Contents

1 Introduction ............................................ 1
   1.1 Comments on edits to the OpenGL 3.1 Specification .... 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 ...................................... 5
   2.1 OpenGL Fundamentals ............................. 5
      2.1.1 Floating-Point Computation .................. 7
      2.1.2 16-Bit Floating-Point Numbers ............... 8
      2.1.3 Unsigned 11-Bit Floating-Point Numbers .... 8
      2.1.4 Unsigned 10-Bit Floating-Point Numbers .... 9
      2.1.5 Fixed-Point Data Conversions ................ 10
   2.2 GL State .......................................... 12
      2.2.1 Shared Object State ......................... 13
   2.3 GL Command Syntax ................................ 13
   2.4 Basic GL Operation ............................... 15
   2.5 GL Errors ........................................ 18
   2.6 Primitives and Vertices ........................... 19
      2.6.1 Primitive Types ............................... 21
   2.7 Vertex Specification ............................. 25
   2.8 Vertex Arrays .................................... 27
      2.8.1 Transferring Array Elements ................. 29
      2.8.2 Drawing Commands ............................ 30
2.9 Buffer Objects .................................................. 34
  2.9.1 Creating and Binding Buffer Objects ..................... 35
  2.9.2 Creating Buffer Object Data Stores ...................... 37
  2.9.3 Mapping and Unmapping Buffer Data ....................... 39
  2.9.4 Effects of Accessing Outside Buffer Bounds ............... 43
  2.9.5 Copying Between Buffers ................................ 43
  2.9.6 Vertex Arrays in Buffer Objects ......................... 44
  2.9.7 Array Indices in Buffer Objects ......................... 45
  2.9.8 Buffer Object State ................................... 45
2.10 Vertex Array Objects ......................................... 46
2.11 Vertex Shaders ................................................ 47
  2.11.1 Shader Objects ........................................ 47
  2.11.2 Program Objects ....................................... 49
  2.11.3 Vertex Attributes ..................................... 51
  2.11.4 Uniform Variables ..................................... 54
  2.11.5 Samplers ............................................... 70
  2.11.6 Varying Variables ..................................... 71
  2.11.7 Shader Execution ...................................... 74
  2.11.8 Required State ........................................ 79
2.12 Geometry Shaders ............................................. 81
  2.12.1 Geometry Shader Input Primitives ....................... 81
  2.12.2 Geometry Shader Output Primitives ...................... 83
  2.12.3 Geometry Shader Variables ............................. 83
  2.12.4 Geometry Shader Execution Environment ................ 84
2.13 Coordinate Transformations ................................. 88
  2.13.1 Controlling the Viewport ............................... 89
2.14 Asynchronous Queries ........................................ 90
2.15 Conditional Rendering ....................................... 91
2.16 Transform Feedback ......................................... 92
2.17 Primitive Queries .......................................... 95
2.18 Flatshading .................................................. 96
2.19 Primitive Clipping .......................................... 96
  2.19.1 Clipping Shader Varying Outputs ....................... 99
3 Rasterization ............................................. 100
  3.1 Discarding Primitives Before Rasterization ................ 101
  3.2 Invariance ................................................ 101
  3.3 Antialiasing .............................................. 102
  3.3.1 Multisampling ......................................... 103
  3.4 Points ................................................... 104
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.1</td>
<td>Basic Point Rasterization</td>
<td>106</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Point Rasterization State</td>
<td>106</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Point Multisample Rasterization</td>
<td>107</td>
</tr>
<tr>
<td>3.5</td>
<td>Line Segments</td>
<td>107</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Basic Line Segment Rasterization</td>
<td>107</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Other Line Segment Features</td>
<td>110</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Line Rasterization State</td>
<td>110</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Line Multisample Rasterization</td>
<td>110</td>
</tr>
<tr>
<td>3.6</td>
<td>Polygons</td>
<td>111</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Basic Polygon Rasterization</td>
<td>112</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Antialiasing</td>
<td>114</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Options Controlling Polygon Rasterization</td>
<td>114</td>
</tr>
<tr>
<td>3.6.4</td>
<td>Depth Offset</td>
<td>115</td>
</tr>
<tr>
<td>3.6.5</td>
<td>Polygon Multisample Rasterization</td>
<td>116</td>
</tr>
<tr>
<td>3.6.6</td>
<td>Polygon Rasterization State</td>
<td>117</td>
</tr>
<tr>
<td>3.7</td>
<td>Pixel Rectangles</td>
<td>117</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Pixel Storage Modes and Pixel Buffer Objects</td>
<td>117</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Transfer of Pixel Rectangles</td>
<td>118</td>
</tr>
<tr>
<td>3.8</td>
<td>Texturing</td>
<td>130</td>
</tr>
<tr>
<td>3.8.1</td>
<td>Texture Image Specification</td>
<td>131</td>
</tr>
<tr>
<td>3.8.2</td>
<td>Alternate Texture Image Specification Commands</td>
<td>143</td>
</tr>
<tr>
<td>3.8.3</td>
<td>Compressed Texture Images</td>
<td>150</td>
</tr>
<tr>
<td>3.8.4</td>
<td>Multisample Textures</td>
<td>154</td>
</tr>
<tr>
<td>3.8.5</td>
<td>Buffer Textures</td>
<td>155</td>
</tr>
<tr>
<td>3.8.6</td>
<td>Texture Parameters</td>
<td>157</td>
</tr>
<tr>
<td>3.8.7</td>
<td>Depth Component Textures</td>
<td>160</td>
</tr>
<tr>
<td>3.8.8</td>
<td>Cube Map Texture Selection</td>
<td>160</td>
</tr>
<tr>
<td>3.8.9</td>
<td>Texture Minification</td>
<td>161</td>
</tr>
<tr>
<td>3.8.10</td>
<td>Texture Magnification</td>
<td>170</td>
</tr>
<tr>
<td>3.8.11</td>
<td>Combined Depth/Stencil Textures</td>
<td>171</td>
</tr>
<tr>
<td>3.8.12</td>
<td>Texture Completeness</td>
<td>171</td>
</tr>
<tr>
<td>3.8.13</td>
<td>Texture State and Proxy State</td>
<td>172</td>
</tr>
<tr>
<td>3.8.14</td>
<td>Texture Objects</td>
<td>174</td>
</tr>
<tr>
<td>3.8.15</td>
<td>Texture Comparison Modes</td>
<td>176</td>
</tr>
<tr>
<td>3.8.16</td>
<td>sRGB Texture Color Conversion</td>
<td>177</td>
</tr>
<tr>
<td>3.8.17</td>
<td>Shared Exponent Texture Color Conversion</td>
<td>178</td>
</tr>
<tr>
<td>3.9</td>
<td>Fragment Shaders</td>
<td>178</td>
</tr>
<tr>
<td>3.9.1</td>
<td>Shader Variables</td>
<td>178</td>
</tr>
<tr>
<td>3.9.2</td>
<td>Shader Execution</td>
<td>179</td>
</tr>
<tr>
<td>3.10</td>
<td>Antialiasing Application</td>
<td>184</td>
</tr>
</tbody>
</table>

OpenGL 3.2 (Core Profile) - July 24, 2009
### Contents

3.11 Multisample Point Fade .......................... 184

### 4 Per-Fragment Operations and the Framebuffer

4.1 Per-Fragment Operations .......................... 186
  4.1.1 Pixel Ownership Test ............................ 187
  4.1.2 Scissor Test ..................................... 187
  4.1.3 Multisample Fragment Operations ............... 188
  4.1.4 Stencil Test ..................................... 189
  4.1.5 Depth Buffer Test ................................ 191
  4.1.6 Occlusion Queries ............................... 192
  4.1.7 Blending ........................................ 193
  4.1.8 sRGB Conversion .................................. 197
  4.1.9 Dithering ........................................ 198
  4.1.10 Logical Operation ............................... 198
  4.1.11 Additional Multisample Fragment Operations ... 200

4.2 Whole Framebuffer Operations ....................... 200
  4.2.1 Selecting a Buffer for Writing .................. 201
  4.2.2 Fine Control of Buffer Updates .................. 205
  4.2.3 Clearing the Buffers ............................ 206

4.3 Reading and Copying Pixels ......................... 209
  4.3.1 Reading Pixels .................................... 209
  4.3.2 Copying Pixels .................................... 216
  4.3.3 Pixel Draw/Read State ......................... 218

4.4 Framebuffer Objects ............................... 219
  4.4.1 Binding and Managing Framebuffer Objects ....... 219
  4.4.2 Attaching Images to Framebuffer Objects ....... 222
  4.4.3 Feedback Loops Between Textures and the Framebuffer ... 230
  4.4.4 Framebuffer Completeness ....................... 233
  4.4.5 Effects of Framebuffer State on Framebuffer Dependent Values .......................... 238
  4.4.6 Mapping between Pixel and Element in Attached Image ............ 238
  4.4.7 Layered Framebuffers ............................ 239

5 Special Functions .................................. 241
  5.1 Flush and Finish ................................... 241
  5.2 Sync Objects and Fences ........................... 241
    5.2.1 Waiting for Sync Objects ....................... 243
    5.2.2 Signalling ..................................... 245
  5.3 Hints .............................................. 246
6 State and State Requests

6.1 Querying GL State

6.1.1 Simple Queries
6.1.2 Data Conversions
6.1.3 Enumerated Queries
6.1.4 Texture Queries
6.1.5 String Queries
6.1.6 Asynchronous Queries
6.1.7 Sync Object Queries
6.1.8 Buffer Object Queries
6.1.9 Vertex Array Object Queries
6.1.10 Shader and Program Queries
6.1.11 Framebuffer Object Queries
6.1.12 Renderbuffer Object Queries

6.2 State Tables

A Invariance

A.1 Repeatability
A.2 Multi-pass Algorithms
A.3 Invariance Rules
A.4 What All This Means

B Corollaries

C Compressed Texture Image Formats

C.1 RGTC Compressed Texture Image Formats
C.1.1 Format COMPRESSED_RED_RGTC1
C.1.2 Format COMPRESSED_SIGNED_RED_RGTC1
C.1.3 Format COMPRESSED_RG_RGTC2
C.1.4 Format COMPRESSED_SIGNED_RG_RGTC2

D Shared Objects and Multiple Contexts

D.1 Object Deletion Behavior
D.1.1 Automatic Unbinding of Deleted Objects
D.1.2 Deleted Object and Object Name Lifetimes
D.2 Sync Objects and Multiple Contexts
D.3 Propagating State Changes
D.3.1 Determining Completion of Changes to an object
D.3.2 Definitions
D.3.3 Rules
## Contents

### E Profiles and the Deprecation Model

- E.1 Core and Compatibility Profiles .............................................. 329
- E.2 Deprecated and Removed Features .......................................... 329
  - E.2.1 Deprecated But Still Supported Features .......................... 329
  - E.2.2 Removed Features .................................................... 330

### F Version 3.0 and Before

- F.1 New Features ............................................................... 335
- F.2 Deprecation Model ....................................................... 336
- F.3 Changed Tokens ........................................................... 337
- F.4 Change Log ................................................................. 337
- F.5 Credits and Acknowledgements ........................................... 339

### G Version 3.1

- G.1 New Features ............................................................... 342
- G.2 Deprecation Model ....................................................... 343
- G.3 Change Log ................................................................. 343
- G.4 Credits and Acknowledgements ........................................... 344

### H Version 3.2

- H.1 New Features ............................................................... 347
- H.2 Deprecation Model ....................................................... 348
- H.3 Changed Tokens ........................................................... 348
- H.4 Change Log ................................................................. 349
- H.5 Credits and Acknowledgements ........................................... 349

### I Extension Registry, Header Files, and ARB Extensions

- I.1 Extension Registry ......................................................... 351
- I.2 Header Files ............................................................... 351
- I.3 ARB Extensions .............................................................. 352
  - I.3.1 Naming Conventions ................................................ 352
  - I.3.2 Promoting Extensions to Core Features .......................... 353
  - I.3.3 Multitexture ........................................................... 353
  - I.3.4 Transpose Matrix .................................................... 353
  - I.3.5 Multisample ........................................................... 354
  - I.3.6 Texture Add Environment Mode ................................... 354
  - I.3.7 Cube Map Textures ................................................ 354
  - I.3.8 Compressed Textures ............................................... 354
  - I.3.9 Texture Border Clamp .............................................. 354
  - I.3.10 Point Parameters .................................................. 354

OpenGL 3.2 (Core Profile) - July 24, 2009
<table>
<thead>
<tr>
<th>I.3.11</th>
<th>Vertex Blend</th>
<th>354</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.3.12</td>
<td>Matrix Palette</td>
<td>355</td>
</tr>
<tr>
<td>I.3.13</td>
<td>Texture Combine Environment Mode</td>
<td>355</td>
</tr>
<tr>
<td>I.3.14</td>
<td>Texture Crossbar Environment Mode</td>
<td>355</td>
</tr>
<tr>
<td>I.3.15</td>
<td>Texture Dot3 Environment Mode</td>
<td>355</td>
</tr>
<tr>
<td>I.3.16</td>
<td>Texture Mirrored Repeat</td>
<td>355</td>
</tr>
<tr>
<td>I.3.17</td>
<td>Depth Texture</td>
<td>355</td>
</tr>
<tr>
<td>I.3.18</td>
<td>Shadow</td>
<td>355</td>
</tr>
<tr>
<td>I.3.19</td>
<td>Shadow Ambient</td>
<td>355</td>
</tr>
<tr>
<td>I.3.20</td>
<td>Window Raster Position</td>
<td>356</td>
</tr>
<tr>
<td>I.3.21</td>
<td>Low-Level Vertex Programming</td>
<td>356</td>
</tr>
<tr>
<td>I.3.22</td>
<td>Low-Level Fragment Programming</td>
<td>356</td>
</tr>
<tr>
<td>I.3.23</td>
<td>Buffer Objects</td>
<td>356</td>
</tr>
<tr>
<td>I.3.24</td>
<td>Occlusion Queries</td>
<td>356</td>
</tr>
<tr>
<td>I.3.25</td>
<td>Shader Objects</td>
<td>356</td>
</tr>
<tr>
<td>I.3.26</td>
<td>High-Level Vertex Programming</td>
<td>357</td>
</tr>
<tr>
<td>I.3.27</td>
<td>High-Level Fragment Programming</td>
<td>357</td>
</tr>
<tr>
<td>I.3.28</td>
<td>OpenGL Shading Language</td>
<td>357</td>
</tr>
<tr>
<td>I.3.29</td>
<td>Non-Power-Of-Two Textures</td>
<td>357</td>
</tr>
<tr>
<td>I.3.30</td>
<td>Point Sprites</td>
<td>357</td>
</tr>
<tr>
<td>I.3.31</td>
<td>Fragment Program Shadow</td>
<td>357</td>
</tr>
<tr>
<td>I.3.32</td>
<td>Multiple Render Targets</td>
<td>357</td>
</tr>
<tr>
<td>I.3.33</td>
<td>Rectangular Textures</td>
<td>358</td>
</tr>
<tr>
<td>I.3.34</td>
<td>Floating-Point Color Buffers</td>
<td>358</td>
</tr>
<tr>
<td>I.3.35</td>
<td>Half-Precision Floating Point</td>
<td>358</td>
</tr>
<tr>
<td>I.3.36</td>
<td>Floating-Point Textures</td>
<td>359</td>
</tr>
<tr>
<td>I.3.37</td>
<td>Pixel Buffer Objects</td>
<td>359</td>
</tr>
<tr>
<td>I.3.38</td>
<td>Floating-Point Depth Buffers</td>
<td>359</td>
</tr>
<tr>
<td>I.3.39</td>
<td>Instanced Rendering</td>
<td>359</td>
</tr>
<tr>
<td>I.3.40</td>
<td>Framebuffer Objects</td>
<td>359</td>
</tr>
<tr>
<td>I.3.41</td>
<td>sRGB Framebuffers</td>
<td>359</td>
</tr>
<tr>
<td>I.3.42</td>
<td>Geometry Shaders</td>
<td>360</td>
</tr>
<tr>
<td>I.3.43</td>
<td>Half-Precision Vertex Data</td>
<td>360</td>
</tr>
<tr>
<td>I.3.44</td>
<td>Instanced Rendering</td>
<td>360</td>
</tr>
<tr>
<td>I.3.45</td>
<td>Flexible Buffer Mapping</td>
<td>360</td>
</tr>
<tr>
<td>I.3.46</td>
<td>Texture Buffer Objects</td>
<td>360</td>
</tr>
<tr>
<td>I.3.47</td>
<td>RGTC Texture Compression Formats</td>
<td>361</td>
</tr>
<tr>
<td>I.3.48</td>
<td>One- and Two-Component Texture Formats</td>
<td>361</td>
</tr>
<tr>
<td>I.3.49</td>
<td>Vertex Array Objects</td>
<td>361</td>
</tr>
<tr>
<td>I.3.50</td>
<td>Versioned Context Creation</td>
<td>361</td>
</tr>
</tbody>
</table>
CONTENTS

I.3.51 Uniform Buffer Objects . . . . . . . . . . . . . . . . 361
I.3.52 Restoration of features removed from OpenGL 3.0 . . 361
I.3.53 Fast Buffer-to-Buffer Copies . . . . . . . . . . . . . 362
I.3.54 Shader Texture Level of Detail Control . . . . . . . . 362
I.3.55 Depth Clamp Control . . . . . . . . . . . . . . . . . 362
I.3.56 Base Vertex Offset Drawing Commands . . . . . . . . 362
I.3.57 Fragment Coordinate Convention Control . . . . . . . 363
I.3.58 Provoking Vertex Control . . . . . . . . . . . . . . . . 363
I.3.59 Seamless Cube Maps . . . . . . . . . . . . . . . . . 363
I.3.60 Fence Sync Objects . . . . . . . . . . . . . . . . . . . 363
I.3.61 Multisample Textures . . . . . . . . . . . . . . . . . 363
I.3.62 BGRA Attribute Component Ordering . . . . . . . . . 363
I.3.63 Per-Buffer Blend Control . . . . . . . . . . . . . . . . 363
I.3.64 Sample Shading Control . . . . . . . . . . . . . . . . 364
I.3.65 Cube Map Array Textures . . . . . . . . . . . . . . . . 364
I.3.66 Texture Gather . . . . . . . . . . . . . . . . . . . . . 364
I.3.67 Texture Level-Of-Detail Queries . . . . . . . . . . . 364
I.3.68 Profiled Context Creation . . . . . . . . . . . . . . . . 364
## List of Figures

2.1 Block diagram of the GL. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15  
2.2 Vertex processing and primitive assembly. . . . . . . . . . . . . . . . . . . . 19  
2.3 Triangle strips, fans, and independent triangles. . . . . . . . . . . . . . . . . 22  
2.4 Lines with adjacency. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23  
2.5 Triangles with adjacency. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24  
2.6 Triangle strips with adjacency. . . . . . . . . . . . . . . . . . . . . . . . . . 24  

3.1 Rasterization. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100  
3.2 Visualization of Bresenham’s algorithm. . . . . . . . . . . . . . . . . . . . . . 108  
3.3 The region used in rasterizing an antialiased line segment. . . . . . . . . . . . 110  
3.4 Transfer of pixel rectangles. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119  
3.5 Selecting a subimage from an image . . . . . . . . . . . . . . . . . . . . . . . 123  
3.6 A texture image and the coordinates used to access it. . . . . . . . . . . . . . 143  

4.1 Per-fragment operations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 187  
4.2 Operation of **ReadPixels**. . . . . . . . . . . . . . . . . . . . . . . . . . . . 209
List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>GL command suffixes</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>GL data types</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>Summary of GL errors</td>
<td>19</td>
</tr>
<tr>
<td>2.4</td>
<td>Triangles generated by triangle strips with adjacency</td>
<td>26</td>
</tr>
<tr>
<td>2.5</td>
<td>Vertex array sizes (values per vertex) and data types</td>
<td>28</td>
</tr>
<tr>
<td>2.6</td>
<td>Buffer object binding targets</td>
<td>35</td>
</tr>
<tr>
<td>2.7</td>
<td>Buffer object parameters and their values</td>
<td>36</td>
</tr>
<tr>
<td>2.8</td>
<td>Buffer object initial state</td>
<td>38</td>
</tr>
<tr>
<td>2.9</td>
<td>Buffer object state set by <strong>MapBufferRange</strong></td>
<td>41</td>
</tr>
<tr>
<td>2.10</td>
<td>OpenGL Shading Language type tokens</td>
<td>63</td>
</tr>
<tr>
<td>2.11</td>
<td>Transform feedback modes</td>
<td>93</td>
</tr>
<tr>
<td>2.12</td>
<td>Provoking vertex selection</td>
<td>97</td>
</tr>
<tr>
<td>3.1</td>
<td><strong>PixelStore</strong> parameters</td>
<td>118</td>
</tr>
<tr>
<td>3.2</td>
<td>Pixel data types</td>
<td>120</td>
</tr>
<tr>
<td>3.3</td>
<td>Pixel data formats</td>
<td>121</td>
</tr>
<tr>
<td>3.4</td>
<td>Swap Bytes bit ordering</td>
<td>122</td>
</tr>
<tr>
<td>3.5</td>
<td>Packed pixel formats</td>
<td>124</td>
</tr>
<tr>
<td>3.6</td>
<td><strong>UNSIGNED_BYTE</strong> formats</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>Bit numbers are indicated for each component</td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td><strong>UNSIGNED_SHORT</strong> formats</td>
<td>126</td>
</tr>
<tr>
<td>3.8</td>
<td><strong>UNSIGNED_INT</strong> formats</td>
<td>127</td>
</tr>
<tr>
<td>3.9</td>
<td><strong>FLOAT</strong> <strong>UNSIGNED_INT</strong> formats</td>
<td>128</td>
</tr>
<tr>
<td>3.10</td>
<td>Packed pixel field assignments</td>
<td>129</td>
</tr>
<tr>
<td>3.11</td>
<td>Conversion from RGBA, depth, and stencil pixel components to internal texture components</td>
<td>133</td>
</tr>
<tr>
<td>3.12</td>
<td>Sized internal color formats</td>
<td>138</td>
</tr>
<tr>
<td>3.13</td>
<td>Sized internal depth and stencil formats</td>
<td>139</td>
</tr>
<tr>
<td>3.14</td>
<td>Generic and specific compressed internal formats</td>
<td>140</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>3.15</td>
<td>Internal formats for buffer textures</td>
<td></td>
</tr>
<tr>
<td>3.16</td>
<td>Texture parameters and their values.</td>
<td></td>
</tr>
<tr>
<td>3.17</td>
<td>Selection of cube map images.</td>
<td></td>
</tr>
<tr>
<td>3.18</td>
<td>Texel location wrap mode application.</td>
<td></td>
</tr>
<tr>
<td>3.19</td>
<td>Depth texture comparison functions.</td>
<td></td>
</tr>
<tr>
<td>3.20</td>
<td>Correspondence of filtered texture components to texture source components.</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>RGB and Alpha blend equations.</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>Blending functions.</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Arguments to <code>LogicOp</code> and their corresponding operations.</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>Buffer selection for the default framebuffer</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>Buffer selection for a framebuffer object</td>
<td></td>
</tr>
<tr>
<td>4.6</td>
<td><code>DrawBuffers</code> buffer selection for the default framebuffer</td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td><code>PixelStore</code> parameters.</td>
<td></td>
</tr>
<tr>
<td>4.8</td>
<td><code>ReadPixels</code> index masks.</td>
<td></td>
</tr>
<tr>
<td>4.9</td>
<td><code>ReadPixels</code> GL data types and reversed component conversion formulas.</td>
<td></td>
</tr>
<tr>
<td>4.10</td>
<td>Correspondence of renderbuffer sized to base internal formats.</td>
<td></td>
</tr>
<tr>
<td>4.11</td>
<td>Framebuffer attachment points.</td>
<td></td>
</tr>
<tr>
<td>4.12</td>
<td>Layer numbers for cube map texture faces.</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Initial properties of a sync object created with <code>FenceSync</code>.</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Hint targets and descriptions</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Texture, table, and filter return values.</td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>Context profile bits</td>
<td></td>
</tr>
<tr>
<td>6.3</td>
<td>State Variable Types</td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td>Vertex Array Object State (cont.)</td>
<td></td>
</tr>
<tr>
<td>6.5</td>
<td>Vertex Array Object State (cont.)</td>
<td></td>
</tr>
<tr>
<td>6.6</td>
<td>Vertex Array Data (not in Vertex Array objects)</td>
<td></td>
</tr>
<tr>
<td>6.7</td>
<td>Buffer Object State</td>
<td></td>
</tr>
<tr>
<td>6.8</td>
<td>Transformation state</td>
<td></td>
</tr>
<tr>
<td>6.9</td>
<td>Coloring</td>
<td></td>
</tr>
<tr>
<td>6.10</td>
<td>Rasterization</td>
<td></td>
</tr>
<tr>
<td>6.11</td>
<td>Rasterization (cont.)</td>
<td></td>
</tr>
<tr>
<td>6.12</td>
<td>Multisampling</td>
<td></td>
</tr>
<tr>
<td>6.13</td>
<td>Textures (state per texture unit and binding point)</td>
<td></td>
</tr>
<tr>
<td>6.14</td>
<td>Textures (state per texture unit and binding point)(cont.)</td>
<td></td>
</tr>
<tr>
<td>6.15</td>
<td>Textures (state per texture object)</td>
<td></td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>6.16</td>
<td>Textures (state per texture image)</td>
<td>281</td>
</tr>
<tr>
<td>6.17</td>
<td>Texture Environment and Generation</td>
<td>282</td>
</tr>
<tr>
<td>6.18</td>
<td>Pixel Operations</td>
<td>283</td>
</tr>
<tr>
<td>6.19</td>
<td>Pixel Operations (cont.)</td>
<td>284</td>
</tr>
<tr>
<td>6.20</td>
<td>Framebuffer Control</td>
<td>285</td>
</tr>
<tr>
<td>6.21</td>
<td>Framebuffer (state per target binding point)</td>
<td>286</td>
</tr>
<tr>
<td>6.22</td>
<td>Framebuffer (state per framebuffer object)</td>
<td>287</td>
</tr>
<tr>
<td>6.23</td>
<td>Framebuffer (state per attachment point)</td>
<td>288</td>
</tr>
<tr>
<td>6.24</td>
<td>Renderbuffer (state per target and binding point)</td>
<td>289</td>
</tr>
<tr>
<td>6.25</td>
<td>Renderbuffer (state per renderbuffer object)</td>
<td>290</td>
</tr>
<tr>
<td>6.26</td>
<td>Pixels</td>
<td>291</td>
</tr>
<tr>
<td>6.27</td>
<td>Shader Object State</td>
<td>292</td>
</tr>
<tr>
<td>6.28</td>
<td>Program Object State</td>
<td>293</td>
</tr>
<tr>
<td>6.29</td>
<td>Program Object State (cont.)</td>
<td>294</td>
</tr>
<tr>
<td>6.30</td>
<td>Program Object State (cont.)</td>
<td>295</td>
</tr>
<tr>
<td>6.31</td>
<td>Program Object State (cont.)</td>
<td>296</td>
</tr>
<tr>
<td>6.32</td>
<td>Vertex and Geometry Shader State</td>
<td>297</td>
</tr>
<tr>
<td>6.33</td>
<td>Query Object State</td>
<td>298</td>
</tr>
<tr>
<td>6.34</td>
<td>Transform Feedback State</td>
<td>299</td>
</tr>
<tr>
<td>6.35</td>
<td>Sync (state per sync object)</td>
<td>300</td>
</tr>
<tr>
<td>6.36</td>
<td>Hints</td>
<td>301</td>
</tr>
<tr>
<td>6.37</td>
<td>Implementation Dependent Values</td>
<td>302</td>
</tr>
<tr>
<td>6.38</td>
<td>Implementation Dependent Values (cont.)</td>
<td>303</td>
</tr>
<tr>
<td>6.39</td>
<td>Implementation Dependent Values (cont.)</td>
<td>304</td>
</tr>
<tr>
<td>6.40</td>
<td>Implementation Dependent Version and Extension Support</td>
<td>305</td>
</tr>
<tr>
<td>6.41</td>
<td>Implementation Dependent Vertex Shader Limits</td>
<td>306</td>
</tr>
<tr>
<td>6.42</td>
<td>Implementation Dependent Geometry Shader Limits</td>
<td>307</td>
</tr>
<tr>
<td>6.43</td>
<td>Implementation Dependent Fragment Processing Limits</td>
<td>308</td>
</tr>
<tr>
<td>6.44</td>
<td>Implementation Dependent Aggregate Shader Limits</td>
<td>309</td>
</tr>
<tr>
<td>6.24</td>
<td>The minimum value for each stage is $\text{MAX}<em>{\text{stage}}</em>{-}\text{UNIFORM_BLOCKS} \times \text{MAX}<em>{\text{stage}}</em>{\text{UNIFORM_BLOCK_SIZE}} + \text{MAX}<em>{\text{stage}}</em>{\text{UNIFORM_COMPONENTS}}$</td>
<td>310</td>
</tr>
<tr>
<td>6.45</td>
<td>Implementation Dependent Values (cont.)</td>
<td>311</td>
</tr>
<tr>
<td>6.46</td>
<td>Framebuffer Dependent Values</td>
<td>312</td>
</tr>
<tr>
<td>6.47</td>
<td>Miscellaneous</td>
<td>313</td>
</tr>
<tr>
<td>F.1</td>
<td>New token names</td>
<td>337</td>
</tr>
<tr>
<td>H.1</td>
<td>New token names</td>
<td>349</td>
</tr>
</tbody>
</table>
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 Comments on edits to the OpenGL 3.1 Specification

