>487, 532</td>
</tr>
<tr>
<td>GL_ARB_vertex_array_bgra</td>
<td>492, 534</td>
</tr>
<tr>
<td>GL_ARB_vertex_array_object</td>
<td>532</td>
</tr>
<tr>
<td>GL_ARB_vertex_attrib_64bit</td>
<td>507–509, 539</td>
</tr>
<tr>
<td>GL_ARB_vertex_blend</td>
<td>525</td>
</tr>
</tbody>
</table>
GL_ARB_vertex_buffer_object, 527
GL_ARB_vertex_program, 527
GL_ARB_vertex_shader, 527
GL_ARB_vertex_type_2_10_10_10_rev, 498, 499, 537
GL_ARB_viewport_array, 507, 508, 539
GL_ARB_window_pos, 526
GL_ARB_name, 523
GL_ARB_draw_buffers_blend, 501
gl_BackColor, 475
gl_BackSecondaryColor, 475
gl_ClipDistance, 115, 127, 129, 142, 144, 152, 481
gl_ClipDistance[], 69, 150, 170
gl_ClipVertex, 127, 481
GL_EXT_draw_buffers2, 480
GL_EXT_framebuffer_blit, 480, 530
GL_EXT_framebuffer_multisample, 480, 530
GL_EXT_framebuffer_object, 480, 530
GL_EXT_framebuffer_sRGB, 480, 530
GL_EXT_geometry_shader4, 483
GL_EXT_gpu_shader4, 479
GL_EXT_packed_depthStencil, 480
GL_EXT_packed_float, 480
GL_EXT_texture_array, 480, 483
GL_EXT_texture_compression_rgtc, 480, 531
GL_EXT_texture_integer, 480
GL_EXT_texture_shared_exponent, 480
GL_EXT_transform_feedback, 480
gl_FragColor, 516
gl_FragCoord, 279
gl_FragCoord.z, 445
gl_FragData[n], 516
gl_FragDepth, 281, 282, 445
gl_FrontFacing, 279
gl_in, 127, 142
gl_in[], 150
gl_InstanceID, 35, 38, 114
gl_InvocationID, 126, 128, 129, 149
gl_Layer, 152, 153, 345, 427
gl_MaxPatchVertices, 127, 128, 142, 143
gl_NextBuffer, 166
gl_NumSamples, 281
GL_NV_conditional_render, 479
GL_NV_depth_buffer_float, 480, 530
GL_NV_half_float, 480, 531
GL_NV_primitive_restart, 486
gl_out, 129
gl_PatchVerticesIn, 128, 142
gl_PointCoord, 181
gl_PointSize, 115, 127, 129, 142, 144, 150, 152, 180
gl_Position, 105, 114, 127, 129, 142, 144, 150, 152, 154, 451
gl_PrimitiveID, 128, 142, 152, 280
gl_PrimitiveIDIn, 150
gl_SampleID, 281
gl_SampleMask, 281, 291
gl_SampleMaskIn, 280
gl_SamplePosition, 281
gl_SkipComponents, 166
gl_TessCoord, 132, 142, 448
gl_TessLevelInner, 129, 130, 143, 144
gl_TessLevelInner[1], 143
gl_TessLevelOuter, 129, 130, 143, 144
gl_TessLevelOuter[2], 143
gl_TessLevelOuter[3], 143
gl_VertexID, 114, 280
gl_VerticesOut, 129, 518
gl_ViewportIndex, 152–154, 427
gl_PerVertex, 70
GL_TRUE, 73
GLX_ARB_create_context, 532
GLX_ARB_create_context_profile, 473, 535
GLX_ARB_create_context_robustness, 539, 540
GLX_ARB_create_context_robustness, 509
GLX_ARB_fbconfig_float, 529
GLX_ARB_framebuffer_sRGB, 530
GREATER, 244, 266, 293, 294
GREEN, 198, 244, 260, 277, 317, 319, 395, 396, 404
GREEN_BITS, 477
GREEN_INTEGER, 198
HALF_FLOAT, 30, 197, 215, 273, 274, 317–320
HIGH_FLOAT, 374
HIGH_INT, 374
Hint, 353, 478
iimage1D, 89
iimage1DArray, 89
iimage2D, 89
iimage2DArray, 89
iimage2DMS, 89
iimage2DMSArray, 89
iimage2DRect, 89
iimage3D, 89
iimageBuffer, 89
iimageCube, 89
iimageCubeArray, 89
image1D, 88
image1DArray, 88
image2D, 88
image2DArray, 88
image2DMS, 88
image2DMSArray, 89
image2DRect, 88
image3D, 88
IMAGE_1D, 88
IMAGE_1D_ARRAY, 88
IMAGE_2D, 88
IMAGE_2D_ARRAY, 88
IMAGE_2D_MULTISAMPLE, 88
IMAGE_2D_MULTISAMPLE_ARRAY, 89
IMAGE_2D_RECT, 88
IMAGE_3D, 88
IMAGE_BINDING_ACCESS, 421
IMAGE_BINDING_FORMAT, 421
IMAGE_BINDING_LAYER, 421
IMAGE_BINDING_LAYERED, 421
IMAGE_BINDING_LEVEL, 421
IMAGE_BINDING_NAME, 421
IMAGE_BUFFER, 88
IMAGE_CUBE, 88
IMAGE_CUBE_MAP_ARRAY, 88
IMAGE_FORMAT_COMPATIBILITY_TYPE, 273, 356, 395
imageBuffer, 88
imageCube, 88
imageCubeArray, 88
IMPLEMENTATION_COLOR_READ_FORMAT, 314, 441
IMPLEMENTATION_COLOR_READ_TYPE, 315, 441
in, 143
INCR, 293
INCR_WRAP, 293
Index*, 474
IndexPointer, 474
INFO_LOG_LENGTH, 369, 370, 372, 373, 409–411
InitNames, 477
INT, 30, 33, 86, 197, 271, 274, 320, 321, 357, 378
int, 70, 74, 86, 95
INT_2_10_10_10_REV, 29–31, 34, 35
INT_IMAGE_1D, 89
INDEX 558

INT_IMAGE_1D_ARRAY, 89
INT_IMAGE_2D, 89
INT_IMAGE_2D_ARRAY, 89
INT_IMAGE_2D_MULTISAMPLE, 89
INT_IMAGE_2D_MULTISAMPLE_ARRAY, 89
INT_IMAGE_2D_RECT, 89
INT_IMAGE_3D, 89
INT_IMAGE_BUFFER, 89
INT_IMAGE_CUBE, 89
INT_IMAGE_CUBE_MAP_ARRAY, 89
INT_SAMPLER_1D, 87
INT_SAMPLER_1D_ARRAY, 87
INT_SAMPLER_2D, 87
INT_SAMPLER_2D_ARRAY, 87
INT_SAMPLER_2D_MULTISAMPLE, 88
INT_SAMPLER_2D_MULTISAMPLE_ARRAY, 88
INT_SAMPLER_2D_RECT, 88
INT_SAMPLER_3D, 87
INT_SAMPLER_BUFFER, 88
INT_SAMPLER_CUBE, 87
INT_SAMPLER_CUBE_MAP_ARRAY, 87
INT_VEC2, 86
INT_VEC3, 86
INT_VEC4, 86
INTENSITY, 476
INTERLEAVED_ATTRIBS, 106, 107, 124, 165, 370, 413
InterleavedArrays, 474
interpolateAtOffset, 438
interpolateAtCentroid, 276
interpolateAtOffset, 276
INVALID_ENUM, 381
INVARIANT_INDEX, 80, 84, 101
INVALID_FRAMEBUFFER_OPERATION, 20, 228, 232, 316, 322, 343
INVERT, 293, 304
isampler1D, 87
isampler1DArray, 87
isampler2D, 87
isampler2DArray, 87
OpenGL 4.2 (Core Profile) - April 27, 2012
INDEX

LoadTransposeMatrix, 474
LOGIC_OP_MODE, 400
LogicOp, 303, 304
LOW_FLOAT, 374
LOW_INT, 374
LOWER_LEFT, 181
LUMINANCE, 476
LUMINANCE ALPHA, 476
main, 131
MAJOR VERSION, 362, 429
Map*, 477
MAP_FLUSH_EXPLICIT_BIT, 49, 51, 52
MAP_INVALIDATE_BUFFER_BIT, 49, 51
MAP_INVALIDATE_RANGE_BIT, 49, 51
MAP_READ_BIT, 48–51
MAP_UNSYNCHRONIZED_BIT, 50, 51
MAP_WRITE_BIT, 49–51
MapBuffer, 45, 48, 51, 78, 167, 482, 513
MapBuffer*, 121
MapBufferRange, 48–51, 482, 513
MapGrid*, 477
matC, 95
matCxR, 95
mat2, 73, 86
mat2x3, 73, 86
mat2x4, 73, 86
mat3, 73, 86
mat3x2, 73, 86
mat3x4, 73, 86
mat4, 73, 86
mat4x2, 73, 86
mat4x3, 73, 86
Material*, 475
MatrixMode, 474
MAX, 296, 298
MAX_3D_TEXTURE_SIZE, 223, 333, 334, 426
MAX_ARRAY_TEXTURE_LAYERS,
  224, 334, 426
MAX_ATOMIC_COUNTER-
  BUFFER_BINDINGS, 100,
  368, 434
MAX_ATOMIC_COUNTER-
  BUFFER_SIZE, 82, 434, 520
MAX_ATOMIC_COUNTER-
  BUFFERS, 99
MAX_ATTRIB_STACK_DEPTH, 478
MAX_CLIENT_ATTRIB_STACK-
  DEPTH, 478
MAX_CLIP_DISTANCES, 426, 481, 487
MAX_CLIP_PLANES, 481
MAX_COLOR_ATTACHMENTS,
  306–308, 325, 332, 342, 438, 517
MAX_COLOR_TEXTURE_SAMPLES, 437
MAX_COMBINED_ATOMIC-
  COUNTER_BUFFERS, 434
MAX_COMBINED_ATOMIC_COUN-
  TERS, 113, 434
MAX_COMBINED_FRAGMENT-
  UNIFORM_COMPONENTS, 275, 436
MAX_COMBINED_GEOMETRY-
  UNIFORM_COMPONENTS, 147, 436
MAX_COMBINED_IMAGE_UNI-
  FORMS, 114, 435, 520
MAX_COMBINED_IMAGE_UNITS-
  AND_FRAGMENT_OUTPUTS, 275, 435
MAX_COMBINED_TESS_CONTROL_UNI-

OpenGL 4.2 (Core Profile) - April 27, 2012
INDEX

FORM_COMPONENTS, 126, 436
MAX_COMBINED_TESS_EVALUATION_UNIFORM_COMPONENTS, 141, 436
MAX_COMBINED_TEXTURE_IMAGE_UNITS, 112, 209, 212, 434
MAX_COMBINED_UNIFORM_BLOCKS, 94, 434, 494
MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS, 78, 436
MAX_CUBE_MAP_TEXTURE_SIZE, 224, 333, 426
MAX_DEPTH_TEXTURE_SAMPLES, 437
MAX_DRAW_BUFFERS, 283, 296, 297, 299, 308, 309, 313, 438, 517
MAX_DUAL_SOURCE_DRAW_BUFFERS, 283, 299, 438, 517
MAX_ELEMENTS_INDICES, 40, 428
MAX_ELEMENTS_VertICES, 40, 428
MAX_FRAGMENT_ATOMIC_COUNTER_BUFFERS, 99, 433
MAX_FRAGMENT_ATOMIC_COUNTERS, 113, 433
MAX_FRAGMENT_IMAGE_UNIFORMS, 114, 435, 520
MAX_FRAGMENT_UNIFORM_COMPONENTS, 275, 433
MAX_FRAGMENT_UNIFORM_VECTORS, 275, 433
MAX_GEOMETRY_ATOMIC_COUNTER_BUFFERS, 99, 432
MAX_GEOMETRY_ATOMIC_COUNTERS, 113, 432
MAX_GEOMETRY_IMAGE_UNIFORMS, 114, 435
MAX_GEOMETRY_INPUT_COMPONENTS, 151, 432
MAX_GEOMETRY_OUTPUT_COMPONENTS, 152, 432
MAX_GEOMETRY_OUTPUT_VERTICES, 151, 432
MAX_GEOMETRY_SHADER_INVOCATIONS, 432
MAX_GEOMETRY_TEXTURE_IMAGE_UNITS, 112, 432
MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS, 151, 432
MAX_GEOMETRY_UNIFORM_BLOCKS, 94, 432
MAX_GEOMETRY_UNIFORM_COMPONENTS, 147, 432
MAX_IMAGE_SAMPLES, 270, 435
MAX_IMAGE_UNITS, 104, 267, 268, 435
MAX_INTEGER_SAMPLES, 331, 437
MAX_PATCH_VERTICES, 27, 431
MAX_PROGRAM_TEXEL_OFFSET, 248, 434
MAX_PROGRAM_TEXTURE_GATHER_OFFSET, 248, 433