Changes in the most recent draft are typeset in magenta.
Older changes are typeset in purple.
Editorial comments and questions are typeset in blue.

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 a GL context and associate it with the window. Once a GL 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

GL 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 3.9). These sections may include references to concepts and terms (such as shading language variable types) defined in the companion document.

OpenGL 3.2 implementations are guaranteed to support versions 1.40 and 1.50 of the OpenGL Shading Language. All references to sections of that specification refer to version 1.50. The supported version of the shading language may be queried as described in section 6.1.4.

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. 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 I).
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
2.1. **OPENGL FUNDAMENTALS**

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.
2.1. OPENGL FUNDAMENTALS

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 Floating-Point Computation

The GL must perform a number of floating-point operations during the course of its operation. In some cases, the representation and/or precision of such operations is defined or limited; by the OpenGL Shading Language Specification for operations in shaders, and in some cases implicitly limited by the specified format of vertex, texture, or renderbuffer data consumed by the GL. Otherwise, the representation of such floating-point numbers, and the details of how operations on them are performed, is not specified. 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 of a floating-point number used to represent positional, normal, or texture coordinates must be at least $2^{32}$; the maximum representable magnitude for colors must be at least $2^{10}$. The maximum representable magnitude for all other floating-point values must be at least $2^{32}$.

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 $\frac{1}{0}$. 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.

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 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:

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

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

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

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

$$M = N \mod 1024.$$

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 Inf or 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.

### 2.1.3 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
2.1.1 OPENGL FUNDAMENTALS

The value \( V \) of an unsigned 10-bit floating-point number is determined by the following:

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

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.

### 2.1.4 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:

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

### 2.1.5 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 3.8.1), $b$ 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.
2.1. OPENGL FUNDAMENTALS

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 \(f\) may be performed in two ways:

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

(2.2)

In this case the full range of the representation is used, so that \(-2^{b-1}\) corresponds to -1.0 and \(2^{b-1} - 1\) corresponds to 1.0. For example, if \(b = 8\), then the integer value -128 corresponds to -1.0 and the value 127 corresponds to 1.0. Note that it is not possible to exactly express 0 in this representation. In general, this representation is used for signed normalized fixed-point parameters in GL commands, such as vertex attribute values.

Alternatively, conversion may be performed using

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

(2.3)

In this case 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 exactly expressed in this representation, one value (-128 in the example) is outside the representable range, and must be clamped before use. In general, this representation is used for signed normalized fixed-point texture or framebuffer values.

Everywhere that signed normalized fixed-point values are converted, the equation used is specified.
Conversion from Floating-Point to Normalized Fixed-Point

The conversion from a floating-point value \( f \) to the corresponding unsigned normalized fixed-point value \( c \) is defined by first clamping \( f \) to the range \([0, 1]\), then computing

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

(2.4)

\( f' \) 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 normalized fixed-point value \( c \) may be performed in two ways, both beginning by clamping \( f \) to the range \([-1, 1]\):

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

(2.5)

In general, this conversion is used when querying floating-point state (see section 6) and returning integers.

Alternatively, conversion may be performed using

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

(2.6)

In general, this conversion is used when specifying signed normalized fixed-point texture or framebuffer values.

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

Everywhere that floating-point values are converted to signed normalized fixed-point, the equation used is specified.

2.2 GL State

The GL maintains considerable state. This document enumerates each state variable and describes how each variable can be changed. For purposes of discussion, state variables are categorized somewhat arbitrarily by their function. Although we 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* followed, depending on the particular command, by up to 4 characters. The first character indicates the number of values of the indicated type that must be presented to the command. The second character or character pair indicates the specific type of the arguments: 8-bit integer, 16-bit integer, 32-bit integer, single-precision floating-point, or double-precision floating-point. The final character, if present, is \( v \), indicating that the command takes a pointer to an array (a vector) of values rather than a series of individual arguments. Two specific examples are:

```c
void Uniform4f(int location, float v0, float v1,
                float v2, float v3);
```

and

```c
void GetFloatv(enum value, float *data);
```
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>
</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.

These examples show the ANSI C declarations for these commands. In general, a command declaration has the form\(^1\)

\[
\text{rtype Name}\{\epsilon1234}\{\epsilon b s i i64 f d ub us ui}\{\epsilon v}\) \(\text{args ,} T \text{arg1 , \ldots ,} T \text{argN [ , args ] ;}
\]

\text{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. \(\epsilon\) indicates no type descriptor. The arguments enclosed in brackets (\text{args ,} and \text{[ , args]}\) may or may not be present. The \(N\) arguments \text{arg1} through \text{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 \text{arg1} is present and it is an array of \(N\) values of the indicated type. Finally, we indicate an unsigned type by the shorthand of prepending a \(u\) to the beginning of the type name (so that, for instance, unsigned byte is abbreviated ubyte).

For example,

\[
\text{void Uniform}\{1234}\{\epsilon f d ub us ui}\{\epsilon v}\) \(\text{int location ,} T \text{value ;}
\]

indicates the eight declarations

\[
\text{void Uniform1i( int location , int value ) ;}
\]

\(^1\)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.
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 v2 );
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.

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 viewing 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. Objects such as curved surfaces, for instance, may be transformed before they are converted to polygons.
### 2.4. BASIC GL OPERATION

<table>
<thead>
<tr>
<th>GL Type</th>
<th>Minimum Bit Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1</td>
<td>Boolean</td>
</tr>
<tr>
<td>byte</td>
<td>8</td>
<td>Signed twos 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 twos 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 twos complement binary integer</td>
</tr>
<tr>
<td>uint</td>
<td>32</td>
<td>Unsigned binary integer</td>
</tr>
<tr>
<td>int64</td>
<td>64</td>
<td>Signed twos 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 twos 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 5.2)</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 may use more bits than the number indicated in the table to represent a GL type. Correct interpretation of integer values outside the minimum range is not required, however.

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.
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. 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.
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 2.8.2. There are seven geometric objects that are drawn this way: points, line segment strips, line segment loops, separated line segments, triangle strips, triangle fans, and separated triangles.

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 varying 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
2.6. PRIMITIVES AND VERTICES

Figure 2.2. Vertex processing and primitive assembly.
is clipped to a viewing volume. This may alter the primitive by altering vertex coordinates and varying vertex shader outputs. In the case of line and polygon primitives, clipping may insert new vertices into the primitive. The vertices defining a primitive to be rasterized have varying outputs 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 2.8.2. 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 \( i \)th vertex (for \( i > 1 \)) specifies the beginning of the \( i \)th segment and the end of the \( i - 1 \)st. 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
2.6. PRIMITIVES AND VERTICES

(a) (b) (c)

Figure 2.3. (a) A triangle strip. (b) A triangle fan. (c) Independent triangles. The numbers give the sequencing of the vertices in order within the vertex arrays. Note that in (a) and (b) triangle edge ordering is determined by the first triangle, while in (c) the order of each triangle’s edges is independent of the other triangles.

odd or even (a segment start or end).

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.
2.6. PRIMITIVES AND VERTICES

Separate Triangles
Separate triangles are specified with mode TRIANGLES. In this case, the \( 3i + 1 \)st, \( 3i + 2 \)nd, and \( 3i + 3 \)rd 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 2.12). 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).

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.
2.6. PRIMITIVES AND VERTICES

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
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 \texttt{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 \texttt{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 \( i + 1 \)st vertices, respectively (see figure 2.5).

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
2.7. VERTEX SPECIFICATION

Figure 2.6. Triangle strips with adjacency. 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.

dot 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).

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 3.9.2).

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,
### 2.7. VERTEX SPECIFICATION

**Primitive Vertices**

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

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}(uint index, T values);
void VertexAttrib4{{bsfd ub us ui}}(uint index, T values);
void VertexAttrib4Nub(uint index, T values);
void VertexAttrib4N{{bsi ub us ui}}(uint index, T values);
```

The `VertexAttrib4N*` commands specify fixed-point values that are converted to a normalized \([0, 1]\) or \([-1, 1]\) range as described in equations 2.1 and 2.2, respectively, while the other 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.

The resulting attribute values are undefined if the base type of the shader attribute at slot `index` is not floating-point (e.g. is signed or unsigned integer). To load current values of a generic shader attribute declared as a signed or unsigned scalar or vector, use the commands

```c
void VertexAttribI{1234}{i ui}(uint index, T values);
void VertexAttribI{1234}{i ui}v(uint index, T values);
void VertexAttribI4{bs ubus}v(uint index, T values);
```

These commands specify values that are extended to full signed or unsigned integers, then loaded into the generic attribute at slot `index` in the same fashion as described above.

The resulting attribute values are undefined if the base type of the shader attribute at slot `index` is floating-point; if the base type is integer and unsigned integer values are supplied (the `VertexAttribI*ui`, `VertexAttribI*us`, and `VertexAttribI*ub` commands); or if the base type is unsigned integer and signed integer values are supplied (the `VertexAttribI*i`, `VertexAttribI*s`, and `VertexAttribI*b` commands).

The error `INVALID_VALUE` is generated by `VertexAttrib*` if `index` is greater than or equal to `MAX_VERTEX_ATTRIBS`.

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 is 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);
```

OpenGL 3.2 (Core Profile) - July 24, 2009
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.

```c
void VertexAttribIPointer( 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`, `FLOAT`, `HALF_FLOAT`, and `DOUBLE` indicate types `byte`, `short`, `int`, `float`, `half`, and `double`, respectively; and the values `UNSIGNED_BYTE`, `UNSIGNED_SHORT`, and `UNSIGNED_INT` indicate types `ubyte`, `ushort`, and `uint`, respectively. The error `INVALID_VALUE` is generated if `size` is specified with a value other than that indicated in the table, if `size` is BGRA and `type` is not `UNSIGNED_BYTE`, or by `VertexAttribPointer` if `size` is BGRA and `normalized` is FALSE.

The `index` parameter in the `VertexAttribPointer` and `VertexAttribIPointer` 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.

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 $i$th 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 command, pointer specifies the offset within a buffer of the first value of the first element of the array being specified.

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

```c
void EnableVertexAttribArray( uint index);
void DisableVertexAttribArray( uint index);
```

where index identifies the generic vertex attribute array to enable or disable. The error INVALID_VALUE is generated if index is greater than or equal to MAX_VERTEX_ATTRIBS.

### 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
2.8. VERTEX ARRAYS

```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 2.8.2 is used, the primitive restart comparison occurs before the `basevertex` offset is added to the array index.

### 2.8.2 Drawing Commands

The command

```c
void DrawArrays( enum mode, int first, sizei count );
```

constructs a sequence of geometric primitives by transferring elements `first` through `first + count - 1` of each enabled array to the GL. `mode` specifies what kind of primitives are constructed, as defined in section 2.6.1. 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).

If an array corresponding to a generic attribute required by a vertex is enabled, the corresponding current generic attribute value is undefined after the execution of `DrawArrays`.

Specifying `first < 0` results in undefined behavior. Generating the error `INVALID_VALUE` is recommended in this case.

The command

```c
void MultiDrawArrays( enum mode, int *first, sizei *count, sizei primcount );
```

behaves identically to `DrawArrays` except that `primcount` separate ranges of elements are specified instead. It has the same effect as:

OpenGL 3.2 (Core Profile) - July 24, 2009
for (i = 0; i < primcount; i++) {
    if (count[i] > 0)
        DrawArrays(mode, first[i], count[i]);
}

The command

    void DrawElements( enum mode, sizei count, enum type,
                    void *indices );

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 DrawElements will be taken from element indices[i] of each enabled 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. 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).

If an array corresponding to a generic attribute required by a vertex is enabled, the corresponding current generic attribute value is undefined after the execution of DrawElements.

The command

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

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

    for (i = 0; i < primcount; i++) {
        if (count[i]) > 0)
            DrawElements(mode, count[i], type, indices[i]);
    }

The command

    void DrawRangeElements( enum mode, uint start,
                        uint end, sizei count, enum type, void *indices );
is a restricted form of DrawElements. \( \text{mode}, \text{count}, \text{type}, \) and \( \text{indices} \) match the corresponding arguments to DrawElements, with the additional constraint that all index values identified by \( \text{indices} \) must lie between \( \text{start} \) and \( \text{end} \) inclusive.

Implementations denote recommended maximum amounts of vertex and index data, which may be queried by calling GetIntegerv with the symbolic constants \text{MAX_ELEMENTS_VERTICES} and \text{MAX_ELEMENTS_INDICES}. If \( \text{end} - \text{start} + 1 \) is greater than the value of \text{MAX_ELEMENTS_VERTICES}, or if \( \text{count} \) is greater than the value of \text{MAX_ELEMENTS_INDICES}, then the call may operate at reduced performance. There is no requirement that all vertices in the range \([\text{start}, \text{end}]\) be referenced. However, the implementation may partially process unused vertices, reducing performance from what could be achieved with an optimal index set.

The error \text{INVALID_VALUE} is generated if \( \text{end} < \text{start} \). Invalid \( \text{mode}, \text{count}, \) or \( \text{type} \) 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 to lie outside the range \([\text{start}, \text{end}]\), but implementations are not required to check for this. Such indices will cause implementation-dependent behavior.

The internal counter \text{instanceID} is a 32-bit integer value which may be read by a vertex shader as \text{gl_InstanceID}, as described in section 2.11.7. The value of this counter is always zero, except as noted below.

The command

\[
\text{void DrawArraysInstanced( } \text{enum mode, int first, sizei count, sizei primcount) };
\]

behaves identically to \text{DrawArrays} except that \( \text{primcount} \) instances of the range of elements are executed and the value of \text{instanceID} advances for each iteration. It has the same effect as:

\[
\text{if (mode or count is invalid) } \text{generate appropriate error}
\]
\[
\text{else } \{
\text{for (int i = 0; i < primcount; i++) } \{
\text{instanceID} = i;
\text{DrawArrays} ( \text{mode, first, count});
\}
\text{instanceID} = 0;
\}
\]

The command

\[
\text{void DrawElementsInstanced( } \text{enum mode, sizei count, enum type, const void *indices, sizei primcount) };
\]
behaves identically to DrawElements except that \texttt{primcount} instances of the set of elements are executed, and the value of \texttt{instanceID} advances for each iteration. It has the same effect as:

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

The commands

\begin{verbatim}
void \textbf{DrawElementsBaseVertex}(\texttt{enum mode, sizei count,}
\texttt{enum type, void *indices, int basevertex});
void \textbf{DrawRangeElementsBaseVertex}(\texttt{enum mode,}
\texttt{uint start, uint end, sizei count, enum type,}
\texttt{void *indices, int basevertex});
void \textbf{DrawElementsInstancedBaseVertex}(\texttt{enum mode,}
\texttt{sizei count, enum type, const void *indices,}
\texttt{sizei primcount, int basevertex});
\end{verbatim}

are equivalent to the commands with the same base name (without the \texttt{BaseVertex} suffix), except that the \texttt{i}th element transferred by the corresponding draw call will be taken from element \texttt{indices[i] + basevertex} of each enabled array. If the resulting value is larger than the maximum value representable by \texttt{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 \textbf{DrawRangeElementsBaseVertex}, the index values must lie between \texttt{start} and \texttt{end} inclusive, prior to adding the \texttt{basevertex} offset. Index values lying outside the range \texttt{[start, end]} are treated in the same way as DrawRangeElements.

The command

\begin{verbatim}
void \textbf{MultiDrawElementsBaseVertex}(\texttt{enum mode,}
\texttt{sizei *count, enum type, void **indices,}
\texttt{sizei primcount, int *basevertex});
\end{verbatim}
behaves identically to DrawElementsBaseVertex, except that primcount separate lists of elements are specified instead. It has the same effect as:

```c
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 client 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, 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, and the restart index is zero.

### 2.9 Buffer Objects

Vertex array data are stored in high-performance server memory. GL buffer objects provide a mechanism that clients can use 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 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>COPY_READ_BUFFER</td>
<td>Buffer copy source</td>
<td>2.9.5</td>
</tr>
<tr>
<td>COPY_WRITE_BUFFER</td>
<td>Buffer copy destination</td>
<td>2.9.5</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>3.8.5</td>
</tr>
<tr>
<td>TRANSFORM_FEEDBACK_BUFFER</td>
<td>Transform feedback buffer</td>
<td>2.16</td>
</tr>
<tr>
<td>UNIFORM_BUFFER</td>
<td>Uniform block storage</td>
<td>2.11.4</td>
</tr>
</tbody>
</table>

Table 2.6: 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 2.6. 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 the state values listed in table 2.7.

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.

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 and other threads are not affected, but
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,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>STREAM_COPY, STATIC_DRAW,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>STREAM_READ, STATIC_COPY,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DYNAMIC_DRAW, DYNAMIC_READ,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DYNAMIC_COPY</td>
</tr>
<tr>
<td>BUFFER_ACCESS</td>
<td>enum</td>
<td>READ_WRITE</td>
<td>READONLY, WRITEONLY, 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 2.7: Buffer object parameters and their values.

attempting to use a deleted buffer in another thread produces undefined results, including but not limited to possible GL errors and rendering corruption. Using a deleted buffer in another context or thread may not, however, result in program termination.

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 TRANSFORM_FEEDBACK_BUFFER or UNIFORM_BUFFER. Additional language specific to each target is included in sections referred to for each target in table 2.6.

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
object named by buffer to both the general binding point, and to the binding point in the array given by index. 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 or if offset + size is greater than the value of BUFFER_SIZE. Additional errors may be generated if offset violates target-specific alignment requirements.

BindBufferBase is equivalent to calling BindBufferRange with offset zero and size equal to the size of buffer.

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 2.6, 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.
### 2.9. BUFFER OBJECTS

<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 2.8: Buffer object initial state.

**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 2.8.

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 3.2 (Core Profile) - July 24, 2009
with \texttt{target} set to one of the targets listed in table 2.6. \texttt{offset} and \texttt{size} indicate the range of data in the buffer object that is to be replaced, in terms of basic machine units. \texttt{data} specifies a region of client memory \texttt{size} basic machine units in length, containing the data that replace the specified buffer range. An \texttt{INVALID\_VALUE} error is generated if \texttt{offset} or \texttt{size} is less than zero or if \texttt{offset + size} is greater than the value of \texttt{BUFFER\_SIZE}. An \texttt{INVALID\_OPERATION} error is generated if any part of the specified buffer range is mapped with \texttt{MapBufferRange} or \texttt{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 \texttt{target} set to one of the targets listed in table 2.6. \texttt{offset} and \texttt{length} indicate the range of data in the buffer object that is to be mapped, in terms of basic machine units. \texttt{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 \texttt{access}:

- \texttt{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 a mapping which excludes this flag, but the result is undefined and system errors (possibly including program termination) may occur.

- \texttt{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.

Pointer values returned by \texttt{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.
2.9. BUFFER OBJECTS

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 each subrange must be explicitly flushed by calling FlushMappedBufferRange. No GL error is set if a subrange of the mapping is modified and not flushed, but data within the corresponding subrange of the buffer are undefined. This flag may only be used in conjunction with 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 2.9.
2.9. BUFFER OBJECTS

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFFER_ACCESS</td>
<td>Depends on access¹</td>
</tr>
<tr>
<td>BUFFER_ACCESS_FLAGS</td>
<td>access</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>offset</td>
</tr>
<tr>
<td>BUFFER_MAP_LENGTH</td>
<td>length</td>
</tr>
</tbody>
</table>

Table 2.9: Buffer object state set by MapBufferRange.

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.

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:

- 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

and mbaccess is the value of the access enum parameter passed to MapBuffer. 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 2.6. 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. FlushMappedBufferRange may be called multiple times to indicate distinct subranges 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 2.6. 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 2.8.

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.

2.9.4 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,
```

OpenGL 3.2 (Core Profile) - July 24, 2009
with readtarget and writetarget each set to one of the targets listed in table 2.6. While any of these targets may be used, the COPY_READ_BUFFER and COPY_WRITE_BUFFER 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 2.8.2 operate as previously defined, where data for enabled generic attribute 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.
2.9. BUFFER OBJECTS

If any enabled array’s buffer binding is zero when DrawArrays or one of the other drawing commands defined in section 2.8.2 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 2.8.2).