OpenGL 4.2 (Core Profile) - April 27, 2012
<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_RECTANGLE_TEXTURE_SIZE</td>
<td>224, 428</td>
</tr>
<tr>
<td>MAX_RENDERBUFFER_SIZE</td>
<td>329, 426</td>
</tr>
<tr>
<td>MAX_SAMPLE_MASK_WORDS</td>
<td>292, 437</td>
</tr>
<tr>
<td>MAX_SAMPLES</td>
<td>330, 437</td>
</tr>
<tr>
<td>MAX_SERVER_WAIT_TIMEOUT</td>
<td>351, 437</td>
</tr>
<tr>
<td>MAX_SUBROUTINE_UNIFORM_LOCATIONS</td>
<td>101, 434</td>
</tr>
<tr>
<td>MAX_SUBROUTINES</td>
<td>101, 434</td>
</tr>
<tr>
<td>MAX_TESS_CONTROL_ATOMIC_COUNTER_BUFFERS</td>
<td>99, 431</td>
</tr>
<tr>
<td>MAX_TESS_CONTROL_ATOMIC_COUNTERS</td>
<td>113, 431</td>
</tr>
<tr>
<td>MAX_TESS_CONTROL_IMAGE_UNIFORMS</td>
<td>114, 435</td>
</tr>
<tr>
<td>MAX_TESS_CONTROL_INPUT_COMPONENTS</td>
<td>128, 431</td>
</tr>
<tr>
<td>MAX_TESS_CONTROL_OUTPUT_COMPONENTS</td>
<td>130, 431</td>
</tr>
<tr>
<td>MAX_TESS_CONTROL_TEXTURE_IMAGE_UNITS</td>
<td>112, 279, 433</td>
</tr>
<tr>
<td>MAX_TESS_CONTROL_UNIFORM_BLOCKS</td>
<td>94, 431</td>
</tr>
<tr>
<td>MAX_TESS_CONTROL_UNIFORM_COMPONENTS</td>
<td>126, 431</td>
</tr>
<tr>
<td>MAX_TESS_EVALUATION_ATOMIC_COUNTER_BUFFERS</td>
<td>99, 431</td>
</tr>
<tr>
<td>MAX_TESS_EVALUATION_ATOMIC_COUNTERS</td>
<td>113, 431</td>
</tr>
<tr>
<td>MAX_TESS_EVALUATION_IMAGE_UNIFORMS</td>
<td>114, 435</td>
</tr>
<tr>
<td>MAX_TESS_EVALUATION_INPUT_COMPONENTS</td>
<td>144, 431</td>
</tr>
<tr>
<td>MAX_TESS_EVALUATION_OUTPUT_COMPONENTS</td>
<td>144, 431</td>
</tr>
<tr>
<td>MAX_TESS_EVALUATION_TEXTURE_IMAGE_UNITS</td>
<td>112, 431</td>
</tr>
<tr>
<td>MAX_TESS_EVALUATION_UNIFORM_BLOCKS</td>
<td>94, 431</td>
</tr>
<tr>
<td>MAX_TESS_EVALUATION_UNIFORM_COMPONENTS</td>
<td>141, 431</td>
</tr>
<tr>
<td>MAX_TESS_GEN_LEVEL</td>
<td>133, 431</td>
</tr>
<tr>
<td>MAX_TESS_PATCH_COMPONENTS</td>
<td>130, 144, 431</td>
</tr>
<tr>
<td>MAX_TEXTURE_BUFFER_SIZE</td>
<td>241, 428</td>
</tr>
<tr>
<td>MAX_TEXTURE_COORDS</td>
<td>477</td>
</tr>
<tr>
<td>MAX_TEXTURE_IMAGE_UNITS</td>
<td>112, 279, 433</td>
</tr>
<tr>
<td>MAX_TEXTURE_LOD_BIAS</td>
<td>248, 426</td>
</tr>
<tr>
<td>MAX_TEXTURE_SIZE</td>
<td>224, 239, 334, 426</td>
</tr>
<tr>
<td>MAX_TEXTURE_UNITS</td>
<td>477</td>
</tr>
<tr>
<td>MAX_TRANSFORM_FEEDBACK_BUFFERS</td>
<td>107, 164, 367, 440</td>
</tr>
<tr>
<td>MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS</td>
<td>107, 440</td>
</tr>
<tr>
<td>MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS</td>
<td>106, 165, 440</td>
</tr>
<tr>
<td>MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS</td>
<td>107, 440</td>
</tr>
<tr>
<td>MAX_UNIFORM_BLOCK_SIZE, 434</td>
<td>MAX_SAMPLES, 381</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>MAX_UNIFORM_BUFFER_BINDINGS, 494</td>
<td>MEDIUM_FLOAT, 374</td>
</tr>
<tr>
<td>MAX_VARYING_COMPONENTS, 492</td>
<td>MEDIUM_INT, 374</td>
</tr>
<tr>
<td>MAX_VARYING_FLOATS, 492</td>
<td>MemoryBarrier, 119, 121, 122</td>
</tr>
<tr>
<td>MAX_VARYING_VECTORS, 434</td>
<td>memoryBarrier(), 118, 122</td>
</tr>
<tr>
<td>MAX_VERTEX_ATOMIC_COUNTER BUFFERS, 99, 430</td>
<td>MIN, 296, 298</td>
</tr>
<tr>
<td>MAX_VERTEX_ATOMIC_COUNTERS, 430</td>
<td>MIN_FRAGMENT_INTERPOLATION_OFFSET, 276, 438</td>
</tr>
<tr>
<td>MAX_VERTEX_ATTRIBS, 28–31, 33, 43, 75, 76, 375, 376, 430</td>
<td>MIN_MAP_BUFFER_ALIGNMENT, 49, 51, 428, 519</td>
</tr>
<tr>
<td>MAX_VERTEX_IMAGE_UNIFORMS, 435</td>
<td>MIN_PROGRAM_TEXEL_OFFSET, 248, 434</td>
</tr>
<tr>
<td>MAX_VERTEX_OUTPUT_COMPONENTS, 105, 128, 130, 144, 151, 152, 281, 430</td>
<td>MIN_PROGRAM_TEXTURE_-GATHER_OFFSET, 248, 433</td>
</tr>
<tr>
<td>MAX_VERTEX_STREAMS, 168, 169, 432</td>
<td>MIN_SAMPLE_SHADING_VALUE, 179, 392</td>
</tr>
<tr>
<td>MAX_VERTEX_TEXTURE_IMAGE_UNITS, 111, 430</td>
<td>MINOR_VERSION, 362, 429</td>
</tr>
<tr>
<td>MAX_VERTEX_UNIFORM_BLOCKS, 94, 430</td>
<td>MinSampleShading, 179, 512</td>
</tr>
<tr>
<td>MAX_VERTEX_UNIFORM_COMPONENTS, 78, 430</td>
<td>MIRRORED_REPEAT, 212, 244, 245, 251</td>
</tr>
<tr>
<td>MAX_VERTEX_UNIFORM_VECTORS, 78, 430</td>
<td>MultiDrawArrays, 38</td>
</tr>
<tr>
<td>MAX_VIEWPORT_DIMS, 157, 364, 427</td>
<td>MultiDrawElements, 40, 54, 513</td>
</tr>
<tr>
<td>MAX_VIEWPORTS, 154–156, 289, 290, 427</td>
<td>MultiDrawElementsBaseVertex, 42, 54, 513</td>
</tr>
<tr>
<td>MAX_COLOR_TEXTURE_SAMPLES, 381</td>
<td>MULTISAMPLE, 178, 179, 182, 187, 192, 276, 290, 304, 392</td>
</tr>
<tr>
<td>MAX_DEPTH_TEXTURE_SAMPLES, 381</td>
<td>MultMatrix, 474</td>
</tr>
<tr>
<td>MAX_INTEGER_SAMPLES, 381</td>
<td>MultTransposeMatrix, 474</td>
</tr>
<tr>
<td></td>
<td>NAND, 304</td>
</tr>
<tr>
<td></td>
<td>NEAREST, 110, 111, 244–246, 250, 252, 254, 256–258, 266, 321, 337</td>
</tr>
<tr>
<td></td>
<td>NEAREST_MIPMAP_-LINEAR, 244, 254, 256, 257, 260, 337</td>
</tr>
<tr>
<td></td>
<td>NEAREST_MIPMAP_NEAREST, 244, 254, 255, 257, 258, 266, 337</td>
</tr>
<tr>
<td></td>
<td>NEVER, 244, 266, 293, 294</td>
</tr>
<tr>
<td></td>
<td>NewList, 477</td>
</tr>
</tbody>
</table>
NICEST, 353
NO_ERROR, 19
NOOP, 304
noperspective, 172
NOR, 304
Normal3*, 474
NORMALIZE, 475
NormalPointer, 474
NOTEQUAL, 244, 266, 293, 294
NUM_COMPATIBLE_SUBROUTINES, 101, 102, 417
NUM_COMPRESSED_TEXTURE_FORMATS, 216, 428, 511
NUM_EXTENSIONS, 363, 429
NUM_PROGRAM_BINARY_FORMATS, 72, 73, 428
NUM_SAMPLE_COUNTS, 439
NUM_SHADER_BINARY_FORMATS, 57, 60, 428
NUM_SAMPLE_COUNTS, 380, 381
NV, 487
OBJECT_TYPE, 350, 365, 424
ONE, 244, 277, 298, 300, 301, 400
ONE_MINUS_CONSTANT_ALPHA, 300
ONE_MINUS_CONSTANT_COLOR, 300
ONE_MINUS_SRC1_COLOR, 299, 300
ONE_MINUS_SRC1_ALPHA, 299, 300
ONE_MINUS_SRC_ALPHA, 300
ONE_MINUS_SRC_COLOR, 300
OR, 304
OR_INVERTED, 304
OR_REVERSE, 304
Ortho, 474
out, 129
OUT_OF_MEMORY, 19, 20, 47, 51, 240, 262, 329, 516
PACK_ALIGNMENT, 315, 408
PACK_COMPRESSED_BLOCK_DEPTH, 315, 360, 408
PACK_COMPRESSED_BLOCK_HEIGHT, 315, 360, 408
PACK_COMPRESSED_BLOCK_SIZE, 315, 360, 408
PACK_COMPRESSED_BLOCK_WIDTH, 315, 360, 408
PACK_IMAGE_HEIGHT, 315, 359, 360, 408
PACK_LSB_FIRST, 315, 408
PACK_ROW_LENGTH, 315, 360, 408
PACK_SKIP_IMAGES, 315, 359, 360, 408
PACK_SKIP_PIXELS, 315, 360, 408
PACK_SKIP_ROWS, 315, 360, 408
PACK_SWAP_BYTES, 315, 408
PassThrough, 477
patch, 125
patch in, 143
patch out, 130
PATCH_DEFAULT_INNER_LEVEL, 131, 383
PATCH_DEFAULT_OUTER_LEVEL, 131, 383
PATCH_VERTICES, 27, 383
INDEX