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 DrawElementsInstancedBaseVertex 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 indices 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 corresponding binding points.

2.9.8 Buffer Object State

The state required to support buffer objects consists of binding names for the array buffer, element buffer, pixel unpack buffer, and pixel pack buffer. 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 command

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

returns \( n \) previous unused vertex array object names in \( \text{arrays} \). These names are marked as used, for the purposes of \textit{GenVertexArrays} only, but they acquire array state only when they are first bound, just as if they were unused.

Vertex array objects are deleted by calling

```c
void DeleteVertexArrays( sizei n, const uint *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 the default vertex array becomes current. 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 \textit{GenVertexArrays} with the command

```c
void BindVertexArray( uint array );
```

\( \text{array} \) is the vertex array object name. The resulting vertex array object is a new state vector, comprising all the state values listed in tables 6.4 and 6.5.

\textit{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 \textit{VertexAttribPointer} and \textit{EnableVertexAttribArray}; all commands which draw from vertex arrays, such as \textit{DrawArrays} and \textit{DrawElements}; and all queries of vertex array state (see chapter 6).

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

An \texttt{INVALID_OPERATION} error is generated if any of the \texttt{*Pointer} commands specifying the location and organization of vertex array data are called while \( \text{zero} \)
is bound to the \texttt{ARRAY\_BUFFER} buffer object binding point, and the \texttt{pointer} argument is not \texttt{NULL}.

\subsection{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 \textit{shader object} and then \textit{compiled}. One or more vertex shader objects are then attached to a \textit{program object}. A program object is then \textit{linked}, which generates executable code from all the compiled shader objects attached to the program. When a linked program object is used as the current program object, the executable code for the vertex shaders it contains is used to process vertices.

In addition to vertex shaders, \textit{geometry shaders} and \textit{fragment shaders} can be created, compiled, and linked into program objects. Geometry shaders affect the processing of primitives assembled from vertices (see section 2.12). Fragment shaders affect the processing of fragments during rasterization (see section 3.9). A single program object can contain all of vertex, geometry, and fragment shaders.

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

A vertex shader can reference a number of variables as it executes. \textit{Vertex attributes} are the per-vertex values specified in section 2.7. \textit{Uniforms} are per-program variables that are constant during program execution. \textit{Samplers} are a special form of uniform used for texturing (section 3.8). \textit{Varying 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.

\subsubsection{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 \textit{shader objects}.

\footnote{This error makes it impossible to create a vertex array object containing client array pointers, while still allowing buffer objects to be unbound.}

OpenGL 3.2 (Core Profile) - July 24, 2009
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

```c
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

```c
void ShaderSource( uint shader, sizei count, const char **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

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

Each shader object has a boolean status, `COMPILE_STATUS`, that is modified as a result of compilation. This status can be queried with `GetShaderiv` (see section 6.1.10). 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.
2.11. VERTEX SHADERS

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 6.1.10).

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 6.1.10). DeleteShader will silently ignore the value zero.

### 2.11.2 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 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 6.1.10). 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. Linking will also fail if one or more of the shader objects, attached to program are not compiled successfully, or if more active uniform or active sampler variables are used in program than allowed (see sections 2.11.4, 2.11.5, and 2.12.3).

Linking will also fail if the program object contains objects to form a geometry shader (see section 2.12), and

- the program 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.

Each program object has an information log that is overwritten as a result of a link operation. This information log can be queried with GetProgramInfoLog to obtain more information about the link operation or the validation information (see section 6.1.10).

If a valid executable is created, it can be made part of the current rendering state with the command
void UseProgram( uint program );

This command will install the executable code as part of current rendering state if the program object program contains valid executable code, i.e. has been linked successfully. If UseProgram is called with program set to 0, then the current rendering state refers to an invalid program object, and the results of vertex and fragment shader execution are undefined. However, this is not an error. If program has not been successfully linked, the error 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.

If the program object that is in use is re-linked successfully, the LinkProgram command will install the generated executable code as part of the current rendering state if the specified program object was already in use as a result of a previous call to UseProgram.

If that program object that is in use is re-linked unsuccessfully, the link status will be set to FALSE, but existing executable and associated state will remain part of the current rendering state until a subsequent call to UseProgram removes it from use. After such a program is removed from use, it can not be made part of the current rendering state until it is successfully re-linked.

Program objects can be deleted with the command

void DeleteProgram( uint program );

If program is not the current program for any GL context, it is deleted immediately. Otherwise, program is flagged for deletion and will be deleted when it is no longer the current program for any context. When a program object is deleted, all shader objects attached to it are detached. DeleteProgram will silently ignore the value zero.

### 2.11.3 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 as a float, vec2, vec3 or vec4 is bound to a generic attribute index i, its value(s) are taken from the x, (x, y), (x, y, z), or (x, y, z, w) components, respectively, of the generic attribute i. When an attribute
2.11. VERTEX SHADERS

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), 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).

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 against the limit. 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 \texttt{MAX_VERTEX_ATTRIBS}.

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 \texttt{index}. An \texttt{index} of 0 selects the first active attribute, and an \texttt{index} of \texttt{ACTIVE_ATTRIBUTES} − 1 selects the last active attribute. The value of \texttt{ACTIVE_ATTRIBUTES} can be queried with \texttt{GetProgramiv} (see section 6.1.10). If \texttt{index} is greater than or equal to \texttt{ACTIVE_ATTRIBUTES}, the error \texttt{INVALID_VALUE} is generated. Note that \texttt{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 \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 attributes exceeded the limit.

The name of the selected attribute is returned as a null-terminated string in \texttt{name}. The actual number of characters written into \texttt{name}, 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{name}, including the null terminator, is specified by \texttt{bufSize}. The returned attribute name must be the name of a generic attribute. The length of the longest attribute name in \texttt{program} is given by
2.11. VERTEX SHADERS

ACTIVE_ATTRIBUTE_MAX_LENGTH, which can be queried with GetProgramiv (see section 6.1.10).

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 FLOAT, FLOAT_VEC2, FLOAT_VEC3, FLOAT_VEC4, FLOAT_MAT2, FLOAT_MAT3, FLOAT_MAT4, FLOAT_MAT2x3, FLOAT_MAT2x4, FLOAT_MAT3x2, FLOAT_MAT3x4, FLOAT_MAT4x2, FLOAT_MAT4x3, INT, INT_VEC2, INT_VEC3, INT_VEC4, UNSIGNED_INT, UNSIGNED_INT_VEC2, UNSIGNED_INT_VEC3, or UNSIGNED_INT_VEC4.

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 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

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

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 through BindAttribLocation 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 **MAX_VERTEX_ATTRIBS**). **LinkProgram** will fail if the attribute bindings assigned by **BindAttribLocation** do not leave enough space to assign a location for an active matrix attribute, which requires multiple contiguous generic attributes.

**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.

### 2.11.4 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. 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. 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 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. 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 in the default uniform block accessed by a vertex shader is specified by the value of the implementation-dependent constant MAX_VERTEX_UNIFORM_COMPONENTS. The total amount of combined 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, integer, or boolean values that can be held in uniform variable storage for a vertex shader. 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 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 uniforms belonging 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 BufferData, 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

\[
\text{int GetUniformLocation}(\text{uint program, const char }*\text{name});
\]

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, 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]".

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 );
```

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.

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.
2.11. VERTEX SHADERS

successfully. The link could have failed because the number of active uniforms exceeded the limit.

uniformBlockIndex must be an active uniform block index of program, in the range zero to the value of ACTIVE_UNIFORM_BLOCKS - 1. The value of ACTIVE_UNIFORM_BLOCKS can be queried with GetProgramiv (see section 6.1.10). 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

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. The link could have failed because the number of active uniforms exceeded the limit.

uniformBlockIndex is an active uniform block index of 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 last selected by the uniform block specified by uniformBlockIndex for program is returned. If no uniform block has been previously specified, zero 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

OpenGL 3.2 (Core Profile) - July 24, 2009
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 2.11.4).

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 `GetIntegeriv` 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_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, geometry, or fragment programming stages of `program`, respectively, is returned.

Each active uniform, whether in a named uniform block or in the default block, is assigned an index when a program is linked. Indices are assigned in consecutive order, beginning with zero. 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 **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` name strings identifying the uniform names to be queried for indices. For each name string in `uniformNames`, the
index assigned to the active uniform of that name will be written to the corresponding element of \texttt{uniformIndices}. If a string in \texttt{uniformNames} is not the name of an active uniform, 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 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} - 1. 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 uniform variable, declared in a shader, 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{GetUniformLocation}, for default uniform block uniform names, or \texttt{GetUniformIndices}, for named uniform block uniform names.

Information about active uniforms can be obtained by calling either

\begin{verbatim}
void GetActiveUniform( uint program, uint index,
    sizei bufSize, sizei *length, int *size, enum *type,
    char *name );
\end{verbatim}
2.11. VERTEX SHADERS

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 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** - 1 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**.

Each uniform variable, declared in a shader, 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 **GetUniformLocation**, for default uniform block uniform names, or **GetUniformIndices**, for named uniform block uniform names.

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 2.10, 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
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</th>
<th>Token</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOAT</td>
<td></td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>FLOAT_VEC2</td>
<td></td>
<td>vec2</td>
<td></td>
</tr>
<tr>
<td>FLOAT_VEC3</td>
<td></td>
<td>vec3</td>
<td></td>
</tr>
<tr>
<td>FLOAT_VEC4</td>
<td></td>
<td>vec4</td>
<td></td>
</tr>
<tr>
<td>INT</td>
<td></td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>INT_VEC2</td>
<td></td>
<td>ivec2</td>
<td></td>
</tr>
<tr>
<td>INT_VEC3</td>
<td></td>
<td>ivec3</td>
<td></td>
</tr>
<tr>
<td>INT_VEC4</td>
<td></td>
<td>ivec4</td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT</td>
<td></td>
<td>unsigned int</td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT_VEC2</td>
<td></td>
<td>uvec2</td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT_VEC3</td>
<td></td>
<td>uvec3</td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT_VEC4</td>
<td></td>
<td>uvec4</td>
<td></td>
</tr>
<tr>
<td>BOOL</td>
<td></td>
<td>bool</td>
<td></td>
</tr>
<tr>
<td>BOOL_VEC2</td>
<td></td>
<td>bvec2</td>
<td></td>
</tr>
<tr>
<td>BOOL_VEC3</td>
<td></td>
<td>bvec3</td>
<td></td>
</tr>
<tr>
<td>BOOL_VEC4</td>
<td></td>
<td>bvec4</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT2</td>
<td></td>
<td>mat2</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT3</td>
<td></td>
<td>mat3</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT4</td>
<td></td>
<td>mat4</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT2x3</td>
<td></td>
<td>mat2x3</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT2x4</td>
<td></td>
<td>mat2x4</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT3x2</td>
<td></td>
<td>mat3x2</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT3x4</td>
<td></td>
<td>mat3x4</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT4x2</td>
<td></td>
<td>mat4x2</td>
<td></td>
</tr>
<tr>
<td>FLOAT_MAT4x3</td>
<td></td>
<td>mat4x3</td>
<td></td>
</tr>
<tr>
<td>SAMPLER_1D</td>
<td></td>
<td>sampler1D</td>
<td></td>
</tr>
<tr>
<td>SAMPLER_2D</td>
<td></td>
<td>sampler2D</td>
<td></td>
</tr>
<tr>
<td>SAMPLER_3D</td>
<td></td>
<td>sampler3D</td>
<td></td>
</tr>
<tr>
<td>SAMPLER_CUBE</td>
<td></td>
<td>samplerCube</td>
<td></td>
</tr>
<tr>
<td>SAMPLER_1D_SHADOW</td>
<td></td>
<td>sampler1DShadow</td>
<td></td>
</tr>
<tr>
<td>SAMPLER_2D_SHADOW</td>
<td></td>
<td>sampler2DShadow</td>
<td></td>
</tr>
<tr>
<td>SAMPLER_1D_ARRAY</td>
<td></td>
<td>sampler1DArray</td>
<td></td>
</tr>
</tbody>
</table>

(Continued on next page)
<table>
<thead>
<tr>
<th>Type Name</th>
<th>Token</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAMPLER_2D_ARRAY</td>
<td></td>
<td>sampler2DArray</td>
</tr>
<tr>
<td>SAMPLER_1D_ARRAY_SHADOW</td>
<td></td>
<td>sampler1DArrayShadow</td>
</tr>
<tr>
<td>SAMPLER_2D_ARRAY_SHADOW</td>
<td></td>
<td>sampler2DArrayShadow</td>
</tr>
<tr>
<td>SAMPLER_2D_MULTISAMPLE</td>
<td></td>
<td>sampler2DMS</td>
</tr>
<tr>
<td>SAMPLER_2D_MULTISAMPLE_-ARRAY</td>
<td></td>
<td>sampler2DMSArray</td>
</tr>
<tr>
<td>SAMPLER_CUBE_SHADOW</td>
<td></td>
<td>samplerCubeShadow</td>
</tr>
<tr>
<td>SAMPLER_BUFFER</td>
<td></td>
<td>samplerBuffer</td>
</tr>
<tr>
<td>SAMPLER_2D_RECT</td>
<td></td>
<td>sampler2DRect</td>
</tr>
<tr>
<td>SAMPLER_2D_RECT_SHADOW</td>
<td></td>
<td>sampler2DRectShadow</td>
</tr>
<tr>
<td>INT_SAMPLER_1D</td>
<td></td>
<td>isampler1D</td>
</tr>
<tr>
<td>INT_SAMPLER_2D</td>
<td></td>
<td>isampler2D</td>
</tr>
<tr>
<td>INT_SAMPLER_3D</td>
<td></td>
<td>isampler3D</td>
</tr>
<tr>
<td>INT_SAMPLER_CUBE</td>
<td></td>
<td>isamplerCube</td>
</tr>
<tr>
<td>INT_SAMPLER_1D_ARRAY</td>
<td></td>
<td>isampler1DArray</td>
</tr>
<tr>
<td>INT_SAMPLER_2D_ARRAY</td>
<td></td>
<td>isampler2DArray</td>
</tr>
<tr>
<td>INT_SAMPLER_2D_MULTISAMPLE</td>
<td></td>
<td>isampler2DMS</td>
</tr>
<tr>
<td>INT_SAMPLER_2D_MULTISAMPLE_ARRAY</td>
<td></td>
<td>isampler2DMSArray</td>
</tr>
<tr>
<td>INT_SAMPLER_BUFFER</td>
<td></td>
<td>isamplerBuffer</td>
</tr>
<tr>
<td>INT_SAMPLER_2D_RECT</td>
<td></td>
<td>isampler2DRect</td>
</tr>
<tr>
<td>UNSIGNED_INT_SAMPLER_1D</td>
<td></td>
<td>usampler1D</td>
</tr>
<tr>
<td>UNSIGNED_INT_SAMPLER_2D</td>
<td></td>
<td>usampler2D</td>
</tr>
<tr>
<td>UNSIGNED_INT_SAMPLER_3D</td>
<td></td>
<td>usampler3D</td>
</tr>
<tr>
<td>UNSIGNED_INT_CUBE</td>
<td></td>
<td>usamplerCube</td>
</tr>
<tr>
<td>UNSIGNED_INT_SAMPLER_-1D_ARRAY</td>
<td></td>
<td>usampler1DArray</td>
</tr>
<tr>
<td>UNSIGNED_INT_SAMPLER_-2D_ARRAY</td>
<td></td>
<td>usampler2DArray</td>
</tr>
<tr>
<td>UNSIGNED_INT_SAMPLER_-2D_MULTISAMPLE</td>
<td></td>
<td>usampler2DMS</td>
</tr>
<tr>
<td>UNSIGNED_INT_SAMPLER_-2D_MULTISAMPLE_ARRAY</td>
<td></td>
<td>usampler2DMSArray</td>
</tr>
</tbody>
</table>

(Continued on next page)
2.11. VERTEX SHADERS

<table>
<thead>
<tr>
<th>Type Name Token</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNSIGNED_INT_SAMPLER_-BUFFER</td>
<td>usamplerBuffer</td>
</tr>
<tr>
<td>UNSIGNED_INT_SAMPLER_-2D_RECT</td>
<td>usampler2DRect</td>
</tr>
</tbody>
</table>

Table 2.10: OpenGL Shading Language type tokens returned by `GetActiveUniform` and `GetActiveUniformsiv`, and corresponding shading language keywords declaring each such type.

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 2.10.

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 uniform 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 uniforms in the default uniform block, -1 will be returned.

If `pname` is `UNIFORM_ARRAY_STRIDE`, then an array identifying the stride between elements, in basic machine units, of each of the uniforms specified by the corresponding array of `uniformIndices` is returned. The stride of a uniform...
2.11. VERTEX SHADERS

associated with the default uniform block is -1. Note that this information only makes sense for uniforms that are arrays. For uniforms that are not arrays, but are declared in a named uniform block, an array stride of zero is returned.

If \textit{pname} is \texttt{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 \texttt{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.

### Loading Uniform Variables In The Default Uniform Block

To load values into the uniform variables of the default uniform block of the program object that is currently in use, use the commands

```c
void Uniform\{1234\}\{if\}( int location, \ T value );
void Uniform\{1234\}\{if\}v( int location, sizei count, 
 T value );
void Uniform\{1,2,3,4\}ui( int location, \ T value );
void Uniform\{1,2,3,4\}uiv( int location, sizei count, 
 T value );
void UniformMatrix\{234\}fv( int location, sizei count, 
 boolean transpose, const float *value );
void UniformMatrix\{2x3,3x2,2x4,4x2,3x4,4x3\}fv( 
 int location, sizei count, boolean transpose, const 
 float *value );
```

The given values are loaded into the default uniform block uniform variable location identified by \textit{location}.

The \texttt{Uniform*f\{v\}} commands will load \texttt{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 \texttt{Uniform*i\{v\}} commands will load \texttt{count} sets of one to four integer values into a uniform location defined as a sampler, an integer, an integer vector, an
2.11. VERTEX SHADERS

array of samplers, an array of integers, or an array of integer vectors. Only the \texttt{Uniform1i\{v\}} commands can be used to load sampler values (see below).

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

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

The \texttt{UniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}fv} commands will load \textit{count} \(2 \times 3, 3 \times 2, 2 \times 4, 4 \times 2, 3 \times 4, \text{or} 4 \times 3\) matrices (corresponding to the numbers in the command name) of floating-point values 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, \texttt{UniformMatrix2x4fv} is used to load a matrix consisting of two columns and four rows. If \texttt{transpose} is \texttt{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 \texttt{Uniform*i\{v\}}, \texttt{Uniform*ui\{v\}}, and \texttt{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 \texttt{FALSE} if the input value is 0 or 0.0f, and set to \texttt{TRUE} otherwise. The \texttt{Uniform*} command used must match the size of the uniform, as declared in the shader. For example, to load a uniform declared as a \texttt{bvec2}, any of the \texttt{Uniform2\{if\}i\*} commands may be used. An \texttt{INVALID_OPERATION} error will be generated if an attempt is made to use a non-matching \texttt{Uniform*} command. In this example using \texttt{Uniform1iv} would generate an error.

For all other uniform types the \texttt{Uniform*} command used must match the size and type of the uniform, 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. To load a \(3\times3\) matrix, \texttt{UniformMatrix3fv} must be used. An \texttt{INVALID_OPERATION} error will be generated if an attempt is made to use a non-matching \texttt{Uniform*} command. In this example, using \texttt{Uniform4i\{v\}} would generate an error.

When loading \textit{N} elements starting at an arbitrary position \(k\) in a uniform declared as an array, elements \(k \text{ through } k + \text{N} - 1\) 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.
2.11. VERTEX SHADERS

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 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 program object currently in use.

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 (vertex, geometry, and fragment). 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, geometry, and fragment shaders can be obtained by calling GetIntegerv with pname values of MAX_VERTEX_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 contains multiple shaders with different declarations for the same named uniform block differs between shader, the program will fail to link.
2.11. **VERTEX SHADERS**

**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.
- Vectors with \(N\) elements with basic data types of `bool`, `int`, `uint`, or `float` 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 `matCxR`, 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 `matCxR`, 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 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, \texttt{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 \texttt{layout} qualifier provides shaders with control of the layout of uniforms within a uniform block. When the \texttt{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 \texttt{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 $4N$. 

OpenGL 3.2 (Core Profile) - July 24, 2009
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 sub-structure 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).

**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 \texttt{GetIntegerv} with the constant \texttt{MAX_UNIFORM_BUFFER_BINDINGS}.

Regions of buffer objects are bound as storage for uniform blocks by calling one of the commands \texttt{BindBufferRange} or \texttt{BindBufferBase} (see section 2.9) with
target set to UNIFORM_BUFFER. In addition to the general errors described in section 2.9, `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.

### 2.11.5 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 \( i \) range 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.
2.11. VERTEX SHADERS

lookup. For example, a variable of type \texttt{sampler2D} selects target \texttt{TEXTURE\_2D}
on its texture image unit. Binding of texture objects to targets is done as usual with \texttt{BindTexture}. Selecting the texture image unit to bind to is done as usual with \texttt{ActiveTexture}.

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

Active samplers are samplers actually being used in a program object. The \texttt{LinkProgram} command determines if a sampler is active or not. The \texttt{LinkProgram} 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.

2.11.6 Varying Variables

A vertex shader may define one or more \textit{varying} variables (see the OpenGL Shading Language Specification). Varying variables are outputs of a vertex shader. The OpenGL Shading Language Specification also defines a set of built-in varying and special variables that vertex shaders can write to (see sections 7.1 and 7.6 of the OpenGL Shading Language Specification). These varying variables are either used as the mechanism to communicate values to a geometry shader, if one is active, or to communicate values to the fragment shader and to the fixed-function processing that occurs after vertex shading.

If a geometry shader is not active, the values of all varying and special variables are expected to be interpolated across the primitive being rendered, unless flatshaded. Otherwise the values of all varying and special variables are collected by the primitive assembly stage and passed on to the geometry shader once enough data for one primitive has been collected (see section 2.12).

The number of components (individual scalar numeric values) of varying and special variables that can be written by the vertex shader, whether or not a geometry shader is active, is given by the value of the implementation-dependent constant \texttt{MAX\_VERTEX\_OUTPUT\_COMPONENTS}. Outputs declared as vectors, matrices, and
arrays will all consume multiple components.

When a program is linked, all components of any varying and special variable 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 varying variables 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. Each varying or special variable component used as either a vertex shader output or fragment shader input count 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 varying and special variables 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 one or more vertex or geometry shader output variables to be recorded in transform feedback mode (see section 2.16). When a geometry shader is active (see section 2.12), transform feedback records the values of the selected geometry shader output variables from the emitted vertices. Otherwise, the values of the selected vertex shader output variables are recorded. The values to record are specified with the command

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

`program` specifies the program object. `count` specifies the number of varying variables used for transform feedback. `varyings` is an array of `count` zero-terminated strings specifying the names of the varying variables to use for transform feedback. Varying 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 varying variables when transform feedback is active. The error INVALID_VALUE is generated if `program` is not the name of a program object, or if `bufferMode` is SEPARATE_ATTRIBS and `count` is greater than the value of the implementation-dependent limit MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS.

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 varying variables
2.11. VERTEX SHADERS

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 or geometry shader;
- any variable name specified in the `varyings` array is not declared as an output in the vertex shader (or the geometry shader, if active);
- any two entries in the `varyings` array specify the same varying variable;
- the total number of components to capture in any varying variable in `varyings` is greater than the constant `MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS` and the buffer mode is `SEPARATE_ATTRIBS`; or
- the total number of components to capture is greater than the constant `MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS` and the buffer mode is `INTERLEAVED_ATTRIBS`.

To determine the set of varying 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 varying variable selected by `index`. An `index` of 0 selects the first varying variable specified in the `varyings` array of `TransformFeedbackVaryings`, and an `index` of `TRANSFORM_FEEDBACK_VARYINGS`-1 selects the last such varying variable. The value of `TRANSFORM_FEEDBACK_VARYINGS` can be queried with `GetProgramiv` (see section 6.1.10). If `index` is greater than or equal to `TRANSFORM_FEEDBACK_VARYINGS`, the error `INVALID_VALUE` is generated. The parameter `program` is the name of a program object for which the command `LinkProgram` has been issued in the past. If `program` has not been linked, the error `INVALID_OPERATION` is generated. If a new set of varying variables is specified by `TransformFeedbackVaryings` after a program object has been linked, the information returned by `GetTransformFeedbackVarying` will not reflect those variables until the program is re-linked.

The name of the selected varying 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

OpenGL 3.2 (Core Profile) - July 24, 2009
number of characters that may be written into name, including the null terminator, is specified by bufSize. The length of the longest varying name in program is given by TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH, which can be queried with GetProgramiv (see section 6.1.10).

For the selected varying variable, its type is returned into type. The size of the varying 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 scalar, vector, or matrix attribute types returned by GetActiveAttrib. If an error occurred, the return parameters length, size, type and name will be unmodified. This command will return as much information about the varying variables as possible. If no information is available, length will be set to zero and name will be an empty string. This situation could arise if GetTransformFeedbackVarying is called after a failed link.

2.11.7 Shader Execution

If a successfully linked program object that contains a vertex shader is made current by calling UseProgram, the executable version of the vertex shader is used to process incoming vertex values.

If a geometry shader (see section 2.12) is active, vertices processed by the vertex shader are passed to the geometry shader for further processing. Otherwise, the following operations are applied to vertices processed by the vertex shader:

- Perspective division on clip coordinates (section 2.13).
- Viewport mapping, including depth range scaling (section 2.13.1).
- Flatshading (section 2.18).
- Clipping, including client-defined half-spaces (section 2.19).
- Front face determination (section 3.6.1).
- Generic attribute clipping (section 2.19.1).

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, geometry, or fragment shaders. Also refer to section 3.8 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 directly 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).

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

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 LOD is less than the texture’s base level \((level_{base})\) or greater than the maximum level \((level_{max})\)
- the computed LOD 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 border texel outside the defined extents of the specified LOD, where any of

\[
i < -b_s \quad i \geq w_s - b_s \\
j < -b_s \quad j \geq h_s - b_s \\
k < -b_s \quad k \geq d_s - b_s
\]

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

- the texture being accessed is not complete, as defined in section 3.8.12 (or cube complete for cubemaps).
2.11. VERTEX SHADERS

Multisample Texel Fetches

Multisample texel fetches are similar to standard texel fetches. Since multisample buffers can’t have mipmaps, there is no LOD parameter. Instead, there is an integer parameter which 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\_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 \([\texttt{level\_base}}, \texttt{level\_max}]\), 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 vertex, geometry, or fragment shaders are respectively the values of the implementation-dependent constants \texttt{MAX\_VERTEX\_TEXTURE\_IMAGE\_UNITS}, \texttt{MAX\_GEOMETRY\_TEXTURE\_IMAGE\_UNITS}, and \texttt{MAX\_TEXTURE\_IMAGE\_UNITS}. The vertex shader, geometry shader, and fragment shader combined cannot use more than the value of \texttt{MAX\_COMBINED\_TEXTURE\_IMAGE\_UNITS} texture image units. If more than one of the vertex shader, geometry shader, and fragment processing stage access the same texture image unit, each such access counts separately against the \texttt{MAX\_COMBINED\_TEXTURE\_IMAGE\_UNITS} limit.

When a texture lookup is performed in a vertex or geometry shader, the filtered texture value $\tau$ is computed in the manner described in sections 3.8.9 and 3.8.10, and converted to a texture source color $C_s$ according to table 3.20 (section 3.9.2). A 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.

OpenGL 3.2 (Core Profile) - July 24, 2009
In a vertex or geometry shader, 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 3.8.9. 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 can either return the depth data directly or return the results of a comparison with a reference depth value specified in the coordinates passed to the texture lookup function, as described in section 3.8.15. The comparison operation is requested in the shader by using any of the shadow sampler types (sampler1DShadow, sampler2DShadow, or sampler2DRectShadow), 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 vertex or geometry shader uses a sampler where the associated texture object is not complete, as defined in section 3.8.12, the texture image unit will return \((R, G, B, A) = (0, 0, 0, 1)\).

**Shader Inputs**

Besides having access to vertex attributes and uniform variables, vertex shaders can access the read-only built-in variables \( \text{gl\_VertexID} \) and \( \text{gl\_InstanceID} \).
2.11. VERTEX SHADERS

`gl_VertexID` holds the integer index $i$ implicitly passed by `DrawArrays` or one of the other drawing commands defined in section 2.8.2.

`gl_InstanceID` holds the integer index of the current primitive in an instanced draw call (see section 2.8.2).

Section 7.1 of the OpenGL Shading Language Specification also describes these variables.

**Shader Outputs**

A vertex shader can write to user-defined varying 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 special variable `gl_Position` is intended to hold the homogeneous vertex position. Writing `gl_Position` is optional.

The built-in special variable `gl_ClipDistance` holds the clip distance(s) used in the clipping stage, as described in section 2.19. If clipping is enabled, `gl_ClipDistance` should be written.

The built-in special variable `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 actually will execute. Therefore validation is done when the first rendering command is issued, to determine if the currently active program object can be executed. If it cannot be executed then no fragments will be rendered, and the error `INVALID_OPERATION` will be generated.

This error is generated by any command that transfers vertices to the GL if:

- 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.

Undefined behavior results if the program object in use has no fragment shader unless transform feedback is enabled, in which case only a vertex shader is required.

The `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

```c
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 6.1.10). 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. Implementations should ensure that all valid shaders and program objects may be successfully compiled, linked and executed.

**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 reads will return undefined values; out-of-bounds writes will have undefined results and could corrupt other variables used by shader or the GL. The level of protection provided against such errors in the shader is implementation-dependent.

**2.11.8 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:
2.11. VERTEX SHADERS

- An unsigned integer specifying the shader object name.
- An integer holding the value of `SHADER_TYPE`.
- A boolean holding the delete status, initially `FALSE`.
- A boolean holding the status of the last compile, initially `FALSE`.
- 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.
- A list of unsigned integers to keep track of the names of the shader objects attached.
- 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.
Additional state required to support vertex shaders consists of:

- A bit indicating whether or not vertex program two-sided color mode is enabled, initially disabled.
- A bit indicating whether or not program point size mode (section 3.4.1) is enabled, initially disabled.

Additionally, one unsigned integer is required to hold the name of the current program object, if any.

2.12 Geometry Shaders

After vertices are processed, they are arranged into primitives, as described in section 2.6.1. This section describes optional geometry shaders, an additional pipeline stage defining operations to further process those primitives. Geometry shaders are 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 2.16).

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 2.11.2. 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 that includes a geometry shader must also include a vertex shader; otherwise a link error will occur.

2.12.1 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...
2.12. GEOMETRY SHADERS

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 currently active 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 2.12.4.

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)
2.12. GEOMETRY SHADERS

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.

### 2.12.2 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 2.12.4. 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.

### 2.12.3 Geometry Shader Variables

Geometry shaders can access uniforms belonging to the current program object. The amount of storage available for geometry shader uniform variables is specified by the implementation dependent constant `MAX_GEOMETRY_UNIFORM_...`
2.12. GEOMETRY SHADERS

COMPONENTS. This value represents the number of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a geometry shader. A link error will be 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 2.11.4. Geometry shaders also have access to samplers to perform texturing operations, as described in sections 2.11.5 and 3.8.

Geometry shaders can access the transformed attributes of all vertices for their input primitive type using input varying variables. A vertex shader writing to output varying variables generates the values of these input varying variables, including values for built-in as well as user-defined varying variables. Values for any varying variables that are not written by a vertex shader are undefined. Additionally, a geometry shader has access to a built-in variable 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 varying variables for each vertex they output. These values are optionally flatshaded (using the OpenGL Shading Language varying 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.

2.12.4 Geometry Shader Execution Environment

If a successfully linked program object that contains a geometry shader is made current by calling UseProgram, the executable version of the 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 2.13).
- Viewport mapping, including depth range scaling (section 2.13.1).
- Flatshading (section 2.18).
- Clipping, including client-defined half-spaces (section 2.19).
- Front face determination (section 3.6.1).
- Generic attribute clipping (section 2.19.1).

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

OpenGL 3.2 (Core Profile) - July 24, 2009
2.12. GEOMETRY SHADERS

Texture Access

The Shader Only Texturing subsection of section 2.11.7 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to geometry shaders.

Geometry Shader Inputs

Section 7.1 of the OpenGL Shading Language Specification describes the built-in variable array \texttt{gl_in[]} available as input to a geometry shader. \texttt{gl_in[]} receives values from equivalent built-in output variables written by the vertex shader, and each array element of \texttt{gl_in[]} is a structure holding values for a specific vertex of the input primitive. The length of \texttt{gl_in[]} is determined by the geometry shader input type (see section 2.12.1). The members of each element of the \texttt{gl_in[]} array are:

- Structure member \texttt{gl_ClipDistance[]} holds the per-vertex array of clip distances, as written by the vertex shader to its built-in output variable \texttt{gl_ClipDistance[]}.

- Structure member \texttt{gl_PointSize} holds the per-vertex point size written by the vertex shader to its built-in output varying variable \texttt{gl_PointSize}. If the vertex shader does not write \texttt{gl_PointSize}, the value of \texttt{gl_PointSize} is undefined, regardless of the value of the enable PROGRAM_POINT_SIZE.

- Structure member \texttt{gl_Position} holds the per-vertex position, as written by the vertex shader to its built-in output variable \texttt{gl_Position}. Note that writing to \texttt{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 special variable \texttt{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 varying variables, each user-defined input varying 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 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 2.11.6), there is a limit on the number of components of built-in and user-defined input varying variables that can be read by the geometry shader, given by the value of the implementation-dependent constant \texttt{MAX\_GEOMETRY\_INPUT\_COMPONENTS}.

When a program is linked, all components of any varying and special variable read by a geometry shader will count against this limit. A program whose geometry shader reads more than the value of \texttt{MAX\_GEOMETRY\_INPUT\_COMPONENTS} components worth of varying variables may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

**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 \texttt{GetProgramiv} with the symbolic constant \texttt{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 \texttt{GEOMETRY\_VERTICES\_OUT}; it may not exceed the value of \texttt{MAX\_GEOMETRY\_OUTPUT\_VERTICES}, and the product of the total number of vertices and the sum of all components of all active varying variables may not exceed the value of \texttt{MAX\_GEOMETRY\_TOTAL\_OUTPUT\_COMPONENTS}. \texttt{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 varying 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 output built-in varying variables and user-defined varying variables are the same as for the vertex shader. Refer to section 2.11.6, 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 built-in and user-defined output variables are undefined, as described in section 8.10 of the OpenGL Shading Language Specification.

The built-in special variable 
\texttt{gl\_Position} is intended to hold the homogeneous vertex position. Writing \texttt{gl\_Position} is optional.

The built-in special variable 
\texttt{gl\_PointSize}, if written, holds the size of the point to be rasterized, measured in pixels.

The built-in special variable 
\texttt{gl\_PrimitiveID} holds the primitive ID counter read by the fragment shader, replacing the value of \texttt{gl\_PrimitiveID} generated by drawing commands when no geometry shader is active. The geometry shader must write to \texttt{gl\_PrimitiveID} for the provoking vertex (see section 2.18) of a primitive being generated, or the primitive ID counter read by the fragment shader for that primitive is undefined.

The built-in special variable 
\texttt{gl\_Layer} is used in layered rendering, and discussed further in the next section.

Similarly to the limit on vertex shader output components (see section 2.11.6), there is a limit on the number of components of built-in and user-defined output varying variables that can be written by the geometry shader, given by the value of the implementation-dependent constant \texttt{MAX\_GEOMETRY\_OUTPUT\_COMPONENTS}.

When a program is linked, all components of any varying and special variable written by a geometry shader will count against this limit. A program whose geometry shader writes more than the value of \texttt{MAX\_GEOMETRY\_OUTPUT\_COMPONENTS} components worth of varying variables may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Layered Rendering

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 \texttt{gl\_Layer}. Layered rendering requires the use of framebuffer objects (see section 4.4.7).
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 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 POINTS and mode is not POINTS;
- the input primitive type of the current geometry shader is LINES and mode is not LINES, LINE_STRIP, or LINE_LOOP;
- the input primitive type of the current geometry shader is TRIANGLES and mode is not TRIANGLES, TRIANGLE_STRIP or TRIANGLE_FAN;
- the input primitive type of the current geometry shader is LINES_ADJACENCY and mode is not LINES_ADJACENCY or LINE_STRIP_ADJACENCY; or,
- the input primitive type of the current geometry shader is TRIANGLES_ADJACENCY and mode is not TRIANGLES_ADJACENCY or TRIANGLE_STRIP_ADJACENCY.

2.13 Coordinate Transformations

Clip coordinates for a vertex result from vertex or, if active, geometry shader execution, which yields a vertex coordinate 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 
\[
\begin{pmatrix}
  x_d \\
  y_d \\
  z_d
\end{pmatrix} = \begin{pmatrix}
  x_c \\
  y_c \\
  z_c \\
  w_c
\end{pmatrix}.
\]
2.13. COORDINATE TRANSFORMATIONS

2.13.1 Controlling the Viewport

The viewport transformation is determined by the 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

\[
\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{p_x}{2} x_d + o_x \\ \frac{p_y}{2} y_d + o_y \\ f - n z_d + \frac{n + f}{2} \end{pmatrix}.
\]

The factor and offset applied to \( z_d \) encoded by \( n \) and \( f \) are set using

```c
void DepthRange(clampd n, clampd f);
```

\( z_w \) is represented as either fixed- or floating-point depending on whether the framebuffer’s depth buffer uses a fixed- or floating-point representation. If the depth buffer uses fixed-point, 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). The parameters \( n \) and \( f \) are clamped to the range \([0,1]\), as are all arguments of type \( \text{clampd} \) or \( \text{clampf} \).

Viewport transformation parameters are specified using

```c
void Viewport(int x, int y, sizei w, sizei h);
```

where \( x \) and \( y \) give the \( x \) and \( y \) window coordinates of the viewport’s lower left corner and \( w \) and \( h \) give the viewport’s width and height, respectively. The viewport parameters shown in the above equations are found from these values as

\[
\begin{align*}
o_x &= x + \frac{w}{2} \\
o_y &= y + \frac{h}{2} \\
p_x &= w \\
p_y &= h.
\end{align*}
\]

Viewport width and height are clamped to implementation-dependent maximums when specified. The maximum width and height may be found by issuing an appropriate \textbf{Get} command (see chapter 6). 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. In the initial state, \( w \) and \( h \) 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.

### 2.14 Asynchronous Queries

Asynchronous queries provide a mechanism to return information about the processing of a sequence of GL commands. There are two query types supported by the GL. Transform feedback queries (see section 2.16) return information on the number of vertices and primitives processed by the GL and 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.

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 6.1.6 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.

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, 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 2.15), the
error INVALID_OPERATION is generated.

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 target is reset to zero. When
the commands issued prior to EndQuery have completed and a final query result
is available, the query object active when 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 target is zero when
EndQuery is called, the error INVALID_OPERATION is generated.

The command

```c
void GenQueries( sizei n, uint *ids );
```

returns n previously unused query object names in ids. These names are marked
as used, but no object is associated with them until the first time they are used by
BeginQuery.

Query objects are deleted by calling

```c
void DeleteQueries( sizei n, const uint *ids );
```

ids contains n names of query objects to be deleted. After a query object is deleted,
its name is again unused. Unused names in ids are silently ignored.

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 used to represent the query result is implementation-dependent. In
the initial state of a query object, the result is available and its value is zero.

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.

### 2.15 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
OpenGL 3.2 (Core Profile) - July 24, 2009
```
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, all rendering commands between BeginConditionalRender and the corresponding EndConditionalRender are discarded. In this case, all vertex array commands (see section 2.8), 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 of the occlusion query is non-zero, 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 unconditionally execute the subsequent rendering commands without waiting for the query to complete.

If BeginConditionalRender is called while conditional rendering is in progress, or if EndConditionalRender is called while conditional rendering is not in progress, the error INVALID_OPERATION is generated. The error INVALID_VALUE is generated if id is not the name of an existing query object. The error INVALID_OPERATION is generated if id is the name of a query object with a target other than SAMPLES_PASSED, or id is the name of a query currently in progress.

2.16 Transform Feedback

In transform feedback mode, attributes of the vertices of transformed primitives processed by a vertex shader, or primitives generated by a geometry shader if one
2.16. 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 2.11: Legal combinations of the transform feedback primitive mode, as passed to `BeginTransformFeedback`, and the current primitive mode.

is active, are written out to one or more buffer objects. The vertices are fed back after vertex color clamping, but before flat shading and clipping. If a geometry shader is active, the vertices recorded are those emitted from the geometry shader. 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.

Transform feedback 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`. `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 2.11.

Transform feedback commands must be paired; the error INVALID_OPERATION is generated by `BeginTransformFeedback` if transform feedback is active, and by `EndTransformFeedback` if transform feedback is inactive.

Transform feedback mode captures the values of varying variables written by the vertex shader (or, if active, geometry shader).

When transform feedback is active, 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 2.8.2 if `mode` is not one of the allowed...
2.16. **TRANSFORM FEEDBACK**

If a geometry shader is active, its output primitive type is used instead of the `mode` parameter passed to drawing commands.

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) with `target` set to `TRANSFORM_FEEDBACK_BUFFER`. In addition to the general errors described in section 2.9, `BindBufferRange` will generate an `INVALID_VALUE` error if `index` is greater than or equal to the value of `MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`, or if `offset` is not a multiple of 4.

When an individual point, line, or triangle primitive reaches the transform feedback stage while transform feedback is active, the values of the specified varying 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 varying variables that are arrays, individual array elements are written in order. For multi-component varying variables or varying array elements, 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.

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 varyings are written, interleaved, into the buffer object bound to the first transform feedback binding point (`index = 0`). If more than one varying variable is written, they will be recorded in the order specified by `TransformFeedbackVaryings` (see section 2.11.6). In `SEPARATE_ATTRIBS` mode, the first varying variable specified by `TransformFeedbackVaryings` is written to the first transform feedback binding point; subsequent varying 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`.

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 `offset + 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_QUERY_RESULT` is not incremented.
2.17. PRIMITIVE QUERIES

FEEDBACK_PRIMITIVES_WRITTEN (see section 2.17) is not incremented.

In either separate or interleaved modes, all transform feedback binding points that will be written to must have buffer objects bound when BeginTransformFeedback is called. The error INVALID_OPERATION is generated by BeginTransformFeedback if any binding point used in transform feedback mode does not have a buffer object bound. In interleaved mode, only the first buffer object binding point is ever written to. 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 varying variables to record.

While transform feedback is active, the set of attached buffer objects and the set of varying variables captured may not be changed. If transform feedback is active, the error INVALID_OPERATION is generated by UseProgram, by LinkProgram if program is the currently active program object, and by BindBufferRange or BindBufferBase if target is TRANSFORM_FEEDBACK_BUFFER.

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 Primitive Queries

Primitive queries use query objects to track the number of primitives generated by the GL and to track the number of primitives written to transform feedback buffers. When BeginQuery is called with a target of PRIMITIVES_GENERATED, the primitives-generated count maintained by the GL is set to zero. When the generated primitive query is active, the primitives-generated count is incremented every time a primitive reaches the “Discarding Primitives Before Rasterization” stage (see section 3.1) immediately before rasterization. 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 enabled.

When BeginQuery is called with a target of TRANSFORM_FEEDBACK_
PRIMITIVES_WRITTEN, the transform-feedback-primitives-written count maintained by the GL is set to zero. When the transform feedback primitive written query is active, the transform-feedback-primitives-written count is incremented every time a primitive is recorded into a buffer object. If transform feedback is not active, this counter is not incremented. If the primitive does not fit in the buffer object, the counter is not incremented.

These two queries can be used together to determine if all primitives 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, the buffer is full.

2.18 Flatshading

Flatshading a vertex shader varying output means to assign all vertices of the primitive the same value for that output.

The varying 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 varying outputs, as shown in table 2.12

If a vertex or geometry shader is active, user-defined varying outputs 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 primitives. The initial value of the provoking vertex mode is `LAST_VERTEX_CONVENTION`.

2.19 Primitive Clipping

Primitives are clipped to the *clip volume*. In clip coordinates, the *view volume* is defined by

\[-w_c \leq x_c \leq w_c\
\[-w_c \leq y_c \leq w_c\
\[-w_c \leq z_c \leq w_c.\]
2.19. 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 $i$</td>
<td>$i$</td>
<td>$i$</td>
</tr>
<tr>
<td>independent line $2i - 1$</td>
<td>$2i$</td>
<td>$2i$</td>
</tr>
<tr>
<td>line loop $i$</td>
<td>$i + 1$, if $i &lt; n$</td>
<td>$1$, if $i = n$</td>
</tr>
<tr>
<td>line strip $i$</td>
<td>$i + 1$</td>
<td>$i + 1$</td>
</tr>
<tr>
<td>independent triangle $3i - 2$</td>
<td>$3i$</td>
<td></td>
</tr>
<tr>
<td>triangle strip $i$</td>
<td>$i + 2$</td>
<td></td>
</tr>
<tr>
<td>triangle fan $i + 1$</td>
<td>$i + 2$</td>
<td></td>
</tr>
<tr>
<td>line adjacency $4i - 2$</td>
<td>$4i - 1$</td>
<td></td>
</tr>
<tr>
<td>line strip adjacency $i + 1$</td>
<td>$i + 2$</td>
<td></td>
</tr>
<tr>
<td>triangle adjacency $6i - 5$</td>
<td>$6i - 1$</td>
<td></td>
</tr>
<tr>
<td>triangle strip adjacency $2i - 1$</td>
<td>$2i + 3$</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.12: Provoking vertex selection. The varying 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.

This view volume may be further restricted by as many as $n$ client-defined half-spaces. ($n$ is an implementation-dependent maximum that must be at least 6.) 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

$$c_n(P) \geq 0,$$

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 Enable command and disabled with the Disable command. The value of the argument to either command is 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 CLIP_DISTANCE$i = CLIP_DISTANCE0 + i$. 

OpenGL 3.2 (Core Profile) - July 24, 2009
2.19. PRIMITIVE CLIPPING

Depth clamping is enabled with the generic Enable command and disabled with the 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 \( P \) and the original vertices’ coordinates are \( P_1 \) and \( P_2 \), then \( t \) is given by

\[ P = tP_1 + (1 - t)P_2. \]

The value of \( t \) is used to clip vertex shader varying variables as described in section 2.19.1.

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.
2.19.1 Clipping Shader Varying Outputs

Next, vertex shader varying variables are clipped. The varying values associated with a vertex that lies within the clip volume are unaffected by clipping. If a primitive is clipped, however, the varying values assigned to vertices produced by clipping are clipped.

Let the varying 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 2.19) for a clipped point \( P \) is used to obtain the varying value associated with \( P \) as

\[
\mathbf{c} = t \mathbf{c}_1 + (1 - t) \mathbf{c}_2.
\]

(Multiplying a varying 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. Varying 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 varying variables specified to be interpolated without perspective correction (using the \texttt{noperspective} qualifier), the value of \( t \) used to obtain the varying value associated with \( P \) will be adjusted to produce results that vary linearly in screen space.

Varying outputs of integer or unsigned integer type must always be declared with the \texttt{flat} qualifier. Since such varyings are constant over the primitive being rasterized (see sections 3.5.1 and 3.6.1), no interpolation is performed.

\footnote{Since this computation is performed in clip space before division by \( w_c \), clipped varying values are perspective-correct.}
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 3.9. 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 varying 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.
3.1 Discarding Primitives Before Rasterization

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 2.16). 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.

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 $f$. 

Figure 3.1. Rasterization.
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 5.3), 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 `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 `SAMPLE_BUFFERS` is queried by calling `GetIntegerv` with `pname` set to `SAMPLE_BUFFERS`.

During multisample rendering the contents of a pixel fragment are changed in two ways. First, each fragment includes a coverage value with `SAMPLES` bits. The value of `SAMPLES` is an implementation-dependent constant, and is queried by calling `GetIntegerv` with `pname` set to `SAMPLES`.