PATCHES, 27, 125
PatchParameterfv, 131
PatchParameteri, 27, 141
PauseTransformFeedback, 163, 164
PERSPECTIVE_CORRECTION_HINT, 478
PIXEL_BUFFER_BARRIER_BIT, 120
PIXEL_PACK_BUFFER, 44, 120, 194, 314
PIXEL_PACK_BUFFER_BINDING, 319, 359, 408
PIXEL_UNPACK_BUFFER, 44, 120, 194
PIXEL_UNPACK_BUFFER_BINDING, 196, 233, 408
PixelStore, 193, 194, 315, 323
PixelZoom, 476
POINT, 191–193, 391
POINT_FADE_THRESHOLD_SIZE, 180, 390
point_mode, 134
POINT_SIZE, 390
POINT_SIZE_GRANULARITY, 427
POINT_SIZE_RANGE, 427
POINT_SMOOTH, 475
POINT_SMOOTH_HINT, 478
POINT_SPRITE, 475
POINT_SPRITE_COORD_ORIGIN, 181, 390
PointParameter, 180
PointParameter*, 181
POINTS, 22, 146, 147, 153, 163, 164, 191, 370, 371
points, 146, 150
PointSize, 180
POLYGON, 475
POLYGON_MODE, 391
POLYGON_OFFSET_FACTOR, 391
POLYGON_OFFSET_FILL, 192, 391
POLYGON_OFFSET_LINE, 192, 391
POLYGON_OFFSET_POINT, 192, 391
POLYGON_OFFSET_UNITS, 391
POLYGON_SMOOTH, 188, 192, 391
POLYGON_SMOOTH_HINT, 353, 425
POLYGON_STIPPLE, 476
PolygonMode, 191, 193, 476
PolygonOffset, 191
PolygonStipple, 476
PopAttrib, 478
PopClientAttrib, 478
PopMatrix, 474
PopName, 477
PRIMITIVE_RESTART, 34, 386, 486, 487
PRIMITIVE_RESTART_INDEX, 386, 487
PrimitiveRestartIndex, 34
PRIMITIVES_GENERATED, 157, 168, 363, 364
PrioritizeTextures, 477
PROGRAM_BINARY_FORMATS, 73, 428
PROGRAM_BINARY_LENGTH, 71, 72, 411
PROGRAM_BINARY_RETRIEVABLE_HINT, 371, 411
PROGRAM_PIPELINE_BINDING, 411
PROGRAM_POINT_SIZE, 150, 180, 419, 493
PROGRAM_SEPARABLE, 64, 65, 67, 115, 371, 411, 519
PROGRAM_BINARY_RETRIEVABLE_HINT, 73
ProgramBinary, 63, 64, 72, 73, 75, 514
ProgramParameteri, 64, 73
ProgramUniform, 93
ProgramUniform{1234}ui, 93
ProgramUniform{1234}uiv, 93
ProgramUniformMatrix{234}, 94
INDEX