The location of a given sample is queried with the command

```c
void GetMultisamplefv(enum pname, uint index,
                      float *val);
```

`pname` must be `SAMPLE_POSITION`, and `index` corresponds to the sample for which the location should be returned. The sample location is returned as two
3.4. POINTS

Floating point values in \( \text{val}[0] \) and \( \text{val}[1] \), each between 0 and 1, corresponding to the \( x \) and \( y \) locations respectively in GL pixel space of that sample. \((0.5, 0.5)\) thus corresponds to the pixel center. The error `INVALID_VALUE` is generated if \( \text{index} \) is greater than or equal to the value of `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 `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 \( x \) and \( y \) grid coordinates, `SAMPLES depth values` and sets of associated data, and a coverage value with a maximum of `SAMPLES` bits.

Multisample rasterization is enabled or disabled by calling `Enable` or `Disable` with the symbolic constant `MULTISAMPLE`.

If `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 `MULTISAMPLE` is enabled, multisample rasterization of all primitives differs substantially from single-sample rasterization. It is understood that each pixel in the framebuffer has `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 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.

### 3.4 Points

A point is drawn by generating a set of fragments in the shape of a square or circle.
3.4. POINTS

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, then the derived point size is taken from the (potentially clipped) shader built-in `glPointSize` written by the geometry shader, or written by the vertex shader if no geometry shader is active, and clamped to the implementation-dependent point size range. If the value written to `glPointSize` is less than or equal to zero, results are undefined. If program point size mode is disabled, then 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 3.11) instead of allowing the point width to go below a given threshold. In this case, the width of the rasterized point is

\[
width = \begin{cases} 
  \text{derived.size} & \text{derived.size} \geq \text{threshold} \\
  \text{threshold} & \text{otherwise}
\end{cases}
\]

and the fade factor is computed as follows:

\[
fade = \begin{cases} 
  1 & \text{derived.size} \geq \text{threshold} \\
  \left(\frac{\text{derived.size}}{\text{threshold}}\right)^2 & \text{otherwise}
\end{cases}
\]

(3.1)

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`.

The point sprite texture coordinate origin is set with the `PointParameter*` commands where `pname` is `POINT_SPRITE_COORD_ORIGIN` and `param` is `LOWER_LEFT` or `UPPER_LEFT`. The default value is `UPPER_LEFT`. 
3.4. **POINTS**

### 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 builtin \texttt{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 \texttt{POINT\_SPRITE\_COORD\_ORIGIN} is \texttt{LOWER\_LEFT}, the \(t\) coordinate varies from 0 to 1 vertically bottom-to-top. Otherwise if the point sprite texture coordinate origin is \texttt{UPPER\_LEFT}, the \(t\) coordinate varies from 0 to 1 vertically top-to-bottom. The \(r\) and \(q\) coordinates are replaced with the constants 0 and 1, respectively.

The following formula is used to evaluate the \(s\) and \(t\) point sprite texture coordinates:

\[
s = \frac{1}{2} + \frac{(x_f + \frac{1}{2} - x_w)}{size} \tag{3.3}
\]

\[
t = \begin{cases} 
\frac{1}{2} + \frac{(y_f + \frac{1}{2} - y_w)}{size}, & \text{POINT\_SPRITE\_COORD\_ORIGIN} = \text{LOWER\_LEFT} \\
\frac{1}{2} - \frac{(y_f + \frac{1}{2} - y_w)}{size}, & \text{POINT\_SPRITE\_COORD\_ORIGIN} = \text{UPPER\_LEFT}
\end{cases} \tag{3.4}
\]

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, \ldots, 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 \texttt{MULTISAMPLE} is enabled, and the value of \texttt{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

\begin{verbatim}
void LineWidth( float width );
\end{verbatim}

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 \texttt{INVALID_VALUE}. Antialiasing is controlled with \texttt{Enable} and \texttt{Disable} using the symbolic constant \texttt{LINE_SMOOTH}.

3.5.1 Basic Line Segment Rasterization

Line segment rasterization begins by characterizing the segment as either \texttt{x-major} or \texttt{y-major}. \texttt{x-major} line segments have slope in the closed interval \([-1, 1]\); all other line segments are \texttt{y-major} (slope is determined by the segment’s endpoints). We shall specify rasterization only for \texttt{x-major} segments except in cases where the modifications for \texttt{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:

\[
R_f = \{ (x, y) \mid |x - x_f| + |y - y_f| < 1/2. \}
\]
3.5. **LINE SEGMENTS**

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

---

*Figure 3.2. Visualization of Bresenham’s algorithm. A portion of a line segment is shown. A diamond shaped region of height 1 is placed around each fragment center; those regions that the line segment exits cause rasterization to produce corresponding fragments.*
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 ap-
   pear 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
   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

\[
t = \frac{(p_r - p_a) \cdot (p_b - p_a)}{\|p_b - p_a\|^2}.
\]

(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 varying shader output or the clip \(w\) coordinate, is found as

\[
f = \frac{(1 - t)f_a/w_a + tf_b/w_b}{(1 - t)/w_a + t/w_b}
\]

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

\[
z = (1 - t)z_a + tz_b
\]

where \(z_a\) and \(z_b\) are the depth values of the starting and ending endpoints of the
segment, respectively.

The \textit{noperspective} and \textit{flat} keywords used to declare varying shader
outputs affect how they are interpolated. When neither keyword is specified, inter-
polation is performed as described in equation 3.6. When the \textit{noperspective}
keyword is specified, interpolation is performed in the same fashion as for depth values, as described in equation 3.7. When the flat keyword is specified, no interpolation is performed, and varying outputs are taken from the corresponding generic attribute value of the provoking vertex corresponding to that primitive (see section 2.18).

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.

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 3.3; 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
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.

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.

Line width range and number of gradations are equivalent to those supported for antialiased lines.

Figure 3.3. The region used in rasterizing and finding corresponding coverage values for an antialiased line segment (an x-major line segment is shown).
3.6. POLYGONS

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

$$a = \frac{1}{2} \sum_{i=0}^{n-1} x_{w_i} y_{w_i+1} - x_{w_i+1} y_{w_i}$$

(3.8)

where $x_{w_i}$ and $y_{w_i}$ 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 );
```

`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...
3.6. POLYGONS

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

\[
a = \frac{A(pp_bpc)}{A(papbpc)}, \quad b = \frac{A(ppapc)}{A(papbpc)}, \quad c = \frac{A(ppapb)}{A(papbpc)},
\]

where \( A(1mn) \) 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

\[
z = az_a + bz_b + cz_c \quad (3.10)
\]

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 varying shader outputs affect how they are interpolated. When neither keyword is specified, interpolation is performed as described in equation \ref{eq:3.9}. When the \texttt{noperspective} keyword is specified, interpolation is performed in the same fashion as for depth values, as described in equation \ref{eq:3.10}. When the \texttt{flat} keyword is specified, no interpolation is performed, and varying outputs are taken from the corresponding generic attribute value of the provoking vertex corresponding to that primitive (see section 2.18).
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 3.10. 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

```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

\[
\text{void PolygonOffset( float factor, float units );}
\]

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 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

$$r = 2^{e-n}.$$  

The offset value $o$ for a polygon is

$$o = m \times \text{factor} + r \times \text{units}. \quad (3.13)$$

$m$ 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 $[0, 1]$, either by clamping after offset addition is performed (preferred), or by clamping the vertex values used in the rasterization of the polygon. 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 BUFFERS} is one, then polygons are rasterized using the following algorithm, regardless of whether polygon antialiasing (\texttt{POLYGON SMOOTH}) 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.

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 \texttt{noperspective} and \texttt{flat} qualifiers affect how varying shader outputs are interpolated in the same fashion as described for basic polygon rasterization in section 3.6.1.

The rasterization described above applies only to the \texttt{FILL} state of \texttt{Polygon Mode}. For \texttt{POINT} and \texttt{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 \texttt{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 \texttt{PolygonMode} is \texttt{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 \texttt{TexImage*D} (see section 3.8.1). Some of the parameters and operations governing the operation of these commands are shared by \texttt{ReadPixels} (used to obtain pixel values from the framebuffer); the discussion of \texttt{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 \texttt{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 \texttt{PixelStore}.

#### 3.7.1 Pixel Storage Modes and Pixel Buffer Objects

Pixel storage modes affect the operation of \texttt{TexImage*D}, \texttt{TexSubImage*D}, and \texttt{ReadPixels} when one of these commands is issued. Pixel storage modes are set with

\begin{verbatim}
void PixelStore\{if\}(\texttt{enum pname, T param});
\end{verbatim}

\texttt{pname} is a symbolic constant indicating a parameter to be set, and \texttt{param} is the value to set it to. Table 3.1 summarizes the pixel storage parameters, their types,
### 3.7. PIXEL RECTANGLES

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Initial Value</th>
<th>Valid Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNPACK_SWAP_BYTES</td>
<td>boolean</td>
<td>FALSE</td>
<td>TRUE/FALSE</td>
</tr>
<tr>
<td>UNPACK_LSB_FIRST</td>
<td>boolean</td>
<td>FALSE</td>
<td>TRUE/FALSE</td>
</tr>
<tr>
<td>UNPACK_ROW_LENGTH</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>UNPACK_SKIP_ROWS</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>UNPACK_SKIP_PIXELS</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>UNPACK_ALIGNMENT</td>
<td>integer</td>
<td>4</td>
<td>1,2,4,8</td>
</tr>
<tr>
<td>UNPACK_IMAGE_HEIGHT</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>UNPACK_SKIP_IMAGES</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
</tbody>
</table>

Table 3.1: **PixelStore** parameters pertaining to one or more of **TexImage1D**, **TexImage2D**, **TexImage3D**, **TexSubImage1D**, **TexSubImage2D**, and **TexSubImage3D**.

their initial values, and their allowable ranges. Setting a parameter to a value outside the given range results in the error **INVALID_VALUE**.

The version of **PixelStore** that takes a floating-point value may be used to set any type of parameter; if the parameter is boolean, then it is set to **FALSE** if the passed value is 0.0 and **TRUE** otherwise, while if the parameter is an integer, then the passed value is rounded to the nearest integer. The integer version of the command may also be used to set any type of parameter; if the parameter is boolean, then it is set to **FALSE** if the passed value is 0 and **TRUE** otherwise, while if the parameter is a floating-point value, then the passed value is converted to floating-point.

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.

#### 3.7.2 Transfer of Pixel Rectangles

The process of transferring pixels encoded in buffer object or client memory is diagrammed in figure 3.4. We describe the stages of this process in the order in
3.7. PIXEL RECTANGLES

Figure 3.4. Transfer of pixel rectangles to the GL. Output is RGBA pixels. Depth and stencil pixel paths are not shown.

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 FLOAT, the error INVALID_ENUM occurs. Some additional constraints on the combinations of *format* and *type* values that are accepted are discussed below. Additional restrictions may be imposed by specific commands.
### 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 3.5.

<table>
<thead>
<tr>
<th>type Parameter Token Name</th>
<th>Corresponding GL Data Type</th>
<th>Special Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNSIGNED_BYTE</td>
<td>ubyte</td>
<td>No</td>
</tr>
<tr>
<td>BYTE</td>
<td>byte</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_SHORT</td>
<td>ushort</td>
<td>No</td>
</tr>
<tr>
<td>SHORT</td>
<td>short</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_INT</td>
<td>uint</td>
<td>No</td>
</tr>
<tr>
<td>INT</td>
<td>int</td>
<td>No</td>
</tr>
<tr>
<td>HALF_FLOAT</td>
<td>half</td>
<td>No</td>
</tr>
<tr>
<td>FLOAT</td>
<td>float</td>
<td>No</td>
</tr>
<tr>
<td>UNSIGNED_BYTE_3_3_2</td>
<td>ubyte</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_BYTE_2_3_3_REV</td>
<td>ubyte</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_6_5</td>
<td>ushort</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_6_5_REV</td>
<td>ushort</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_4_4_4_4</td>
<td>ushort</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_4_4_4_4_REV</td>
<td>ushort</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_5_5_1</td>
<td>ushort</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_1_5_5_5_REV</td>
<td>ushort</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_INT_8_8_8_8</td>
<td>uint</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_INT_8_8_8_8_REV</td>
<td>uint</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_INT_10_10_10_2</td>
<td>uint</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_INT_2_10_10_10_REV</td>
<td>uint</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_INT_24_8</td>
<td>uint</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_INT_10F_11F_11F_REV</td>
<td>uint</td>
<td>Yes</td>
</tr>
<tr>
<td>UNSIGNED_INT_5_9_9_9_REV</td>
<td>uint</td>
<td>Yes</td>
</tr>
<tr>
<td>FLOAT_32_UNSIGNED_INT_24_8_REV</td>
<td>n/a</td>
<td>Yes</td>
</tr>
</tbody>
</table>

OpenGL 3.2 (Core Profile) - July 24, 2009
### 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.

**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...
3.7. PIXEL RECTANGLES

<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.

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 `N`th row is indicated by

\[ p + Nk \]

where \( N \) is the row number (counting from zero) and \( k \) is defined as

\[
  k = \begin{cases} 
    n \frac{l}{s} & s \geq a, \\
    a/s \lfloor s \frac{n}{l} / a \rfloor & s < a 
  \end{cases} 
\]

where \( n \) is the number of elements in a group, \( l \) is the number of groups in
3.7. PIXEL RECTANGLES

the row, \( a \) is the value of UNPACK_ALIGNMENT, and \( s \) is the size, in units of GLubyte, of an element. If the number of bits per element is not 1, 2, 4, or 8 times the number of bits in a GLubyte, 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 3.5.

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_8_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.
3.7. PIXEL RECTANGLES

<table>
<thead>
<tr>
<th>Token Name</th>
<th>Type</th>
<th>Components</th>
<th>Pixel Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNSIGNED_BYTE_3_3_2</td>
<td>ubyte</td>
<td>3</td>
<td>RGB</td>
</tr>
<tr>
<td>UNSIGNED_BYTE_2_3_3_REV</td>
<td>ubyte</td>
<td>3</td>
<td>RGB</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_6_5</td>
<td>ushort</td>
<td>3</td>
<td>RGB</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_6_5_REV</td>
<td>ushort</td>
<td>3</td>
<td>RGB</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_4_4_4</td>
<td>ushort</td>
<td>4</td>
<td>RGBA,BGRA</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_4_4_4_REV</td>
<td>ushort</td>
<td>4</td>
<td>RGBA,BGRA</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_5_5_1</td>
<td>ushort</td>
<td>4</td>
<td>RGBA,BGRA</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_1_5_5_5_REV</td>
<td>ushort</td>
<td>4</td>
<td>RGBA,BGRA</td>
</tr>
<tr>
<td>UNSIGNED_INT_8_8_8_8</td>
<td>uint</td>
<td>4</td>
<td>RGBA,BGRA</td>
</tr>
<tr>
<td>UNSIGNED_INT_8_8_8_8_REV</td>
<td>uint</td>
<td>4</td>
<td>RGBA,BGRA</td>
</tr>
<tr>
<td>UNSIGNED_INT_10_10_10_2</td>
<td>uint</td>
<td>4</td>
<td>RGBA,BGRA</td>
</tr>
<tr>
<td>UNSIGNED_INT_2_10_10_10_REV</td>
<td>uint</td>
<td>4</td>
<td>RGBA,BGRA</td>
</tr>
<tr>
<td>UNSIGNED_INT_24_8</td>
<td>uint</td>
<td>2</td>
<td>DEPTH_STENCIL</td>
</tr>
<tr>
<td>UNSIGNED_INT_10F_11F_11F_REV</td>
<td>uint</td>
<td>3</td>
<td>RGB</td>
</tr>
<tr>
<td>UNSIGNED_INT_5_9_9_9_REV</td>
<td>uint</td>
<td>4</td>
<td>RGB</td>
</tr>
<tr>
<td>FLOAT_32_UNSIGNED_INT_24_8_REV</td>
<td>n/a</td>
<td>2</td>
<td>DEPTH_STENCIL</td>
</tr>
</tbody>
</table>

Table 3.5: Packed pixel formats.

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. If the base GL type is supported with more than the minimum precision (e.g. a 9-bit byte) the packed components are right-justified in the pixel.

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.
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 3.7: UNSIGNED_SHORT formats**

<table>
<thead>
<tr>
<th>Format</th>
<th>Bit Positions</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNSIGNED_SHORT_5_6_5:</td>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_6_5_REV:</td>
<td>3rd 2nd 1st Component</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_4_4_4_4:</td>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_4_4_4_4_REV:</td>
<td>4th 3rd 2nd 1st Component</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_5_5_1:</td>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_1_5_5_5_REV:</td>
<td>4th 3rd 2nd 1st Component</td>
</tr>
</tbody>
</table>

OpenGL 3.2 (Core Profile) - July 24, 2009
### UNSIGNED_INT_8_8_8_8:

<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>

### UNSIGNED_INT_8_8_8_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>4th</td>
</tr>
</tbody>
</table>

### UNSIGNED_INT_10_10_10_2:

<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>

### UNSIGNED_INT_2_10_10_10_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>4th</td>
</tr>
</tbody>
</table>

### UNSIGNED_INT_24_8:

<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>

### UNSIGNED_INT_10F_11F_11F_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>3rd</td>
</tr>
</tbody>
</table>

### UNSIGNED_INT_5_9_9_9_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>4th</td>
</tr>
</tbody>
</table>

#### 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</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 2.1.3. $f_{\text{blue}}$ is treated as an unsigned 10-bit floating-point value and converted to a floating-point blue component as described in section 2.1.4.

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 $p_{\exp}$ 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:

\[
\text{red} = p_{\text{red}}2^{p_{\exp} - B - N}
\]
\[
\text{green} = p_{\text{green}}2^{p_{\exp} - B - N}
\]
\[
\text{blue} = p_{\text{blue}}2^{p_{\exp} - B - N}
\]

where $B = 15$ (the exponent bias) and $N = 9$ (the number of mantissa bits).
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 unless the final destination of the transferred element is a texture or framebuffer component in one of the SNORM formats described in table 3.12, in which case equation 2.3 is used instead.

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 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 spe-
cial 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. 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 3.8.9) 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 3.9.

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 3.8.

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.

### 3.8.1 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, void *data );
```

OpenGL 3.2 (Core Profile) - July 24, 2009
is used to specify a three-dimensional texture image. target must be one of TEXTURE_3D for a three-dimensional texture or TEXTURE_2D_ARRAY for an two-dimensional array texture. Additionally, target may be either PROXY_TEXTURE_3D for a three-dimensional proxy texture, or PROXY_TEXTURE_2D_ARRAY for a two-dimensional proxy array texture, as discussed in section 3.8.13. 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 internal format 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 internal format 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
3.8. 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 3.20.

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 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, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_1D_ARRAY, PROXY_TEXTURE_2D_ARRAY, PROXY_TEXTURE_RECTANGLE, or PROXY_TEXTURE_CUBE_MAP. 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. The error INVALID_OPERATION is generated if the internal format is integer and format is not one of the integer formats listed in table 3.3; 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.

In addition to the specific compressed internal formats listed in table 3.14, the GL provides a mechanism to obtain token values for all such formats provided
by extensions. The number of specific compressed internal formats supported by the renderer can be obtained by querying the value of `NUM_COMPRESSED_TEXTURE_FORMATS`. The set of specific compressed internal formats supported by the renderer can be obtained by querying the value of `COMPRESSED_TEXTURE_FORMATS`. The only values returned by this query are those corresponding to formats suitable for general-purpose usage. The renderer will not enumerate formats with restrictions that need to be specifically understood prior to use.

Generic compressed internal formats are never used directly as the internal formats of texture images. If `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, `internalformat` is instead replaced by the corresponding base internal format. If `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), `internalformat` is replaced by the corresponding base internal format and the texture image will not be compressed by the GL.

The `internal component resolution` is the number of bits allocated to each value in a texture image. If `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 `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 desired allocation in tables 3.12-3.13 is non-zero, and zero bits must be allocated for all other components).

**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:
3.8. TEXTURING

- Texture and renderbuffer color formats (see section 4.4.2)).
  - RGBA32F, RGBA32I, RGBA32UI, RGBA16, RGBA16F, RGBA16I, RGBA16UI, RGBA8, RGBA8I, RGBA8UI, SRGB8_ALPHA8, and RGB10_A2.
  - R11F_G11F_B10F.
  - 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 2.1.3 and 2.1.4.

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:
3.8. 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:

\[
max_c = \max(red_c, green_c, blue_c)
\]

A preliminary shared exponent \(exp_p\) is computed:

\[
exp_p = \max(-B - 1, \lfloor \log_2(max_c) \rfloor) + 1 + B
\]

A refined shared exponent \(exp_s\) is computed:

\[
max_s = \left\lfloor \frac{max_c}{2^{exp_p - B - N} + 0.5} \right\rfloor
\]

\[
exp_s = \begin{cases} 
exp_p, & 0 \leq max_s < 2^N \\
exp_p + 1, & max_s = 2^N 
\end{cases}
\]

Finally, three integer values in the range 0 to \(2^N - 1\) are computed:

\[
red_s = \left\lfloor \frac{red_c}{2^{exp_s - B - N} + 0.5} \right\rfloor \\
green_s = \left\lfloor \frac{green_c}{2^{exp_s - B - N} + 0.5} \right\rfloor \\
blue_s = \left\lfloor \frac{blue_c}{2^{exp_s - B - N} + 0.5} \right\rfloor
\]

The resulting \(red_s\), \(green_s\), \(blue_s\), and \(exp_s\) 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 \texttt{format RGB} is allowed to store the components “as is”.

OpenGL 3.2 (Core Profile) - July 24, 2009
<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>s16</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>RGBA</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB5</td>
<td>RGBA</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB8</td>
<td>RGBA</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB8_SNORM</td>
<td>RGBA</td>
<td>s8</td>
<td>s8</td>
<td>s8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB10</td>
<td>RGBA</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB12</td>
<td>RGBA</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB16</td>
<td>RGBA</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB16_SNORM</td>
<td>RGBA</td>
<td>s16</td>
<td>s16</td>
<td>s16</td>
<td>s16</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>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>RGBA</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRGB8_ALPHA8</td>
<td>RGBA</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</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>
<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>
</tbody>
</table>

Sized internal color formats continued on next page
### 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>
<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>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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG8UI</td>
<td>RG</td>
<td>ui8</td>
<td>ui8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG16I</td>
<td>RG</td>
<td>i16</td>
<td>i16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG16UI</td>
<td>RG</td>
<td>ui16</td>
<td>ui16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG32I</td>
<td>RG</td>
<td>i32</td>
<td>i32</td>
<td>i32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG32UI</td>
<td>RG</td>
<td>ui32</td>
<td>ui32</td>
<td>ui32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB8I</td>
<td>RGB</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>i32</td>
<td></td>
</tr>
<tr>
<td>RGB32UI</td>
<td>RGB</td>
<td>ui32</td>
<td>ui32</td>
<td>ui32</td>
<td>ui32</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 3.13: 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.

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 \texttt{TexImage3D}, \texttt{TexImage2D} (see below), or \texttt{TexImage1D} (see below) parameter (except \texttt{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 \texttt{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 3.8.13.

The image itself (referred to by \texttt{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 \texttt{width} from left to right; \texttt{height} rows are stacked from bottom to top forming a single two-dimensional image slice; and \texttt{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

\[
i = (N \mod \texttt{width}) - w_0
\]
### 3.8. 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>
</tbody>
</table>

Table 3.14: Generic and specific compressed internal formats. The specific *RGTC* formats are described in appendix C.1.

\[
j = \left( \left\lfloor \frac{N}{\text{width}} \right\rfloor \mod \text{height} \right) - h_b
\]

\[
k = \left( \left\lfloor \frac{N}{\text{width} \times \text{height}} \right\rfloor \mod \text{depth} \right) - d_b
\]

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 \(\text{target}\) is \(\text{TEXTURE}_3\text{D}\), or zero if \(\text{target}\) is \(\text{TEXTURE}_2\text{D}_\text{ARRAY}\). Thus the last two-dimensional image slice of the three-dimensional image is indexed with the highest value of \(k\).

If the internal data type of the image array is signed or unsigned normalized fixed-point, each color component is converted using equation 2.6 or 2.4, 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 \(\text{level}\) argument to \(\text{TexImage3D}\) is an integer \(\text{level-of-detail}\) number. Levels of detail are discussed below, under \textit{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 \texttt{INVALID_VALUE} is generated.

The \(\text{border}\) argument to \(\text{TexImage3D}\) is a border width. The significance of borders is described below. The border width affects the dimensions of the texture.
image: let

\[ w_s = w_t + 2w_b \]
\[ h_s = h_t + 2h_b \]
\[ d_s = d_t + 2d_b \]  

(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`, \( \text{lod} \) 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 \( k \). The error `INVALID_VALUE` is generated if the specified image is too large to be stored under any conditions.

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 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 `MAX_CUBE_MAP_TEXTURE_SIZE`. The maximum number of layers for one- and two-dimensional array textures (height or depth, respectively) must be at least `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 3.8.12.
The command

```c
void TexImage2D( enum target, int level,
                 int internalformat, sizei width, sizei height,
                 int border, enum format, enum type, 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 3.8.13. 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, sizei height,
                 int border, enum format, enum type, void *data );
```
3.8. TEXTURING

```c
int internalformat, sizei width, int border,
enum format, enum type, 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 3.8.13.

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.

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 3.6.

If the \textit{data} argument of TexImage1D, TexImage2D, or TexImage3D is 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 \textit{target}, \textit{level}, \textit{internalformat}, \textit{border}, \textit{width}, \textit{height}, and \textit{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 \textit{data} pointer were valid. Otherwise if the pixel unpack buffer object is non-zero, the \textit{data} argument is treatedly normally to refer to the beginning of the pixel unpack buffer object’s data.

### 3.8.2 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 );
```

OpenGL 3.2 (Core Profile) - July 24, 2009
Figure 3.6. A texture image and the coordinates used to access it. This is a two-dimensional texture with $n = 3$ and $m = 2$. 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.
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, 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 ReadPixels 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

```c
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,

```c
void TexSubImage3D( enum target, int level, int xoffset, int yoffset, int zoffset, sizei width, sizei height, sizei depth, enum format, enum type, void *data );
void TexSubImage2D( enum target, int level, int xoffset, int yoffset, int x, int y, sizei width, sizei height, enum format, enum type, void *data );
void TexSubImage1D( enum target, int level, int xoffset, int yoffset, int zoffset, int x, int y, sizei width, enum format, enum type, 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 );
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. Currently 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 or TEXTURE_2D_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\(^1\). 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 CopyTexSubImage3D specify the lower left texel coordinates of a width-wide by height-high by depth-deep rectangular subregion of the texel array. 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 3.6. 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:

\[
x < -w_b \\
x + w > w_s - w_b \\
y < -h_b
\]

\(^1\) Because the framebuffer is inherently two-dimensional, there is no CopyTexImage3D command.
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 + (\lfloor \frac{n}{w} \rfloor \mod h) \\
k &= z + (\lfloor \frac{n}{\text{width} \times \text{height}} \rfloor \mod d)
\end{align*}

Arguments $\text{xoffset}$ and $\text{yoffset}$ of \textbf{TexSubImage2D} and \textbf{CopyTexSubImage2D} specify the lower left texel coordinates of a $\text{width}$-wide by $\text{height}$-high rectangular subregion of the texel array. Negative values of $\text{xoffset}$ and $\text{yoffset}$ correspond to the coordinates of border texels, addressed as in figure 3.6. 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 $\text{xoffset}$, $\text{yoffset}$, width, and height argument values, any of the following relationships generates the error $\text{INVALID\_VALUE}$:

\begin{align*}
x &< -b_s \\
x + w &> w_s - b_s \\
y &< -b_s \\
y + h &> h_s - b_s
\end{align*}

Counting from zero, the $n$th pixel group is assigned to the texel with internal integer coordinates $[i, j]$, where

\begin{align*}
i &= x + (n \mod w) \\
j &= y + (\lfloor \frac{n}{w} \rfloor \mod h)
\end{align*}

The $\text{xoffset}$ argument of \textbf{TexSubImage1D} and \textbf{CopyTexSubImage1D} specifies the left texel coordinate of a $\text{width}$-wide subregion of the texel array. Negative values of $\text{xoffset}$ 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 $\text{xoffset}$ and $\text{width}$ argument values, either of the following relationships generates the error $\text{INVALID\_VALUE}$:

\begin{align*}
x &< -b_s \\
x + w &> w_s - b_s
\end{align*}
3.8. TEXTURING

\[ x < -b_s \]
\[ x + w > w_s - b_s \]

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 \( \text{TexSubImage3D} \), \( \text{CopyTexSubImage3D} \), \( \text{TexSubImage2D} \), \( \text{CopyTexSubImage2D} \), \( \text{TexSubImage1D} \), or \( \text{CopyTexSubImage1D} \) will result in an \text{INVALID_OPERATION} error if \( x_{\text{offset}} \), \( y_{\text{offset}} \), or \( z_{\text{offset}} \) 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 \text{RGTC} formats described in table 3.14, the texture is stored using one of the \text{RGTC} texture image encodings (see appendix C.1). Since \text{RGTC} images are easily edited along \( 4 \times 4 \) texel boundaries, the limitations on subimage location and size are relaxed for \( \text{TexSubImage2D} \), \( \text{TexSubImage3D} \), \( \text{CopyTexSubImage2D} \), and \( \text{CopyTexSubImage3D} \). These commands will generate an \text{INVALID_OPERATION} error if one of the following conditions occurs:

- \text{width} is not a multiple of four, \( \text{width} + x_{\text{offset}} \) is not equal to the value of \( \text{TEXTURE_WIDTH} \), and either \( x_{\text{offset}} \) or \( y_{\text{offset}} \) is non-zero.

- \text{height} is not a multiple of four, \( \text{height} + y_{\text{offset}} \) is not equal to the value of \( \text{TEXTURE_HEIGHT} \), and either \( x_{\text{offset}} \) or \( y_{\text{offset}} \) is non-zero.

- \( x_{\text{offset}} \) or \( y_{\text{offset}} \) is not a multiple of four.

The contents of any \( 4 \times 4 \) block of texels of an \text{RGTC} compressed texture image that does not intersect the area being modified are preserved during valid \( \text{TexSubImage}^{\ast} \) and \( \text{CopyTexSubImage}^{\ast} \) calls.
3.8. TEXTURING

Calling `CopyTexImage1D`, `CopyTexImage2D`, `CopySubTexImage2D`, `CopySubTexImage1D`, or `CopySubTexImage1D` 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 `CopyTexImage1D`, `CopyTexImage2D`, `CopySubTexImage2D`, `CopySubTexImage1D`, or `CopySubTexImage1D` 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.

3.8.3 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, void *data);
void CompressedTexImage2D(enum target, int level,
    enum internalformat, sizei width, sizei height,
    int border, sizei imageSize, void *data);
void CompressedTexImage3D(enum target, int level,
    enum internalformat, sizei width, sizei height,
    sizei depth, int border, sizei imageSize, 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.
3.8. TEXTURING

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. All pixel storage modes are ignored when decoding a compressed texture image. 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.

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 `border` values. Any such restrictions will be documented in the extension 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 invariant, meaning that if the GL accepts and stores a texture image in compressed form, providing the same image to `CompressedTexImage1D`, `CompressedTexImage2D`, or `CompressedTexImage3D` will not result in an `INVALID_OPERATION` error if the following restrictions are satisfied:

- `data` points to a compressed texture image returned by `GetCompressedTexImage` (section 6.1.4).
- `target`, `level`, and `internalformat` match the `target`, `level` and `format` parameters provided to the `GetCompressedTexImage` call returning `data`.
- `width`, `height`, `depth`, `border`, `internalformat`, and `imageSize` match the values of `TEXTURE_WIDTH`, `TEXTURE_HEIGHT`, `TEXTURE_DEPTH`, `TEXTURE_BORDER`, `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 formats described in table 3.14, the compressed image data is stored using one of the RGTC compressed texture image encodings (see appendix C.1). The RGTC texture compression algorithm supports only two-dimensional images without borders. 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 the `data` argument of `CompressedTexImage1D`, `CompressedTexImage2D`, or `CompressedTexImage3D` is 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 CompressedTexSubImage1D( enum target, int level,
                               int xOffset, sizei width, enum format, sizei imageSize,
                               void *data );
void CompressedTexSubImage2D( enum target, int level,
                               int xOffset, int yOffset, sizei width, sizei height,
                               enum format, sizei imageSize, void *data );
void CompressedTexSubImage3D( enum target, int level,
                               int xOffset, int yOffset, int zOffset, sizei width,
                               sizei height, sizei depth, enum format,
                               sizei imageSize, void *data );
```

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`, `zOffset`, `width`, `height`, and `depth` parameters have the same meaning as in `TexImage1D`, `TexImage2D`, and `TexImage3D`. `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*nD` commands is `TEXTURE_RECTANGLE` or `PROXY_TEXTURE_RECTANGLE`, the error `INVALID_ENUM` is generated.
The image pointed to by \textit{data} and the \textit{imageSize} parameter are interpreted as though they were provided to \texttt{CompressedTexImage1D}, \texttt{CompressedTexImage2D}, and \texttt{CompressedTexImage3D}. These commands do not provide for image format conversion, so an \texttt{INVALID_OPERATION} error results if \textit{format} does not match the internal format of the texture image being modified. If the \textit{imageSize} parameter is not consistent with the format, dimensions, and contents of the compressed image (too little or too much data), an \texttt{INVALID_VALUE} error results.

As with \texttt{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 \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{CompressedTexSubImage1D}, \texttt{CompressedTexSubImage2D}, \texttt{CompressedTexSubImage3D} will not result in an \texttt{INVALID_OPERATION} error if the following restrictions are satisfied:

\begin{itemize}
  \item \textit{data} points to a compressed texture image returned by \texttt{GetCompressedTexImage} (section 6.1.4).
  \item \textit{target}, \textit{level}, and \textit{format} match the \textit{target}, \textit{level} and \textit{format} parameters provided to the \texttt{GetCompressedTexImage} call returning \textit{data}.
  \item \textit{width}, \textit{height}, \textit{depth}, \textit{format}, and \textit{imageSize} match the values of \texttt{TEXTURE_WIDTH}, \texttt{TEXTURE_HEIGHT}, \texttt{TEXTURE_DEPTH}, \texttt{TEXTURE_INTERNAL_FORMAT}, and \texttt{TEXTURE_COMPRESSED_IMAGE_SIZE} for image level \textit{level} in effect at the time of the \texttt{GetCompressedTexImage} call returning \textit{data}.
  \item \textit{width}, \textit{height}, \textit{depth}, \textit{format} match the values of \texttt{TEXTURE_WIDTH}, \texttt{TEXTURE_HEIGHT}, \texttt{TEXTURE_DEPTH}, and \texttt{TEXTURE_INTERNAL_FORMAT} currently in effect for image level \textit{level}.
  \item \textit{xoffset}, \textit{yoffset}, and \textit{zoffset} are all $-b$, where \texttt{b} is the value of \texttt{TEXTURE_BORDER} currently in effect for image level \textit{level}.
\end{itemize}

This guarantee applies not just to images returned by \texttt{GetCompressedTexImage}, but also to any other properly encoded compressed texture image of the same size.

Calling \texttt{CompressedTexSubImage3D}, \texttt{CompressedTexSubImage2D}, or \texttt{CompressedTexSubImage1D} will result in an \texttt{INVALID_OPERATION} error if \textit{xoffset}, \textit{yoffset}, or \textit{zoffset} is not equal to $-b$ (border width), or if \textit{width}, \textit{height},
3.8. TEXTURING

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 internalformat is one of the specific RGTC formats described in table 3.14, the texture is stored using one of the RGTC compressed texture image encodings (see appendix C.1). 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; and CompressedTexImage3D will generate an INVALID_OPERATION error if border is non-zero or target is not TEXTURE_2D_ARRAY. Since RGTC images are easily edited along 4 × 4 texel boundaries, the limitations on subimage location and size are relaxed for CompressedTexImage2D and CompressedTexImage3D. 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.

3.8.4 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 );
```

OpenGL 3.2 (Core Profile) - July 24, 2009
establish the data storage, format, dimensions, and number of samples of a multisample texture’s image. For TexImage2DMultisample, \texttt{target} must be \texttt{TEXTURE\_2D\_MULTISAMPLE} or \texttt{PROXY\_TEXTURE\_2D\_MULTISAMPLE} and for TexImage3DMultisample \texttt{target} must be \texttt{TEXTURE\_2D\_MULTISAMPLE\_ARRAY} or \texttt{PROXY\_TEXTURE\_2D\_MULTISAMPLE\_ARRAY}. \texttt{width} and \texttt{height} are the dimensions in texels of the texture.

\texttt{internalformat} must be color-renderable, depth-renderable, or stencil-renderable (as defined in section 4.4.4). The error \texttt{INVALID\_OPERATION} may be generated if any of the following are true:

- \texttt{internalformat} is a depth/stencil-renderable format and \texttt{samples} is greater than the value of \texttt{MAX\_DEPTH\_TEXTURE\_SAMPLES}
- \texttt{internalformat} is a color-renderable format and \texttt{samples} is greater than the value of \texttt{MAX\_COLOR\_TEXTURE\_SAMPLES}
- \texttt{internalformat} is a signed or unsigned integer format and \texttt{samples} is greater than the value of \texttt{MAX\_INTEGER\_SAMPLES}.

If \texttt{fixedsamplelocations} is \texttt{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. If either \texttt{width} or \texttt{height} is greater than \texttt{MAX\_TEXTURE\_SIZE}, or if \texttt{samples} is greater than \texttt{MAX\_SAMPLES}, then the error \texttt{INVALID\_VALUE} is generated. If the GL is unable to create a texture level of the requested size, the error \texttt{OUT\_OF\_MEMORY} 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.

3.8.5 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

```c
void TexBuffer( enum target, enum internalformat, uint buffer );
```

attaches the storage for the buffer object named `buffer` to the active buffer texture, and specifies an internal format for the texel array found in the attached buffer object. If `buffer` is zero, any buffer object attached to the buffer texture is detached, and no new buffer object is attached. If `buffer` is non-zero, but is not the name of an existing buffer object, the error `INVALID_OPERATION` is generated. `target` must be `TEXTURE_BUFFER`. `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

\[
\text{buffer}_\text{size} \times \text{components} \times \text{sizeof} \text{(base type)}
\]

where `buffer_size` is the size of the buffer object, in basic machine units and `components` and `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 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 2.1 or 2.3, respectively. The components of the texture are then converted to an (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
3.8. TEXTURING

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.

3.8.6 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,
T *params);
void TexParameterI{i ui}v( enum target, enum pname,
T *params);
```

target is the target, either TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_RECTANGLE, or TEXTURE_CUBE_MAP. 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.

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.

In the remainder of section 3.8, denote by lodmin, lodmax, levelbase, and levelmax the values of the texture parameters TEXTURE_MIN_LOD, TEXTURE_MAX_LOD, TEXTURE_BASE_LEVEL, and TEXTURE_MAX_LEVEL respectively.

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, TEXTURE_WRAP_T, or TEXTURE_WRAP_R parameter is set.
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.
### 3.8. TEXTURING

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Legal Values</th>
</tr>
</thead>
<tbody>
<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>
<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_MAG_FILTER</td>
<td>enum</td>
<td>NEAREST, LINEAR</td>
</tr>
<tr>
<td>TEXTURE_BORDER_COLOR</td>
<td>4 floats, integers, or unsigned integers</td>
<td>any 4 values</td>
</tr>
<tr>
<td>TEXTURE_MIN_LOD</td>
<td>float</td>
<td>any value</td>
</tr>
<tr>
<td>TEXTURE_MAX_LOD</td>
<td>float</td>
<td>any value</td>
</tr>
<tr>
<td>TEXTURE_BASE_LEVEL</td>
<td>integer</td>
<td>any non-negative integer</td>
</tr>
<tr>
<td>TEXTURE_MAX_LEVEL</td>
<td>integer</td>
<td>any non-negative integer</td>
</tr>
<tr>
<td>TEXTURE_LOD_BIAS</td>
<td>integer</td>
<td>any non-negative integer</td>
</tr>
<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>
</tbody>
</table>

Table 3.16: Texture parameters and their values.
3.8. TEXTURING

### Table 3.17: Selection of cube map images based on major axis direction of texture coordinates.

<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>$r_z$</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>$-r_z$</td>
<td>$r_y$</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>

3.8.7 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 3.8.15). The initial state for depth and depth/stencil textures treats them as `RED` textures.

3.8.8 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 can be ignored, since it merely scales the vector without affecting the direction.) 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:

$$s = \frac{1}{2} \left( \frac{s_c}{|m_a|} + 1 \right)$$
3.8. TEXTURING

\[ t = \frac{1}{2} \left( \frac{t_c}{|m_a|} + 1 \right) \]

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 3.8.9 through 3.8.10.

When seamless cube map filtering is enabled, the rules for texel selection in sections 3.8.9 through 3.8.10 are modified so that texture wrap modes are ignored. Instead,

- If `NEAREST` filtering is done within a mipmap, always apply wrap mode `CLAMP_TO_EDGE`.
- If `LINEAR` filtering is done within a mipmap, 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.

3.8.9 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

$$\lambda_{\text{base}}(x, y) = \log_2[\rho(x, y)]$$  \hspace{1cm} (3.17)

$$\lambda'(x, y) = \lambda_{\text{base}}(x, y) + \text{clamp}(bias_{\text{texobj}} + bias_{\text{shader}})$$  \hspace{1cm} (3.18)

$$\lambda = \begin{cases} 
\text{lod}_{\text{max}}, & \lambda' > \text{lod}_{\text{max}} \\
\lambda', & \text{lod}_{\text{min}} \leq \lambda' \leq \text{lod}_{\text{max}} \\
\text{lod}_{\text{min}}, & \lambda' < \text{lod}_{\text{min}} \\
\text{undefined}, & \text{lod}_{\text{min}} > \text{lod}_{\text{max}} 
\end{cases}$$  \hspace{1cm} (3.19)

$bias_{\text{texobj}}$ is the value of TEXTURE_LOD_BIAS for the bound texture object (as described in section 3.8.6). $bias_{\text{shader}}$ 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 $bias_{\text{shader}}$ is zero. The sum of these values is clamped to the range $[-bias_{\text{max}}, bias_{\text{max}}]$ 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 3.8.10) 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 3.8.10.

The initial values of $\text{lod}_{\text{min}}$ and $\text{lod}_{\text{max}}$ are chosen so as to never clamp the normal range of $\lambda$. They may be respecified for a specific texture by calling **TexParameter**[if] 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

\[
\begin{align*}
u(x, y) &= \begin{cases} 
w_t + \delta_u, & \text{rectangular texture} \\w_t \times s(x, y) + \delta_u, & \text{otherwise}
\end{cases}, \\
v(x, y) &= \begin{cases} 
h_t + \delta_u, & \text{rectangular texture} \\h_t \times t(x, y) + \delta_v, & \text{otherwise}
\end{cases}, \\
w(x, y) &= d_t \times r(x, y) + \delta_w
\end{align*}
\]

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 \(\text{level}_{\text{base}}.\) 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, or cube map 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 any of the offset values are outside the range of the implementation-defined values \(\text{MIN\_PROGRAM\_TEXEL\_OFFSET}\) and \(\text{MAX\_PROGRAM\_TEXEL\_OFFSET},\) results of the texture lookup are undefined.

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 imple-
Texturing 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

$$m_u = \max \left\{ \frac{|\partial u|}{\partial x}, \frac{|\partial u|}{\partial y} \right\}$$

$$m_v = \max \left\{ \frac{|\partial v|}{\partial x}, \frac{|\partial v|}{\partial y} \right\}$$

$$m_w = \max \left\{ \frac{|\partial w|}{\partial x}, \frac{|\partial w|}{\partial y} \right\}.$$

Then $\max\{m_u, m_v, m_w\} \leq f(x, y) \leq m_u + m_v + m_w$.

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

$$u'(x, y) = \begin{cases} u(x, y), & \text{otherwise} \\ \end{cases}$$

$$v'(x, y) = \begin{cases} v(x, y), & \text{otherwise} \\ \end{cases}$$

$$w'(x, y) = \begin{cases} w(x, y), & \text{otherwise} \\ \end{cases}$$

where $\text{clamp}(a, b, c)$ returns $b$ if $a < b$, $c$ if $a > c$, and $a$ otherwise.

The value assigned to $\text{TEXTURE\_MIN\_FILTER}$ is used to determine how the texture value for a fragment is selected.

When the value of $\text{TEXTURE\_MIN\_FILTER}$ is $\text{NEAREST}$, the texel in the image array of level $\text{level}_\text{base}$ that is nearest (in Manhattan distance) to $(u', v', w')$ is obtained. Let $(i, j, k)$ be integers such that

Open GL 3.2 (Core Profile) - July 24, 2009
3.8. TEXTURING

\[ i = \text{wrap}(\lfloor u'(x,y) \rfloor) \]
\[ j = \text{wrap}(\lfloor v'(x,y) \rfloor) \]
\[ k = \text{wrap}(\lfloor w'(x,y) \rfloor) \]

and the value returned by \( \text{wrap}() \) 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

\[ l = \begin{cases} 
\text{clamp}(\lfloor t + 0.5 \rfloor, 0, h_t - 1), & \text{for one-dimensional array textures} \\
\text{clamp}(\lfloor r + 0.5 \rfloor, 0, d_t - 1), & \text{for two-dimensional array textures} 
\end{cases} \]

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 \text{w_t}, \text{TEXTURE_WRAP_T} and \text{h_t}, and \text{TEXTURE_WRAP_R} and \text{d_t} when wrapping \(i\), \(j\), or \(k\) coordinates, respectively.

If the selected \((i, j, k)\), \((i, j)\), or \(i\) location refers to a border texel that satisfies any of the conditions

\[ i < -b_s \quad i \geq w_t + b_s \]
\[ j < -b_s \quad j \geq h_t + b_s \]
\[ k < -b_s \quad k \geq d_t + b_s \]

then the border values defined by \text{TEXTURE_BORDER_COLOR} are used in place of the non-existent texel. If the texture contains color components, the values of

OpenGL 3.2 (Core Profile) - July 24, 2009
3.8. TEXTURING

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 border values must be consistent with the type returned by the texture as described in section 3.8, 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 TEXTURE_BORDER_COLOR is interpreted as a depth value.

When the value of TEXTURE_MIN_FILTER is LINEAR, a \(2 \times 2 \times 2\) cube of texels in the image array of level levelbase is selected. Let

\[
\begin{align*}
    i_0 &= \text{wrap}(\lfloor u' - 0.5 \rfloor) \\
    j_0 &= \text{wrap}(\lfloor v' - 0.5 \rfloor) \\
    k_0 &= \text{wrap}(\lfloor w' - 0.5 \rfloor) \\
    i_1 &= \text{wrap}(\lfloor u' - 0.5 \rfloor + 1) \\
    j_1 &= \text{wrap}(\lfloor v' - 0.5 \rfloor + 1) \\
    k_1 &= \text{wrap}(\lfloor w' - 0.5 \rfloor + 1)
\end{align*}
\]

\[
\begin{align*}
    \alpha &= \text{frac}(u' - 0.5) \\
    \beta &= \text{frac}(v' - 0.5) \\
    \gamma &= \text{frac}(w' - 0.5)
\end{align*}
\]

where frac(x) denotes the fractional part of x.

For a three-dimensional texture, the texture value \(\tau\) is found as

\[
\tau = (1 - \alpha)(1 - \beta)(1 - \gamma)\tau_{i_0j_0k_0} + \alpha(1 - \beta)(1 - \gamma)\tau_{i_1j_0k_0} \\
+ (1 - \alpha)\beta(1 - \gamma)\tau_{i_0j_1k_0} + \alpha\beta(1 - \gamma)\tau_{i_1j_1k_0} \\
+ (1 - \alpha)(1 - \beta)\gamma\tau_{i_0j_0k_1} + \alpha(1 - \beta)\gamma\tau_{i_1j_0k_1} \\
+ (1 - \alpha)\beta\gamma\tau_{i_0j_1k_1} + \alpha\beta\gamma\tau_{i_1j_1k_1}
\]

(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,

\[
\tau = (1 - \alpha)(1 - \beta)\tau_{i_0j_0} + \alpha(1 - \beta)\tau_{i_1j_0} \\
+ (1 - \alpha)\beta\tau_{i_0j_1} + \alpha\beta\tau_{i_1j_1}
\]
3.8. TEXTURING

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

\[
l = \text{clamp}([r + 0.5], 0, d_t - 1).
\]

And for a one-dimensional or one-dimensional array texture,

\[
\tau = (1 - \alpha)\tau_i + \alpha\tau_i^1
\]

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

\[
l = \text{clamp}([t + 0.5], 0, h_t - 1).
\]

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 \textsc{nearest} filtering.