ProgramUniformMatrix \{2x3,3x2,2x4,4x2,3x4,4x3\}, 94
PROVOKING_VERTEX, 153, 389
ProvokingVertex, 153, 169
PROXY_TEXTURE_1D, 215, 225, 261, 263, 357
PROXY_TEXTURE_1D_ARRAY, 215, 224, 261, 263, 264, 357
PROXY_TEXTURE_2D, 215, 224, 261, 263, 357
PROXY_TEXTURE_2D_ARRAY, 213, 215, 261, 264, 265, 357
PROXY_TEXTURE_2D_MULTISAMPLE, 239, 261, 357
PROXY_TEXTURE_2D_MULTISAMPLE_ARRAY, 239, 261, 357
PROXY_TEXTURE_3D, 213, 260, 264, 265, 357
PROXY_TEXTURE_CUBE_MAP, 215, 224, 261, 263, 357
PROXY_TEXTURE_CUBE_MAP_ARRAY, 214, 215, 223, 261, 264, 265, 357
PROXY_TEXTURE_RECTANGLE, 215, 224, 233, 237, 261, 263, 357
PushAttrib, 478
PushClientAttrib, 478
PushMatrix, 474
PushName, 477
QUAD_STRIP, 475
QUADS, 371, 416, 475
quads, 131, 133, 136, 142
QUERY_BY_REGION_NO_WAIT, 160
QUERY_BY_REGION_WAIT, 160
QUERY_COUNTER_BITS, 364, 437
QUERY_NO_WAIT, 160
QUERY_RESULT, 364, 420
QUERY_RESULT_AVAILABLE, 364, 420
QUERY_WAIT, 160
QueryCounter, 159, 347, 348, 516
R, 480
R11F_G11F_B10F, 217, 220, 271, 273
r11f_g11f_b10f, 271
R16, 217, 219, 241, 272, 274
r16, 272
R16_SNORM, 217, 219, 272, 274
r16_snorm, 272
R16F, 217, 219, 241, 271, 274
r16f, 271
R16I, 217, 220, 241, 272, 274
r16i, 272
R16UI, 217, 220, 241, 272, 274
r16ui, 272
R32F, 217, 219, 241, 271, 274
r32f, 271
R32I, 217, 220, 241, 272, 274
r32i, 272
R32UI, 217, 220, 241, 272, 274
r32ui, 272
R3_G3_B2, 219
R8, 217, 219, 241, 272, 274, 396, 421
r8, 272
R8_SNORM, 217, 219, 272, 274
r8_snorm, 272
R8I, 217, 220, 241, 272, 274
r8i, 272
R8UI, 217, 220, 241, 272, 274
r8ui, 272
RASTERIZER_DISCARD, 175, 343, 390
RasterPos*, 475
READ_FRAMEBUFFER, 317, 403
READ_FRAMEBUFFER, 324–326, 331–333, 342, 377, 402
OpenGL 4.2 (Core Profile) - April 27, 2012
INDEX

READ_FRAMEBUFFER_BINDING, 228, 232, 315–317, 322, 327, 402
READ_ONLY, 45, 50, 51, 268, 269, 421
READ_WRITE, 45, 48, 50, 51, 268, 387
ReadBuffer, 307, 316, 323
ReadPixels, 120, 167, 193, 194, 199, 227, 313–317, 319, 321, 343, 345, 359, 476, 515, 516
Rect*, 475
RED_BITS, 477
RED_INTEGER, 198, 274
ReleaseShaderCompiler, 59
RENDERBUFFER, 328, 329, 331, 343, 378, 380, 381, 405
RENDERBUFFER_ALPHA_SIZE, 380, 406
RENDERBUFFER_BINDING, 328, 405
RENDERBUFFER_BLUE_SIZE, 380, 406
RENDERBUFFER_DEPTH_SIZE, 380, 406
RENDERBUFFER_GREEN_SIZE, 380, 406
RENDERBUFFER_HEIGHT, 330, 380, 406
RENDERBUFFER_INTERNAL_FORMAT, 330, 380, 406
RENDERBUFFER_RED_SIZE, 380, 406
RENDERBUFFER_SAMPLES, 330, 340, 342, 380, 406
RENDERBUFFER_STENCIL_SIZE, 380, 406
RENDERBUFFER_WIDTH, 329, 380, 406
RenderbufferStorage, 330, 341
RenderbufferStorageMultisample, 329, 330, 488
RENDERER, 362, 429
RendererMode, 477
REPEAT, 212, 244, 245, 251, 260
REPLACE, 293
RESCALE_NORMAL, 474
ResumeTransformFeedback, 163, 164, 167, 514
RG16, 217, 219, 241, 272, 274
rg16, 272
RG16_SNORM, 217, 219, 272
rg16_snorm, 272
RG16F, 217, 219, 241, 271, 273
rg16f, 271
RG16I, 217, 220, 242, 272, 274
rg16i, 272
RG16UI, 217, 220, 242, 272, 274
rg16ui, 272
RG32F, 217, 220, 241, 271, 273
rg32f, 271
RG32I, 217, 220, 242, 272, 274
rg32i, 272
RG32UI, 217, 220, 242, 272, 274
rg32ui, 272
RG8, 217, 219, 241, 272, 274
rg8, 272
RG8_SNORM, 217, 219, 272, 274
rg8_snorm, 272
RG8I, 217, 220, 242, 272, 274
rg8i, 272
RG8UI, 217, 220, 242, 272, 274
rg8ui, 272
RG_INTEGER, 198, 274
OpenGL 4.2 (Core Profile) - April 27, 2012
SAMPLE_ALPHA_TO_COVERAGE, 291
SAMPLE_ALPHA_TO_ONE, 291
SampleCoverage, 291, 512
SampleMaski, 292
sampler*Shadow, 112, 278
sampler1D, 87
sampler1DArray, 87
sampler1DArrayShadow, 87
sampler1DShadow, 87
sampler2D, 87, 103
sampler2DArray, 87
sampler2DArrayShadow, 87
sampler2DMS, 87
sampler2DMSArray, 87
sampler2DRect, 87
sampler2DRectShadow, 87
sampler2DShadow, 87
sampler3D, 87
SAMPLER_1D, 87
SAMPLER_1D_ARRAY, 87
SAMPLER_1D_ARRAY_SHADOW, 87
SAMPLER_1D_SHADOW, 87
SAMPLER_2D, 87
SAMPLER_2D_ARRAY, 87
SAMPLER_2D_ARRAY_SHADOW, 87
SAMPLER_2D_MULTISAMPLE, 87
SAMPLER_2D_MULTISAMPLE_ARRAY, 87
SAMPLER_2D_RECT, 87
SAMPLER_2D_RECT_SHADOW, 87
SAMPLER_2D_SHADOW, 87
SAMPLER_3D, 87
SAMPLER_BINDING, 212, 394
SAMPLER_BUFFER, 87
SAMPLER_CUBE, 87
SAMPLER_CUBE_MAP_ARRAY, 87
SAMPLER_CUBE_MAP_ARRAY_SHADOW, 87
SAMPLER_CUBE_SHADOW, 87
samplerCube, 87
samplerCubeArray, 87
samplerCubeArrayShadow, 87
samplerCubeShadow, 87
SamplerParameter, 212
SamplerParameter*, 211, 213, 361
SamplerParameter{1 i u}v, 212
SamplerParameterIiv, 213
SamplerParameterIuiv, 213
SAMPLER_INDEXED, 289
SAMPLER_INDEXEDV, 289
SecondaryColor3*, 474
SecondaryColorPointer, 474
SelectBuffer, 477
SEPARATE_ATTRIBS, 106, 107, 165, 370
SET, 304
ShadeModel, 475
SHADER_BINARY_FORMATS, 60, 428
SHADER_COMPILER, 57, 428
SHADER_IMAGE_ACCESS_BARRIER_BIT, 120, 122
SHADER_SOURCE_LENGTH, 369, 374, 409
SHADER_TYPE, 122, 369, 409