**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 \textsc{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 \textsc{texture_min_filter} is \textsc{nearest} or \textsc{linear}, and the value of \textsc{framebuffer_attachment_texture_level} for attachment point \( A \) is equal to the value of \textsc{texture_base_level}

-or-

The value of \textsc{texture_min_filter} is \textsc{nearest_mipmap_nearest}, \textsc{nearest_mipmap_linear}, \textsc{linear_mipmap_nearest}, or \textsc{linear_mipmap_linear}, and the value of \textsc{framebuffer_attachment_texture_level} for attachment point \( A \) is within the the inclusive range from \textsc{texture_base_level} to \( q \).

OpenGL 3.2 (Core Profile) - July 24, 2009
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

$$maxsize = \begin{cases} 
  w_t, & \text{for 1D and 1D array textures} \\
  \max(w_t, h_t), & \text{for 2D, 2D array, and cube map textures} \\
  \max(w_t, h_t, d_t), & \text{for 3D textures}
\end{cases}$$

Numbering the levels such that level $level_{base}$ is the 0th level, the $i$th array has dimensions

$$\max(1, \lfloor \frac{w_t}{w_d} \rfloor) \times \max(1, \lfloor \frac{h_t}{h_d} \rfloor) \times \max(1, \lfloor \frac{d_t}{d_d} \rfloor)$$

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 TexImage3D, TexImage2D, CopyTexImage2D, TexImage1D, or CopyTexImage1D; the array being set is indicated with the level-of-detail argument $level$. Level-of-detail numbers proceed from $level_{base}$ for the original texel array through $p = \lceil \log_2(maxsize) \rceil + level_{base}$ 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. All arrays from $level_{base}$ through $q = \min\{p, level_{max}\}$ must be defined, as discussed in section 3.8.12.

The values of $level_{base}$ and $level_{max}$ may be respecified for a specific texture by calling TexParameter[i] with $pname$ set to TEXTURE_BASE_LEVEL or TEXTURE_MAX_LEVEL respectively.
The error **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 **NEAREST_MIPMAP_NEAREST** and **LINEAR_MIPMAP_NEAREST**, the \( d \)th mipmap array is selected, where

\[
d = \begin{cases} 
  \text{level} \_ \text{base}, & \lambda \leq \frac{1}{2} \\
  \left\lfloor \text{level} \_ \text{base} + \lambda + \frac{1}{2} \right\rfloor - 1, & \lambda > \frac{1}{2}, \text{level} \_ \text{base} + \lambda \leq q + \frac{1}{2} \\
  q, & \lambda > \frac{1}{2}, \text{level} \_ \text{base} + \lambda > q + \frac{1}{2}
\end{cases}
\]  

(3.24)

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

\[
d_1 = \begin{cases} 
  q, & \text{level} \_ \text{base} + \lambda \geq q \\
  \left\lfloor \text{level} \_ \text{base} + \lambda \right\rfloor, & \text{otherwise}
\end{cases}
\]  

(3.25)

\[
d_2 = \begin{cases} 
  q, & \text{level} \_ \text{base} + \lambda \geq q \\
  d_1 + 1, & \text{otherwise}
\end{cases}
\]  

(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

\[
\tau = [1 - \text{frac}(\lambda)] \tau_1 + \text{frac}(\lambda) \tau_2.
\]
3.8. TEXTURING

Manual Mipmap Generation

Mipmaps can be generated manually with the command

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

where `target` is one of `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`, or `TEXTURE_CUBE_MAP`. Mipmap generation affects the texture image attached to `target`. For cube map textures, an `INVALID_OPERATION` error is generated if the texture bound to `target` is not cube complete, as defined in section 3.8.12.

Mipmap generation replaces texel array levels `level_base + 1` through `q` with arrays derived from the `level_base` array, regardless of their previous contents. All other mipmap arrays, including the `level_base` array, are left unchanged by this computation.

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 3.8.12.

The contents of the derived arrays are computed by repeated, filtered reduction of the `level_base` array. For one- and two-dimensional array textures, each layer is filtered independently. No particular filter algorithm is required, though a box filter is recommended as the default filter.

3.8.10 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 3.8.9, 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`.
3.8.11 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.

3.8.12 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 the texture dimensionality.

For multisample textures, a texture is complete if each dimension of the texture’s level is positive.

For one-, two-, or three-dimensional textures and one- or two-dimensional array textures, a texture is complete if the following conditions all hold true:

- The set of mipmap arrays $level_{\text{base}}$ through $q$ (where $q$ is defined in the Mipmapping discussion of section 3.8.9) were each specified with the same internal format.
- The dimensions of the arrays follow the sequence described in the Mipmapping discussion of section 3.8.9.
- $level_{\text{base}} \leq level_{\text{max}}$
- Each dimension of the $level_{\text{base}}$ array is positive.
- If the internal format of the arrays is integer (see table 3.12), `TEXTURE_MAG_FILTER` must be `NEAREST` and `TEXTURE_MIN_FILTER` must be `NEAREST` or `NEAREST_MIPMAP_NEAREST`.

Array levels $k$ where $k < level_{\text{base}}$ or $k > q$ are insignificant to the definition of completeness.

For cube map textures, a texture is cube complete if the following conditions all hold true:

- The $level_{\text{base}}$ arrays of each of the six texture images making up the cube map have identical, positive, and square dimensions.
- The $level_{\text{base}}$ arrays were each specified with the same internal format.

Finally, a cube map texture is mipmap cube complete if, in addition to being cube complete, each of the six texture images considered individually is complete.
3.8. TEXTURING

Effects of Completeness on Texture Application

Texture lookups performed in vertex and fragment shaders are affected by completeness of the texture being sampled as described in sections 2.11.7 and 3.9.2.

Effects of Completeness on Texture Image Specification

An implementation may allow a texture image array of level 1 or greater to be created only if a \textit{mipmap complete} set of image arrays consistent with the requested array can be supported. A mipmap complete set of arrays is equivalent to a complete set of arrays where $level_{base} = 0$ and $level_{max} = 1000$, and where the dimensions of the image array being created are understood to be half the corresponding dimensions of the next lower numbered array (rounded down to the next integer if fractional).

3.8.13 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, and cube map only), and depth (three-dimensional and two-dimensional 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, and an integer size of a compressed image. Each initial texel array is null (zero width, height, and depth, internal format $\text{RGBA}$, component sizes set to zero and component types set to $\text{NONE}$, the compressed flag set to $\text{FALSE}$, and a zero compressed size). Multisample textures contain an integer identifying the number of samples in each texel. 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, and an integer identifying the internal format of the texture, initially $\text{R8}$.

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 \texttt{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 texture is resident, and three integers describing the depth texture mode, compare mode, and compare function. 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 values of TEXTURE_COMPARE_MODE, and TEXTURE_COMPARE_FUNC are NONE, and LEQUAL 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, and one- and two-dimensional 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 and one- and two-dimensional 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.

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,
3.8. TEXTURING

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.

3.8.14 Texture Objects

In addition to the default textures TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_RECTANGLE, TEXTURE_BUFFER, TEXTURE_CUBE_MAP, TEXTURE_2D_MULTISAMPLE, and TEXTURE_2D_MULTISAMPLE_ARRAY, named one-, two-, and three-dimensional, one- and two-dimensional array, rectangular, buffer, cube map, two-dimensional multisample, and two-dimensional multisample array texture objects can be created and operated upon. The name space for texture objects is the unsigned integers, with zero reserved by the GL.

A texture object is created by binding an unused name to one of these texture targets. The binding is effected by calling

```c
void BindTexture( enum target, uint texture );
```

with target set to the desired texture target and texture set to the unused name. The resulting texture object is a new state vector, comprising all the state values listed in section 3.8.13, set to the same initial values. The new texture object bound to target is, and remains a texture of the dimensionality and type specified by target until it is deleted.

BindTexture may also be used to bind an existing texture object to any of these targets. The error INVALID_OPERATION is generated if an attempt is made to bind a texture object of different dimensionality than the specified target. If the bind is successful no change is made to the state of the bound texture object, and any previous binding to target is broken.

BindTexture fails and an INVALID_OPERATION error is generated if texture is not zero or a name returned from a previous call to GenTextures, or if such a name has since been deleted with DeleteTextures.

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
3.8. TEXTURING

When a texture object is bound, the state of the bound texture object directs the texturing operation.

In the initial state, TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_RECTANGLE, TEXTURE_BUFFER, TEXTURE_2D_MULTISAMPLE, and TEXTURE_2D_MULTISAMPLE_ARRAY have one-, two-, and three-dimensional, one- and two-dimensional array, rectangular, buffer, cube map, two-dimensional multisample, and two-dimensional multisample array texture state vectors respectively associated with them. In order that access to these initial textures not be lost, they are treated as texture objects all of whose names are 0. The initial one-, two-, three-dimensional, one- and two-dimensional array, rectangular, buffer, cube map texture is therefore operated upon, queried, and applied as TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_RECTANGLE, TEXTURE_BUFFER, TEXTURE_CUBE_MAP, TEXTURE_2D_MULTISAMPLE, or TEXTURE_2D_MULTISAMPLE_ARRAY respectively while 0 is bound to the corresponding targets.

Texture objects are deleted by calling

void DeleteTextures( sizei n, uint *textures );

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 BindTexture is deleted, it is as though 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 textures are silently ignored, as is the value zero.

The command

void GenTextures( sizei n, uint *textures );

returns n previously unused texture object names in textures. These names are marked as used, for the purposes of GenTextures only, but they acquire texture state and a dimensionality only when they are first bound, just as if they were unused.

The texture object name space, including the initial one-, two-, and three-dimensional, one- and two-dimensional array, rectangular, buffer, cube map, 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 \texttt{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.

### 3.8.15 Texture Comparison Modes

Texture values can also be computed according to a specified comparison function.
Texture parameter \texttt{TEXTURE\_COMPARE\_MODE} specifies the comparison operands,
and parameter \texttt{TEXTURE\_COMPARE\_FUNC} specifies the comparison function.

**Depth Texture Comparison Mode**

If the currently bound texture’s base internal format is \texttt{DEPTH\_COMPONENT} or
\texttt{DEPTH\_STENCIL}, then \texttt{TEXTURE\_COMPARE\_MODE} and \texttt{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 \texttt{TEXTURE\_COMPARE\_MODE} is \texttt{NONE}, then

$$r = D_t$$

If the value of \texttt{TEXTURE\_COMPARE\_MODE} is \texttt{COMPARE\_REF\_TO\_TEXTURE},
then $r$ depends on the texture comparison function as shown in table \ref{table:compare_func}.

The resulting $r$ is assigned to $R_t$.

If the value of \texttt{TEXTURE\_MAG\_FILTER} is not \texttt{NEAREST}, or the value of
\texttt{TEXTURE\_MIN\_FILTER} is not \texttt{NEAREST} or \texttt{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.
3.8. TEXTURING

### Table 3.19: Depth texture comparison functions.

<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>

3.8.16 sRGB Texture Color Conversion

If the currently bound texture’s internal format is one of \texttt{SRGB}, \texttt{SRGB8}, \texttt{SRGB\_ALPHA}, \texttt{SRGB8\_ALPHA8}, \texttt{COMPRESSED\_SRGB}, or \texttt{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 3.8.9 and 3.8.10. 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 inferior 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.

\[
    c_l = \begin{cases} \frac{c_s}{12.92}, & c_s \leq 0.04045 \\ \left( \frac{c_s + 0.055}{1.055} \right)^{2.4}, & c_s > 0.04045 \end{cases}
\]

(3.27)

Assume \( c_s \) is the sRGB component in the range \([0, 1]\).
3.9. FRAGMENT SHADERS

3.8.17 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 3.8.1) are treated as unsigned integers and are converted to red, green, and blue as follows:

\[
\begin{align*}
\text{red} &= \text{red}_s 2^{\text{exp}_\text{shared} - B} \\
\text{green} &= \text{green}_s 2^{\text{exp}_\text{shared} - B} \\
\text{blue} &= \text{blue}_s 2^{\text{exp}_\text{shared} - B}
\end{align*}
\]

3.9 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 2.11.2.

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

3.9.1 Shader Variables

Fragment shaders can access uniforms belonging to the current shader object. The amount of storage available for fragment shader uniform variables in the default uniform block is specified by the value of the implementation-dependent constant MAX_FRAGMENT_UNIFORM_COMPONENTS. The total amount of combined storage available for fragment shader uniform variables in all uniform blocks (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 will
3.9. FRAGMENT SHADERS

consume no more than \(4 \times \min(r,c)\) such values, where \(r\) and \(c\) are the number of rows and columns in the matrix. A link error will be generated if an attempt is made to utilize more than the space available for fragment shader uniform variables.

Fragment shaders can read varying variables that correspond to the attributes of the fragments produced by rasterization. The OpenGL Shading Language Specification defines a set of built-in varying variables that can be be accessed by a fragment shader. These built-in varying variables include data associated with a fragment such as the fragment’s position.

Additionally, when a vertex shader is active, it may define one or more varying variables (see section 2.11.6 and the OpenGL Shading Language Specification). These values are, if not flat shaded, interpolated across the primitive being rendered. The results of these interpolations are available when varying variables of the same name are defined in the fragment shader.

A fragment shader can also write to varying out variables. Values written to these variables are used in the subsequent per-fragment operations. Varying out 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 3.9.2 describes how to direct these values to buffers.

3.9.2 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 2.11.7 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 3.8.9 and 3.8.10, and converts it to a texture source color \(C_s\) according to table 3.20. The GL returns a four-component vector \((R_s, G_s, B_s, A_s)\) to the fragment shader. For the purposes of level-of-detail calculations, the derivatives \(\frac{du}{dx}, \frac{du}{dy}, \frac{dv}{dx}, \frac{dv}{dy}, \frac{dw}{dx}\) and \(\frac{dw}{dy}\) 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 can either return the depth data directly or return the results of a comparison with the \(D_{ref}\) value (see section 3.8.15) used to perform the lookup. The comparison operation is requested.
3.9. FRAGMENT SHADERS

<table>
<thead>
<tr>
<th>Texture Base Internal Format</th>
<th>Texture source color</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED</td>
<td>$(R_t, 0, 0)$</td>
</tr>
<tr>
<td>RG</td>
<td>$(R_t, G_t, 0)$</td>
</tr>
<tr>
<td>RGB</td>
<td>$(R_t, G_t, B_t)$</td>
</tr>
<tr>
<td>RGBA</td>
<td>$(R_t, G_t, B_t)$</td>
</tr>
</tbody>
</table>

Table 3.20: Correspondence of filtered texture components to texture source components.

in the shader by using any of the shadow sampler types (sampler1DShadow, sampler2DShadow, or sampler2DRectShadow), and in the texture using the TEXTURE_COMPARE_MODE parameter. These requests must be consistent; the results of a texture lookup are undefined if:

- 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 fragment shader uses a sampler whose associated texture object is not complete, as defined in section 3.8.12, the texture image unit will return $(R, G, B, A) = (0, 0, 0, 1)$.

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 `MAX_TEXTURE_IMAGE_UNITS`.

**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:

\[
\begin{align*}
x_f &= \begin{cases} 
x_w - \frac{1}{2}, & \text{pixel-center convention is integer} \\
x_w, & \text{otherwise} 
\end{cases} \\
y'_f &= \begin{cases} 
H - y_w, & \text{origin convention is upper-left} \\
y_w, & \text{otherwise} 
\end{cases} \\
y_f &= \begin{cases} 
y'_f - \frac{1}{2}, & \text{pixel-center convention is integer} \\
y'_f & \text{otherwise} 
\end{cases} \\
z_f &= z_w \\
w_f &= \frac{1}{w_c}
\end{align*}
\]  

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 2.13).

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 \texttt{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...
3.9. **FRAGMENT SHADERS**

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.

`gl_PrimitiveID` is only defined under the same conditions that `gl_VertexID` is defined, as described under “Shader Inputs” in section 2.11.7.

Similarly to the limit on geometry shader output components (see section 2.12.4), there is a limit on the number of components of built-in and user-defined input varying variables that can be read by the fragment shader, given by the value of the implementation-dependent constant `MAX_FRAGMENT_INPUT_COMPONENTS`.

When a program is linked, all components of any varying and special variable read by a fragment shader will count against this limit. A program whose fragment shader reads more than the value of `MAX_FRAGMENT_INPUT_COMPONENTS` components worth of varying variables may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

**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 varying out variables and the built-in variables `gl_FragColor`, `gl_FragData[n]`, and `gl_FragDepth`. 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 2.13.1). 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.

Color values written by a fragment shader may be floating-point, signed integer, or unsigned integer. If the color buffer has an 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 2.6 or 2.4, 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.

Writing to `gl_FragColor` specifies the fragment color (color number zero) that will be used by subsequent stages of the pipeline. Writing to `gl_FragData[n]` specifies the value of fragment color number n. Any colors, or color components, associated with a fragment that are not written by the fragment shader are undefined. A fragment shader may not statically assign values to
more than one of \texttt{gl\_FragColor}, \texttt{gl\_FragData}, and any user-defined varying out 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 varying out variable to a fragment color number can be specified explicitly. The command

\begin{verbatim}
void BindFragDataLocation( uint program,
    uint colorNumber, const char *name );
\end{verbatim}

specifies that the varying out variable \texttt{name} in \texttt{program} should be bound to fragment color \texttt{colorNumber} when the program is next linked. If \texttt{name} was bound previously, its assigned binding is replaced with \texttt{colorNumber}. \texttt{name} must be a null-terminated string. The error \texttt{INVALID\_VALUE} is generated if \texttt{colorNumber} is equal or greater than \texttt{MAX\_DRAW\_BUFFERS}. \texttt{BindFragDataLocation} has no effect until the program is linked. In particular, it doesn’t modify the bindings of varying out variables 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 varying out variables without a binding specified through \texttt{BindFragDataLocation} will automatically be bound to fragment colors by the GL. Such bindings can be queried using the command \texttt{GetFragDataLocation}. \texttt{LinkProgram} will fail if the number of active outputs is greater than the value of \texttt{MAX\_DRAW\_BUFFERS}. \texttt{LinkProgram} will also fail if more than one varying out variable is bound to the same number. This type of aliasing is not allowed.

\texttt{BindFragDataLocation} 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, including a name that is never used as a varying out variable 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 varying out variable names to color numbers can be queried. The command

\begin{verbatim}
OpenGL 3.2 (Core Profile) - July 24, 2009
\end{verbatim}
int GetFragDataLocation(uint program, const char *name);

returns the number of the fragment color to which the varying out variable name was bound when the program object program was last linked. name must be a null-terminated string. If program has not been successfully linked, the error INVALID_OPERATION is generated. If name is not a varying out variable, or if an error occurs, -1 will be returned.

### 3.10 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.

### 3.11 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).
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 2.13.1 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\(^1\), 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

\(^1\)The 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 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. These values are set with

Figure 4.1. Per-fragment operations.
4.1. PER-FRAGMENT OPERATIONS

```c
void Scissor( int left, int bottom, sizei width, sizei height );
```

If \( left \leq x_w < left + width \) and \( bottom \leq y_w < bottom + height \), then the scissor test passes. Otherwise, the test fails and the fragment is discarded. The test is enabled or disabled using `Enable` or `Disable` using the constant `SCISSOR_TEST`. When disabled, it is as if the scissor test always passes. If either `width` or `height` is less than zero, then the error `INVALID_VALUE` is generated. The state required consists of four integer values and a bit indicating whether the test is enabled or disabled. In the initial state, \( left = bottom = 0, width \) and `height` 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 `width` and `height` are initially set to zero. Initially, the scissor test is disabled.

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`, and `SAMPLE_MASK_VALUE`. 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.

`SAMPLE_ALPHA_TO_COVERAGE`, `SAMPLE_ALPHA_TO_ONE`, and `SAMPLE_COVERAGE` are enabled and disabled by calling `Enable` and `Disable` with `cap` specified as one of the three token values. All three values are queried by calling `IsEnabled` with `cap` set to the desired token value. If `SAMPLE_ALPHA_TO_COVERAGE` is enabled and the color buffer has a fixed-point or floating-point format, 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. Otherwise the fragment coverage value is unchanged at this point. If multiple colors are written by a fragment shader, the alpha value of fragment color zero is used to determine the temporary coverage value.

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\texttt{SAMPLE\_ALPHA\_TO\_ONE} is enabled, each alpha value is replaced
by the maximum representable alpha value. Otherwise, the alpha values are not
changed.

Next, if\texttt{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\texttt{SAMPLE\_-
COVERAGE\_VALUE}. The function need not be identical, but it must have the same
properties of proportionality and invariance. If\texttt{SAMPLE\_COVERAGE\_INVERT}
is\texttt{TRUE}, the temporary coverage is inverted (all bit values are inverted) before it is
ANDed with the fragment coverage.

The values of\texttt{SAMPLE\_COVERAGE\_VALUE} and\texttt{SAMPLE\_COVERAGE\_INVERT}
are specified by calling

\begin{verbatim}
void SampleCoverage( clampf value, boolean invert);
\end{verbatim}

with\texttt{value} set to the desired coverage value, and\texttt{invert} set to \texttt{TRUE} or \texttt{FALSE}.
\texttt{value} is clamped to [0,1] before being stored as\texttt{SAMPLE\_COVERAGE\_VALUE}.
\texttt{SAMPLE\_COVERAGE\_VALUE} is queried by calling\texttt{GetFloatv} with\texttt{pname} set to
\texttt{SAMPLE\_COVERAGE\_VALUE}.\texttt{SAMPLE\_COVERAGE\_INVERT} is queried by calling
\texttt{GetBooleanv} with\texttt{pname} set to\texttt{SAMPLE\_COVERAGE\_INVERT}.

Finally, if\texttt{SAMPLE\_MASK} is enabled, the fragment coverage is ANDed with
the coverage value\texttt{SAMPLE\_MASK\_VALUE}. The value of\texttt{SAMPLE\_MASK\_VALUE} is
specified using

\begin{verbatim}
void SampleMaski( uint maskNumber, bitfield mask);
\end{verbatim}

with\texttt{mask} set to the desired mask for mask word\texttt{maskNumber}.\texttt{SAMPLE\_MASK\_-
VALUE} is queried by calling\texttt{GetIntegeri} with\texttt{pname} set to\texttt{SAMPLE\_MASK\_-
VALUE} and the index set to\texttt{maskNumber}. Bit\texttt{B} of mask word\texttt{M} corresponds to
sample $32 \times M + B$ as described in section 3.3.1. The error\texttt{INVALID\_VALUE} is
generated if the mask word indexed is greater than or equal to the value of\texttt{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\texttt{Enable} and\texttt{Disable} commands,
using the symbolic constant \texttt{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

\begin{verbatim}
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 dppass );
void StencilOpSeparate( enum face, enum sfail, enum dpfail,
                        enum dppass );
\end{verbatim}

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 two-sided lighting and face culling (see section 3.6.1).

\texttt{StencilFuncSeparate} and \texttt{StencilOpSeparate} take a \texttt{face} argument which can be \texttt{FRONT}, \texttt{BACK}, or \texttt{FRONT\_AND\_BACK} and indicates which set of state is affected. \texttt{StencilFunc} and \texttt{StencilOp} set front and back stencil state to identical values.

\texttt{StencilFunc} and \texttt{StencilFuncSeparate} take three arguments that control whether the stencil test passes or fails. \texttt{ref} is an integer reference value that is used in the unsigned stencil comparison. Stencil comparison operations and queries of \texttt{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 \texttt{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 \texttt{func}. \texttt{func} is a symbolic constant that determines the stencil comparison function; the eight symbolic constants are \texttt{NEVER}, \texttt{ALWAYS}, \texttt{LESS}, \texttt{LEQUAL}, \texttt{EQUAL}, \texttt{GEQUAL}, \texttt{GREATER}, or \texttt{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.

\texttt{StencilOp} and \texttt{StencilOpSeparate} take three arguments that indicate what happens to the stored stencil value if this or certain subsequent tests fail or pass. \texttt{sfail} indicates what action is taken if the stencil test fails. The symbolic constants are \texttt{KEEP}, \texttt{ZERO}, \texttt{REPLACE}, \texttt{INCR}, \texttt{DECR}, \texttt{INVERT}, \texttt{INCR\_WRAP}, and \texttt{DECR\_WRAP}.  

OpenGL 3.2 (Core Profile) - July 24, 2009
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 stenciling 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.

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
4.1. PER-FRAGMENT OPERATIONS

the depth value stored at the location given by the incoming fragment’s $(x_w, y_w)$ coordinates.

If depth clamping (see section 2.19) is enabled, before the incoming fragment’s $z_w$ is compared $z_w$ is clamped to the range $[\text{min}(n, f), \text{max}(n, f)]$, where $n$ and $f$ are the current near and far depth range values (see section 2.13.1)

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.

When an occlusion query is started, 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.