OpenGL 4.2 (Core Profile) - April 27, 2012
INDEX

TESS_CONTROL_SHADER, 126, 369, 372, 410
TESS_CONTROL_SHADER_BIT, 67
TESS_EVALUATION_SHADER, 139, 369, 372, 410
TESS_EVALUATION_SHADER_BIT, 67
TESS_GENERATE_MODE, 371, 416
TESS_GENERATE_POINT_MODE, 371, 416
TESS_GENERATE_SPACING, 371, 416
TESS_GENERATE_VERTEX_ORDER, 371, 416
TexBuffer, 240
TexCoord*, 474
TexCoordPointer, 474
TexEnv, 477
TexGen*, 475
TexImage, 209, 229
TexImage*, 221, 222, 225, 226, 228, 229, 233, 236, 255, 261
TexImage2D, 228–232, 237
TexImage3D, 228, 229, 231, 232, 237, 273
TexStorage3D, 264
TexSubImage, 120, 229
TexSubImage*, 119, 232, 238, 468
TexSubImage*D, 194
TexSubImage1D, 228, 229, 231, 237
TexSubImage2D, 228–232, 237
TexSubImage3D, 228, 229, 231, 232, 237, 273
TEXTURE, 335, 339, 343, 378, 379
TEXTUREi, 209
TEXTURE0, 209, 398
TEXTURE_x_SIZE, 396
TEXTURE_x_TYPE, 396
TEXTURE_xD, 393, 394
TEXTURE_1D, 209, 215, 225, 228, 229, 243, 256, 263, 269, 334, 356–358, 477
TEXTURE_1D_ARRAY, 209, 215, 224, 227, 229, 243, 256, 263, 264, 269, 356–358, 393, 394, 477
TEXTURE_2D_ARRAY, 209, 213, 215, 222, 229, 236, 238, 243, 256, 264, 265, 269, 356–358, 393, 394, 477
TEXTURE_2D_MULTISAMPLE, 209, 239, 261
TEXTURE_2D_MULTISAMPLE_ARRAY, 209, 239, 269, 333, 334, 357, 380, 393
TEXTURE_3D, 209, 213, 222, 229, 243, 266, 269, 333, 334, 356–358, 477
TEXTURE_ALPHA_SIZE, 357
TEXTURE_ALPHA_TYPE, 357
TEXTURE_BASE_LEVEL, 243, 245, 255, 259, 260, 337, 395

OpenGL 4.2 (Core Profile) - April 27, 2012
INDEX

TEXTURE_BINDING_2D, 393
TEXTURE_BINDING_1D_ARRAY, 393
TEXTURE_BINDING_2D_ARRAY, 393
TEXTURE_BINDING_2D_MULTISAMPLE, 393
TEXTURE_BINDING_2D_MULTISAMPLE_ARRAY, 393
TEXTURE_BINDING_BUFFER, 393
TEXTURE_BINDING_CUBE_MAP, 393
TEXTURE_BINDING_CUBE_MAP_ARRAY, 393
TEXTURE_BINDING_RECTANGLE, 393
TEXTURE_BLUE_SIZE, 357
TEXTURE_BLUE_TYPE, 357
TEXTURE_BORDER_COLOR, 212, 213, 243, 250, 251, 260, 356, 361, 395, 397, 483
TEXTURE_BUFFER, 44, 209, 240, 242, 269, 357, 393
TEXTURE_BUFFER_DATA_STORE_BINDING, 396
TEXTURE_COMPARE_FAIL_VALUE_ARB, 526
TEXTURE_COMPARE_FUNC, 213, 244, 260, 265, 395, 397
TEXTURE_COMPARE_MODE, 113, 212, 244, 260, 265, 266, 278, 395, 397
TEXTURE_COMPONENTS, 476
TEXTURE_COMPRESSED, 396
TEXTURE_COMPRESSED_IMAGE_SIZE, 236, 237, 358, 360, 396
TEXTURE_COMPRESSION_HINT, 353, 425
TEXTURE_CUBE_MAP, 209, 215, 225, 226, 243, 256, 263, 269, 356, 357, 393, 477
TEXTURE_CUBE_MAP_*, 225
TEXTURE_CUBE_MAP_NEGATIVE_X, 224, 227, 229, 246, 333, 334, 346, 357, 358, 394
TEXTURE_CUBE_MAP_NEGATIVE_Y, 224, 227, 229, 246, 333, 334, 346, 357, 358, 394
TEXTURE_CUBE_MAP_NEGATIVE_Z, 224, 227, 229, 246, 333, 334, 346, 357, 358, 394
TEXTURE_CUBE_MAP_POSITIVE_X, 224, 225, 227, 229, 246, 333, 334, 346, 357, 358, 394
TEXTURE_CUBE_MAP_POSITIVE_Y, 224, 227, 229, 246, 333, 334, 346, 357, 358, 394
TEXTURE_CUBE_MAP_POSITIVE_Z, 224, 227, 229, 246, 333, 334, 346, 357, 358, 394
TEXTURE_CUBE_MAP_SEAMLESS, 246, 442
TEXTURE_DEPTH, 236–238, 358, 396
TEXTURE_DEPTH_SIZE, 357
TEXTURE_DEPTH_TYPE, 357
TEXTURE_ENV, 477
TEXTURE_FETCH_BARRIER_BIT, 120
TEXTURE_FILTER_CONTROL, 477
TEXTURE_FIXED_SAMPLE_LOCATIONS, 239, 340, 396
TEXTURE_GEN.*, 475
TEXTURE_GREEN_SIZE, 357
TEXTURE_GREEN_TYPE, 357

OpenGL 4.2 (Core Profile) - April 27, 2012
INDEX

TEXTURE_HEIGHT, 232, 236–239, 358, 396
TEXTURE_IMMUTABLE_FORMAT, 245, 260–262, 265, 356, 395
TEXTURE_INTERNAL_FORMAT, 236–239, 358, 396, 476
TEXTURE_LOD_BIAS, 212, 244, 247, 395, 397, 477
TEXTURE_MAG_FILTER, 212, 244, 257, 260, 266, 395, 397
TEXTURE_MAX_LEVEL, 244, 245, 255, 259, 260, 337, 395
TEXTURE_MAX_LOD, 212, 244, 245, 248, 260, 395, 397
TEXTURE_MIN_FILTER, 212, 244, 245, 250, 251, 254, 257, 259, 260, 266, 337, 395, 397
TEXTURE_MIN_LOD, 212, 244, 245, 248, 260, 395, 397
TEXTURE_PRIORITY, 477
TEXTURE_RECTANGLE_ARB, 528
TEXTURE_RED_SIZE, 357
TEXTURE_RED_TYPE, 357
TEXTURE_SAMPLES, 239, 340, 342, 396
TEXTURE_SHARED_SIZE, 357, 396
TEXTURE_STENCIL_SIZE, 357
TEXTURE_SWIZZLE_A, 243, 244, 260, 277, 395
TEXTURE_SWIZZLE_B, 243, 244, 260, 277, 395
TEXTURE_SWIZZLE_G, 243, 244, 260, 277, 395
TEXTURE_SWIZZLE_R, 243, 244, 260, 277, 395
TEXTURE_SWIZZLE_RGBA, 243, 244

TEXTURE_UPDATE_BARRIER_BIT, 120
TEXTURE_WIDTH, 232, 236–239, 358, 396
TEXTURE_WRAP_R, 212, 244, 245, 251, 395, 397, 476, 515
TEXTURE_WRAP_S, 212, 244, 245, 251, 395, 397, 476
TEXTURE_WRAP_T, 212, 244, 245, 251, 395, 397, 476
TEXTURE_2D_MULTISAMPLE, 381
TEXTURE_2D_MULTISAMPLE_ARRAY, 381
textureGather, 248, 252, 253, 433, 502, 535
textureGatherOffset, 252
textureLOD, 501, 535
timeElapsed, 347, 348, 363, 364
timeoutExpired, 351
timeoutIgnored, 351, 352
timestamp, 347, 348, 363, 364
transformFeedback, 162
transformFeedback_ACTIVE, 422, 512, 513
transformFeedback_BUFFER, 44, 45, 164, 167
transformFeedback_BUFFER_ACTIVE, 512
transformFeedback_BUFFER_BINDING, 367, 422
transformFeedback_BUFFER_MODE, 370, 413
transformFeedback_BUFFERPaused, 512

OpenGL 4.2 (Core Profile) - April 27, 2012
INDEX

TRANSFORM_FEEDBACK_BUFFER_SIZE, 367, 422
TRANSFORM_FEEDBACK_BUFFER_START, 367, 422
TRANSFORM_FEEDBACK_PAUSED, 422, 512, 513
TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, 157, 166, 169, 363, 364
TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH, 108, 370, 413
TRANSFORM_FEEDBACK_VARYINGS, 108, 370, 413
Translate, 474
TRIANGLE_FAN, 23, 28, 146, 153, 164
TRIANGLE_STRIP, 23, 28, 146, 147, 153, 155, 164, 371, 413
TRIANGLE_STRIP_ADJACENCY, 26, 28, 146, 154
TRIANGLES, 23, 28, 146, 153, 163, 164, 370, 371, 413
triangles, 131, 133, 134, 146
TRIANGLES_ADJACENCY, 25, 28, 146, 154, 370
triangles_adjacency, 146
uimage1D, 89
uimage1DArray, 89
uimage2D, 89
uimage2DArray, 89
uimage2DMS, 89
uimage2DMSArray, 89
uimage2DRect, 89
uimage3D, 89
uimageBuffer, 89
uimageCube, 89
uimageCubeArray, 89
uint, 70, 74, 86, 95, 99, 514
UNDEFINED_VERTEX, 153
Uniform, 14, 91
Uniform*, 78, 79, 92, 93, 103
Uniform*d{v}, 92
Uniform*f{v}, 92
Uniform*i{v}, 92
Uniform*ui{v}, 92
Uniform1f, 15
Uniform1i, 15
Uniform1i{v}, 92, 103
Uniform1iv, 92
Uniform2{if ui}*{v}, 92
Uniform2f, 15
Uniform2i, 15
Uniform3f, 15
Uniform3i, 15
Uniform4f, 13, 15
Uniform4f{v}, 93
Uniform4i, 15
UNIFORM_ARRAY_STRIDE, 96, 415
UNIFORM_ATOMIC_COUNTER_BUFFER_INDEX, 91, 418
UNIFORM_BARRIER_BIT, 120
UNIFORM_BLOCK_ACTIVE_UNIFORM_INDICES, 81, 415
UNIFORM_BLOCK_ACTIVE_UNIFORMS, 81, 415
UNIFORM_BLOCK_BINDING, 81, 415
OpenGL 4.2 (Core Profile) - April 27, 2012
UNIFORM_BLOCK_DATA_SIZE, 81, 98, 415
UNIFORM_BLOCK_INDEX, 90, 414
UNIFORM_BLOCK_NAME, LENGTH, 81
UNIFORM_BLOCK_REFERENCED_BY_FRAGMENT_SHADER, 81, 415
UNIFORM_BLOCK_REFERENCED_BY_GEOMETRY_SHADER, 81, 415
UNIFORM_BLOCK_REFERENCED_BY_TESS_CONTROL_SHADER, 81, 415
UNIFORM_BLOCK_REFERENCED_BY_TESS_EVALUATION_SHADER, 81, 415
UNIFORM_BUFFER, 44, 45, 98
UNIFORM_BUFFER_BINDING, 367, 414
UNIFORM_BUFFER_OFFSET_ALIGNMENT, 98, 434
UNIFORM_BUFFER_SIZE, 367, 414
UNIFORM_BUFFER_START, 367, 414
UNIFORM_IS_ROW_MAJOR, 91, 415
UNIFORM_MATRIX_ROW_MAJOR, 90, 95, 96, 415
UNIFORM_NAME_LENGTH, 90, 102, 103, 414, 417
UNIFORM_OFFSET, 90, 414
UNIFORM_SIZE, 90, 102, 414, 417
UNIFORM_TYPE, 90, 414
Uniform{1234}ui, 91
Uniform{1234}uiv, 91
UniformBlockBinding, 98
UniformMatrix2x4fv, 92
UniformMatrix3dv, 92
UniformMatrix{234}, 91
UniformMatrix{234}dv, 92
UniformMatrix{234}fv, 92
UniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}, 91
UniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}dv, 92
UniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}fv, 92
UniformSubroutinesuiv, 102
UnmapBuffer, 50, 52, 78, 469
UNPACK_ALIGNMENT, 194, 199, 214, 235, 407
UNPACK_COMPRESSED_BLOCK_DEPTH, 194, 233, 407
UNPACK_COMPRESSED_BLOCK_HEIGHT, 194, 233, 407
UNPACK_COMPRESSED_BLOCK_SIZE, 194, 233, 407
UNPACK_COMPRESSED_BLOCK_WIDTH, 194, 233, 407
UNPACK_IMAGE_HEIGHT, 194, 214, 234, 235, 407
UNPACK_LSB_FIRST, 194, 407
UNPACK_ROW_LENGTH, 194, 196, 199, 214, 234, 235, 407
UNPACK_SKIP_IMAGES, 194, 214, 225, 234, 235, 407
UNPACK_SKIP_PIXELS, 194, 199, 234, 235, 407
UNPACK_SKIP_ROWS, 194, 199, 234, 235, 407
UNPACK_SWAP_BYTES, 194, 196, 198, 407
UNSIGNALED, 350, 365, 424
Open GL 4.2 (Core Profile) - April 27, 2012
INDEX

UNSIGNED, 462
UNSIGNED_BYTE, 30, 31, 33, 38, 197, 274, 315, 320, 321
UNSIGNED_BYTE.2.3.3.REV, 197, 201, 202, 320
UNSIGNED_BYTE.3.3.2, 197, 201, 202, 320
UNSIGNED_INT, 30, 33, 38, 86, 197, 271, 274, 320, 321, 357, 378
UNSIGNED_INT.10.10.10.2, 197, 201, 204, 320
UNSIGNED_INT.10F.11F.11F.REV, 197, 201, 204, 206, 215, 273, 317, 318, 320
UNSIGNED_INT.24.8, 195, 197, 201, 204, 315, 320, 321
UNSIGNED_INT.2.10.10.10.REV, 29–31, 34, 197, 201, 204, 274, 320
UNSIGNED_INT.5.9.9.9.REV, 197, 201, 204, 206, 215, 218, 317, 318, 320
UNSIGNED_INT.8.8.8.8, 197, 201, 204, 320
UNSIGNED_INT.8.8.8.8.REV, 197, 201, 204, 320
UNSIGNED_INT.ATOMIC.-COUNTER, 89
UNSIGNED_INT_IMAGE.1D, 89
UNSIGNED_INT_IMAGE.1D_ARRAY, 89
UNSIGNED_INT_IMAGE.2D, 89
UNSIGNED_INT_IMAGE.2D_ARRAY, 89
UNSIGNED_INT_IMAGE.2D_MULTISAMPLE, 89
UNSIGNED_INT_IMAGE.2D_MULTISAMPLE_ARRAY, 89
UNSIGNED_INT_IMAGE.2D_RECT, 89
UNSIGNED_INT_IMAGE.3D, 89
UNSIGNED_INT_IMAGE_BUFFER, 89
UNSIGNED_INT_IMAGE_CUBE, 89
UNSIGNED_INT_IMAGE_CUBE.-MAP_ARRAY, 89
UNSIGNED_INT_SAMPLER.1D, 88
UNSIGNED_INT_SAMPLER.1D_ARRAY, 88
UNSIGNED_INT_SAMPLER.2D, 88
UNSIGNED_INT_SAMPLER.2D_ARRAY, 88
UNSIGNED_INT_SAMPLER.2D.-MULTISAMPLE, 88
UNSIGNED_INT_SAMPLER.2D.-MULTISAMPLE_ARRAY, 88
UNSIGNED_INT_SAMPLER.2D.-RECT, 88
UNSIGNED_INT_SAMPLER.3D, 88
UNSIGNED_INT_SAMPLER.-BUFFER, 88
UNSIGNED_INT_SAMPLER.CUBE, 88
UNSIGNED_INT_SAMPLER.CUBE.-MAP_ARRAY, 88
UNSIGNED_INT_VEC2, 86
UNSIGNED_INT_VEC3, 86
UNSIGNED_INT_VEC4, 86
UNSIGNED_NORMALIZED, 357, 378
UNSIGNED_SHORT, 30, 33, 38, 197, 274, 320, 321
UNSIGNED_SHORT.1.5.5.5.REV, 197, 201, 203, 320
UNSIGNED_SHORT.4.4.4.4, 197, 201, 203, 320
UNSIGNED_SHORT.4.4.4.4.REV, 197, 201, 203, 320
UNSIGNED_SHORT.5.5.5.1, 197, 201, 203, 320
UNSIGNED_SHORT.5.6.5, 197, 201,
INDEX

203, 320
UNSIGNED_SHORT_5_6_5_REV, 197, 201, 203, 320
UPPER_LEFT, 181, 390
usampler1D, 88
usampler1DArray, 88
usampler2D, 88
usampler2DArray, 88
usampler2DMS, 88
usampler2DMSArray, 88
usampler2DRect, 88
usampler3D, 88
usamplerBuffer, 88
usamplerCube, 88
usamplerCubeArray, 88
UseProgram, 63, 64, 66, 91, 103, 166
UseProgramStages, 64, 67, 103, 115, 166, 371
uvec2, 74, 86
uvec3, 74, 86
uvec4, 74, 86, 271
VALIDATE_STATUS, 116, 370, 372, 410, 411
ValidateProgram, 116, 370
ValidateProgramPipeline, 116, 372
vec2, 74, 86
vec3, 74, 86
vec4, 74, 86, 93, 271
VENDOR, 362, 429
VERSION, 362, 429
Vertex*, 474
VERTEX_ATTRIB_ARRAY_BINDING, 356, 375, 386
VERTEX_ATTRIB_ARRAY_BINDING_BIT, 119
VERTEX_ATTRIB_ARRAY_BUFFER, 120
VERTEX_ATTRIB_ARRAY_BUFFER_BINDING, 54, 375, 385
VERTEX_ATTRIB_ARRAY_DIVISOR, 375, 384
VERTEX_ATTRIB_ARRAY_ENABLED, 375, 384
VERTEX_ATTRIB_ARRAY_INTEGER, 375, 384
VERTEX_ATTRIB_ARRAY_NORMALIZED, 375, 384
VERTEX_ATTRIB_ARRAY_POINTER, 376, 384
VERTEX_ATTRIB_ARRAY_SIZE, 375, 384
VERTEX_ATTRIB_ARRAY_STRIDE, 375, 384
VERTEX_ATTRIB_ARRAY_TYPE, 375, 384
VERTEX_PROGRAM_POINT_SIZE, 493
VERTEX_PROGRAM_TWO_SIDE, 475
VERTEX_SHADER, 58, 369, 372, 374, 410
VERTEX_SHADER_BIT, 67
VertexAttrib, 28, 160
VertexAttrib*, 29, 73, 474
VertexAttrib1*, 29, 74
VertexAttrib2*, 29, 74
VertexAttrib3*, 29, 74
VertexAttrib4, 28
VertexAttrib4*, 29, 74
VertexAttrib4N, 28
VertexAttrib4Nub, 28
VertexAttribDivisor, 33, 36, 38, 41
VertexAttribI, 28
VertexAttribI1i, 74
VertexAttribI2i, 74
VertexAttribI2ui, 74
VertexAttribI3i, 74
VertexAttribI3ui, 74

OpenGL 4.2 (Core Profile) - April 27, 2012