If the samples-passed count overflows (exceeds the value $2^n - 1$, where $n$ is the number of bits in the samples-passed count), 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.

The necessary state is a single bit indicating whether an occlusion query is active, the identifier of the currently active occlusion query, and a counter keeping track of the number of samples that have passed.
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 clamped to \([0, 1]\) 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

```c
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 equations, defined by the commands

```c
void BlendEquation(enum mode);
```
void BlendEquationSeparate( enum modeRGB,
enum modeAlpha );

BlendEquationSeparate argument modeRGB determines the RGB blend function while modeAlpha determines the alpha blend equation. BlendEquation argument mode determines both the RGB and alpha blend equations. modeRGB and modeAlpha must each be one of FUNC_ADD, FUNC_SUBTRACT, FUNC_REVERSE_SUBTRACT, MIN, or MAX.

Signed or unsigned normalized fixed-point destination (framebuffer) components are represented as described in section 2.1.5. 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 2.1.5.

Prior to blending, signed and unsigned normalized fixed-point color components undergo an implied conversion to floating-point using equations 2.1 and 2.3, respectively. This conversion must leave the values 0 and 1 invariant. Blending computations are treated as if carried out in floating-point.

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 3.8.16.

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,
4.1. PER-FRAGMENT OPERATIONS

<table>
<thead>
<tr>
<th>Mode</th>
<th>RGB Components</th>
<th>Alpha Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUNC_ADD</td>
<td>$R = R_s \times S_r + R_d \times D_r$</td>
<td>$A = A_s \times S_a + A_d \times D_a$</td>
</tr>
<tr>
<td></td>
<td>$G = G_s \times S_g + G_d \times D_g$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$B = B_s \times S_b + B_d \times D_b$</td>
<td></td>
</tr>
<tr>
<td>FUNC_SUBTRACT</td>
<td>$R = R_s \times S_r - R_d \times D_r$</td>
<td>$A = A_s \times S_a - A_d \times D_a$</td>
</tr>
<tr>
<td></td>
<td>$G = G_s \times S_g - G_d \times D_g$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$B = B_s \times S_b - B_d \times D_b$</td>
<td></td>
</tr>
<tr>
<td>FUNC_REVERSE_SUBTRACT</td>
<td>$R = R_d \times D_r - R_s \times S_r$</td>
<td>$A = A_d \times D_a - A_s \times S_a$</td>
</tr>
<tr>
<td></td>
<td>$G = G_d \times D_g - G_s \times S_g$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$B = B_d \times D_b - B_s \times S_b$</td>
<td></td>
</tr>
<tr>
<td>MIN</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>MAX</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>

Table 4.1: RGB and alpha blend equations.

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.

**Blend Functions**

The weighting factors used by the blend equation are determined by the blend functions. Blend functions are specified with the commands

```c
void BlendFuncSeparate( enum srcRGB, enum dstRGB,
                        enum srcAlpha, enum dstAlpha );
void BlendFunc( enum src, enum dst );
```

**BlendFuncSeparate** 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** argument src determines both RGB and alpha source functions, while dst determines both RGB and alpha destination functions.

The possible source and destination blend functions and their corresponding computed blend factors are summarized in table 4.2.
### 4.1. PER-FRAGMENT OPERATIONS

<table>
<thead>
<tr>
<th>Function</th>
<th>RGB Blend Factors</th>
<th>Alpha Blend Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ZERO</strong></td>
<td>(0, 0, 0)</td>
<td>0</td>
</tr>
<tr>
<td><strong>ONE</strong></td>
<td>(1, 1, 1)</td>
<td>1</td>
</tr>
<tr>
<td><strong>SRC_COLOR</strong></td>
<td>((R_s, G_s, B_s))</td>
<td>(A_s)</td>
</tr>
<tr>
<td><strong>ONE_MINUS_SRC_COLOR</strong></td>
<td>((1, 1, 1) - (R_s, G_s, B_s))</td>
<td>(1 - A_s)</td>
</tr>
<tr>
<td><strong>DST_COLOR</strong></td>
<td>((R_d, G_d, B_d))</td>
<td>(A_d)</td>
</tr>
<tr>
<td><strong>ONE_MINUS_DST_COLOR</strong></td>
<td>((1, 1, 1) - (R_d, G_d, B_d))</td>
<td>(1 - A_d)</td>
</tr>
<tr>
<td><strong>SRC_ALPHA</strong></td>
<td>((A_s, A_s, A_s))</td>
<td>(A_s)</td>
</tr>
<tr>
<td><strong>ONE_MINUS_SRC_ALPHA</strong></td>
<td>((1, 1, 1) - (A_s, A_s, A_s))</td>
<td>(1 - A_s)</td>
</tr>
<tr>
<td><strong>DST_ALPHA</strong></td>
<td>((A_d, A_d, A_d))</td>
<td>(A_d)</td>
</tr>
<tr>
<td><strong>ONE_MINUS_DST_ALPHA</strong></td>
<td>((1, 1, 1) - (A_d, A_d, A_d))</td>
<td>(1 - A_d)</td>
</tr>
<tr>
<td><strong>CONSTANT_COLOR</strong></td>
<td>((R_c, G_c, B_c))</td>
<td>(A_c)</td>
</tr>
<tr>
<td><strong>ONE_MINUS_CONSTANT_COLOR</strong></td>
<td>((1, 1, 1) - (R_c, G_c, B_c))</td>
<td>(1 - A_c)</td>
</tr>
<tr>
<td><strong>CONSTANT_ALPHA</strong></td>
<td>((A_c, A_c, A_c))</td>
<td>(A_c)</td>
</tr>
<tr>
<td><strong>ONE_MINUS_CONSTANT_ALPHA</strong></td>
<td>((1, 1, 1) - (A_c, A_c, A_c))</td>
<td>(1 - A_c)</td>
</tr>
<tr>
<td><strong>SRC_ALPHA_SATURATE</strong></td>
<td>((f, f, f)^2)</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

1. **SRC_ALPHA_SATURATE** is valid only for source RGB and alpha blending functions.
2. \(f = \min(A_s, 1 - A_d)\).

OpenGL 3.2 (Core Profile) - July 24, 2009
4.1. PER-FRAGMENT OPERATIONS

**Blend Color**

The constant color \( C_c \) to be used in blending is specified with the command

```c
void BlendColor(clampf red, clampf green, clampf blue, clampf alpha);
```

The constant color can be used in both the source and destination blending functions.

The state required for blending is two integers for the RGB and alpha blend equations, four integers indicating the source and destination RGB and alpha blending functions, four floating-point values to store the RGBA constant blend color, and a bit indicating whether blending is enabled or disabled for each of the `MAX_DRAW_BUFFERS` draw buffers.

The initial blend equations for RGB and alpha are both `FUNC_ADD`. The initial blending functions are `ONE` for the source RGB and alpha functions and `ZERO` for the destination RGB and alpha functions. The initial constant blend color is \((R, G, B, A) = (0, 0, 0, 0)\). Initially, blending is disabled for all draw buffers.

The value of the blend enable for draw buffer \( i \) can be queried by calling `IsEnabled` with `target BLEND` and `index i`. The value of the blend enable for draw buffer zero may also be queried by calling `IsEnabled` with `value BLEND`.

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 `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 values after blending are converted into the non-linear sRGB color space by computing

\[
\begin{align*}
c_s &= \begin{cases} 
0.0, & c_l \leq 0 \\
12.92c_l, & 0 < c_l < 0.0031308 \\
1.055c_l^{0.41666} - 0.055, & 0.0031308 \leq c_l < 1 \\
1.0, & c_l \geq 1
\end{cases}
\end{align*}
\]

(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 `FRAMEBUFFER_SRGB` is disabled or the value of `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` is not `SRGB`, then

OpenGL 3.2 (Core Profile) - July 24, 2009
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 2.4. 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 positive representable color value (for that particular color component) that is less than or equal to the incoming color component value, \(c\), or the smallest negative 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.

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 each incoming color component \(c\) is replaced with the largest positive representable color value (for that particular component) that is less than or equal to \(c\), or by the smallest negative representable value, if no representable value is less than or equal to \(c\).

Dithering is enabled with Enable and disabled with Disable using the symbolic constant 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 \(x_w\) and \(y_w\) 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 Enable or Disable using the symbolic constant COLOR_LOGIC_OP. If the logical operation is enabled for color values, it is as if blending were disabled, regardless of the value of 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.
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.

Logical operation has no effect on a floating-point destination color buffer. However, if logical operation is enabled, blending is still disabled.

The logical operation is selected by

```c
void LogicOp(enum op);
```

*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 **COPY**, and to be disabled.
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 buffer (to be described in a following section). 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 sample, 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.

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
4.2. WHOLE FRAMEBUFFER OPERATIONS

the whole framebuffer.

4.2.1 Selecting a Buffer 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 val-
ues 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
the default framebuffer. For more information about framebuffer objects, see sec-
tion 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 a 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_ATTACHMENT 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 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 framebuffer object and `buf` is one of the constants
from table 4.4, 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.
4.2. WHOLE FRAMEBUFFER OPERATIONS

<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 4.4: Arguments to DrawBuffer(s) and ReadBuffer when the context is bound to a default framebuffer, and the buffers they indicate.

<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_ATTACHMENTi (see caption)</td>
<td>Output fragment color to image attached at color attachment point i</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_ATTACHMENTi may range from zero to the value of MAX_COLOR_ATTACHMENTS - 1.
4.2. WHOLE FRAMEBUFFER OPERATIONS

<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>

Table 4.6: Arguments to DrawBuffers when the context is bound to the default framebuffer, and the buffers they indicate.

**DrawBuffer** will set the draw buffer for fragment colors other than zero to NONE.

The command

```c
void DrawBuffers(sizei n, const enum *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.

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 an 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 **GetIntegerv** 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 then once in the array pointed
4.2. WHOLE FRAMEBUFFER OPERATIONS

If a fragment shader writes to gl_FragColor, DrawBuffers specifies a set of draw buffers into which the single fragment color defined by gl_FragColor is written. If a fragment shader writes to gl_FragData, or a user-defined varying out 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 none of gl_FragColor, gl_FragData, nor any user-defined varying out 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_OPERATION. 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 framebuffer object and DrawBuffers is supplied with a constant from table 4.6, or COLOR_ATTACHMENT m 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.

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 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 then zero is NONE.

The value of the draw buffer selected for fragment color i can be queried by

OpenGL 3.2 (Core Profile) - July 24, 2009
4.2. WHOLE FRAMEBUFFER OPERATIONS

calling \texttt{GetIntegerv} with the symbolic constant \texttt{DRAW_BUFFERi}. \texttt{DRAW_BUFFERi} 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 \textit{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_BUFFERi} is modified by passing \texttt{i} as the parameter \texttt{buf}. \texttt{r}, \texttt{g}, \texttt{b}, and \texttt{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 \texttt{r}, \texttt{g}, \texttt{b}, and \texttt{a} is applied to the color buffer associated with \texttt{DRAW_BUFFERi}. If \texttt{DRAW_BUFFERi} 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 \texttt{r}, \texttt{g}, \texttt{b}, and \texttt{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 \texttt{i} can be queried by calling \texttt{GetBooleani} with \texttt{target COLOR_WRITEMASK} and \texttt{index i}. The value of the color writemask for draw buffer zero may also be queried by calling \texttt{GetBooleani} with \texttt{value COLOR_WRITEMASK}.

The depth buffer can be enabled or disabled for writing \texttt{zw} values using

\begin{verbatim}
void DepthMask( boolean mask );
\end{verbatim}

If \texttt{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

\begin{verbatim}
void StencilMask( uint mask );
\end{verbatim}
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.

### 4.2.3 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 the bitwise OR of a number of values indicating which buffers are to be cleared. The values are COLOR_BUFFER_BIT, DEPTH_BUFFER_BIT, STENCIL_BUFFER_BIT, and 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 the mask is not a bitwise OR of the specified values, then the error INVALID_VALUE is generated.
4.2. WHOLE FRAMEBUFFER OPERATIONS

void ClearColor(clampf r, clampf g, clampf b, clampf a);

sets the clear value for fixed- and floating-point color buffers. The specified components are stored as floating-point values.

The command

void ClearDepth(clampf d);

sets the depth value used when clearing the depth buffer. d is clamped to the range [0, 1]. When clearing a fixed-point depth buffer, d is converted to fixed-point according to the rules for a window z value given in section 2.13.1. 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, 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 to fixed-point using equations 2.4 or 2.6, 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

void ClearBuffer(if ui)v(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_BUFFERi 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 \texttt{FRONT}, \texttt{BACK}, \texttt{LEFT}, \texttt{RIGHT}, or \texttt{FRONT\_AND\_BACK}, identifying multiple buffers, each selected buffer is cleared to the same value. The \texttt{ClearBufferfv}, \texttt{ClearBufferiv}, and \texttt{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 \texttt{ClearColor}.

If \texttt{buffer} is \texttt{DEPTH}, \texttt{drawbuffer} must be zero, and \texttt{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 \texttt{ClearDepth}. Only \texttt{ClearBufferfv} should be used to clear depth buffers.

If \texttt{buffer} is \texttt{STENCIL}, \texttt{drawbuffer} must be zero, and \texttt{value} points to the single stencil value to clear the stencil buffer to. Masking and type conversion are performed in the same fashion as \texttt{ClearStencil}. Only \texttt{ClearBufferiv} should be used to clear stencil buffers.

The command

\begin{verbatim}
void ClearBufferfi( enum buffer, int drawbuffer,
                  float depth, int stencil );
\end{verbatim}
clears both depth and stencil buffers of the currently bound draw framebuffer. \texttt{buffer} must be \texttt{DEPTH\_STENCIL} and \texttt{drawbuffer} must be zero. \texttt{depth} and \texttt{stencil} are the values to clear the depth and stencil buffers to, respectively. Clamping and type conversion of \texttt{depth} for fixed-point depth buffers is performed in the same fashion as \texttt{ClearDepth}. Masking of \texttt{stencil} for stencil buffers is performed in the same fashion as \texttt{ClearStencil}. \texttt{ClearBufferfi} is equivalent to clearing the depth and stencil buffers separately, but may be faster when a buffer of internal format \texttt{DEPTH\_STENCIL} is being cleared.

The result of \texttt{ClearBuffer} is undefined if no conversion between the type of the specified \texttt{value} and the type of the buffer being cleared is defined (for example, if \texttt{ClearBufferiv} is called for a fixed- or floating-point buffer, or if \texttt{ClearBufferfv} is called for a signed or unsigned integer buffer). This is not an error.

When \texttt{ClearBuffer} is called, the same per-fragment and masking operations defined for \texttt{Clear} are applied.

**Errors**

- \texttt{ClearBuffer\{iufi\}} generates an \texttt{INVALID\_ENUM} error if \texttt{buffer} is not \texttt{COLOR}, \texttt{DEPTH}, or \texttt{STENCIL}. \texttt{ClearBufferfi} generates an \texttt{INVALID\_ENUM} error if \texttt{buffer} is not \texttt{DEPTH\_STENCIL}.

- \texttt{ClearBuffer} generates an \texttt{INVALID\_VALUE} error if \texttt{buffer} is \texttt{COLOR} and \texttt{drawbuffer} is less than zero, or greater than the value of \texttt{MAX\_DRAW\_BUFFERS} minus one; or if \texttt{buffer} is \texttt{DEPTH}, \texttt{STENCIL}, or \texttt{DEPTH\_STENCIL} and \texttt{drawbuffer} is not zero.
4.3. READING AND COPYING PIXELS

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 multisample buffers corresponding to the specified buffer.

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.

4.3.1 Reading 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, sizei width, sizei 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.

**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.
4.3. READING AND COPYING PIXELS

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.

<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>
</tbody>
</table>

Table 4.7: **PixelStore** parameters pertaining to **ReadPixels**, and **GetTexImage**.

OpenGL 3.2 (Core Profile) - July 24, 2009
4.3. READING AND COPYING PIXELS

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 samples in this buffer. It is recommended that the stencil value of the centermost sample 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

```c
void ReadBuffer( enum src );
```

takes a symbolic constant as argument. src must be one of the values from tables 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 framebuffer (i.e., READ_FRAMEBUFFER_BINDING is zero), or a framebuffer object (i.e., READ_FRAMEBUFFER_BINDING is non-zero). For more information about framebuffer objects, see section 4.4.

If the object bound to READ_FRAMEBUFFER_BINDING is not framebuffer complete (as defined in section 4.4.4), then ReadPixels generates the error INVALID_FRAMEBUFFER_OPERATION. If ReadBuffer is supplied with a constant that is neither legal for the default framebuffer, nor legal for a framebuffer object, then the error INVALID_ENUM results.
4.3. READING AND COPYING PIXELS

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 and `BACK` otherwise.

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_ATTACHMENT` enables reading from the image attached to the framebuffer at `COLOR_ATTACHMENT`. For framebuffer objects, the initial setting for `ReadBuffer` is `COLOR_ATTACHMENT0`.

`ReadPixels` generates an `INVALID_OPERATION` error if it attempts to select a color buffer while `READ_BUFFER` is `NONE`.

`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 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`, `ALPHA`, `RG`, `RGB`, `RGBA`, `BGR`, or `BGRA`, then red, green, blue, and alpha values are obtained from the selected buffer at each pixel location. If the framebuffer does not support alpha values then the A that is obtained is 1.0.

If `format` is an integer format and the color buffer is not an integer format; 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` or `HALF_FLOAT`, the error `INVALID_OPERATION` occurs.

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.
4.3. READING AND COPYING PIXELS

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 2.3 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 2.13.1). No conversion is necessary if the depth buffer uses a floating-point representation.

Final Conversion

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 4.8; if the type is FLOAT or HALF_FLOAT, then the integer index is converted to a GL float or half data value.

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 a floating-point RGBA color, if type is not one of FLOAT, HALF_FLOAT, UNSIGNED_INT_5_9_9_9_REV, or UNSIGNED_INT_10F_11F_11F_REV; or if read color clamping is enabled, each component is first clamped to \([0, 1]\). Then the appropriate conversion formula from table 4.9 is applied to the component.

In the special case of calling ReadPixels with type of UNSIGNED_INT_10F_11F_11F_REV and format of RGB, conversion is performed as follows: the returned data are 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 2.1.3 and 2.1.4. 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.

In the special case of calling **ReadPixels** with **type** of **UNSIGNED_INT_5_9_9_9_REV** and **format** **RGB**, the conversion is performed as follows: the returned data are packed into a series of uint values. The red, green, and blue components are converted to **red**, **green**, **blue**, and **exp** shared integers as described in section 3.8.1 when **internalformat** is **RGB9_E5**. The **red**, **green**, **blue**, and **exp** shared 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 an integer RGBA color, each component is clamped to the representable range of **type**.

## 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

### Table 4.8: Index masks used by **ReadPixels**. Floating point data are not masked.

<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_REV</td>
<td>(2^8 - 1)</td>
</tr>
</tbody>
</table>
### 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>$c = (2^8 - 1)f$</td>
</tr>
<tr>
<td>BYTE</td>
<td>byte</td>
<td>$c = (2^8 - 1)f - \frac{1}{2}$</td>
</tr>
<tr>
<td>UNSIGNED_SHORT</td>
<td>ushort</td>
<td>$c = (2^{16} - 1)f$</td>
</tr>
<tr>
<td>SHORT</td>
<td>short</td>
<td>$c = \frac{2^{16} - 1}{2}f$</td>
</tr>
<tr>
<td>UNSIGNED_INT</td>
<td>uint</td>
<td>$c = (2^{32} - 1)f$</td>
</tr>
<tr>
<td>INT</td>
<td>int</td>
<td>$c = \frac{2^{32} - 1}{2}f$</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>$c = (2^N - 1)f$</td>
</tr>
<tr>
<td>UNSIGNED_BYTE_2_3_3_REV</td>
<td>ubyte</td>
<td>$c = (2^N - 1)f$</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_6_5</td>
<td>ushort</td>
<td>$c = (2^N - 1)f$</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_6_5_REV</td>
<td>ushort</td>
<td>$c = (2^N - 1)f$</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_4_4_4</td>
<td>ushort</td>
<td>$c = (2^N - 1)f$</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_4_4_4_REV</td>
<td>ushort</td>
<td>$c = (2^N - 1)f$</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5_5_5</td>
<td>ushort</td>
<td>$c = (2^N - 1)f$</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_1_5_5_5</td>
<td>ushort</td>
<td>$c = (2^N - 1)f$</td>
</tr>
<tr>
<td>UNSIGNED_INT_8_8_8_8</td>
<td>uint</td>
<td>$c = (2^{24} - 1)f$</td>
</tr>
<tr>
<td>UNSIGNED_INT_8_8_8_8_REV</td>
<td>uint</td>
<td>$c = (2^{24} - 1)f$</td>
</tr>
<tr>
<td>UNSIGNED_INT_10_10_10_2</td>
<td>uint</td>
<td>$c = (2^{24} - 1)f$</td>
</tr>
<tr>
<td>UNSIGNED_INT_2_10_10_10_2</td>
<td>uint</td>
<td>$c = (2^{24} - 1)f$</td>
</tr>
<tr>
<td>UNSIGNED_INT_24_8</td>
<td>uint</td>
<td>$c = (2^{24} - 1)f$</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 4.9: Reversed component conversions, used when component data are being returned to client memory. Color, normal, and depth components are converted from the internal floating-point representation ($f$) to a datum of the specified GL data type ($c$) using the specified equation. 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.) Equations with $N$ as the exponent are performed for each bitfield of the packed data type, with $N$ set to the number of bits in the bitfield.
4.3. READING AND COPYING PIXELS

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, BLUE, or ALPHA, 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);
```

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 $(srcX0, srcY0)$ and $(srcX1, srcY1)$ to the destination rectangle bounded by the locations $(dstX0, dstY0)$ and $(dstX1, dstY1)$. 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 destination dimensions are identical, no filtering is applied. If either the source or destination rectangle specifies a negative width or height ($X_1 < X_0$ or $Y_1 < Y_0$), the image is reversed in the corresponding direction. If both the source and destination 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 LINEAR sampling would require sampling outside the bounds of a source buffer, it is as though CLAMP_TO_EDGE texture sampling were being performed. If a linear filter is selected 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 destination rectangles overlap, the result of the blit operation is undefined.

Blit operations bypass the fragment pipeline. The only fragment operations which affect a blit are the pixel ownership test and the scissor test.

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 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 mask includes COLOR_BUFFER_BIT, pixel groups are converted to match the destination format. However, colors are clamped only if all draw color buffers have fixed-point components. Format conversion is not supported for all data types, and an INVALID_OPERATION error is generated under any of the following conditions:

- The read buffer contains floating-point values and any draw buffer does not contain floating-point values.
- The read buffer contains non-floating-point values and any draw buffer contains 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.
4.3. READING AND COPYING PIXELS

Calling `BlitFramebuffer` will result in an `INVALID_FRAMEBUFFER_OPERATION` error if the objects bound to `DRAW_FRAMEBUFFER_BINDING` and `READ_FRAMEBUFFER_BINDING` are not framebuffer complete (section 4.4.4).

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, if the formats of the read and draw framebuffers are not identical, or if the values of `SAMPLES` for the read and draw buffers 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, if the formats differ, or if the source and destination rectangles are not defined with the same \((X_0, Y_0)\) and \((X_1, Y_1)\) 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 tables 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 \texttt{FIXED\_ONLY}. State set with \texttt{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. GL defines 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 \textit{off-screen} rendering. Further, by allowing the images of a texture to be attached to a framebuffer, the GL provides a mechanism to support \textit{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 namespace 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 \texttt{GenFramebuffers} (see below) to \texttt{DRAW\_FRAMEBUFFER} or \texttt{READ\_FRAMEBUFFER}. The binding is effected by calling

```c
void BindFramebuffer(enum target, uint framebuffer);
```

OpenGL 3.2 (Core Profile) - July 24, 2009
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 values listed in table 6.22, as well as one set of the state values listed in table 6.23 for each attachment point of the framebuffer, set to the same initial values. There are **MAX_COLOR_ATTACHMENTs** color attachment points, plus one each for the depth and stencil attachment points.

**BindFramebuffer** may also be used to bind an existing framebuffer object to **DRAW_FRAMEBUFFER** and/or **READ_FRAMEBUFFER**. If the bind is successful no change is made to the state of the bound framebuffer object, and any previous binding to **target** is broken.

**BindFramebuffer** fails and an **INVALID_OPERATION** error is generated if **framebuffer** is not zero or a name returned from a previous call to **GenFramebuffers**, or if such a name has since been deleted with **DeleteFramebuffers**.

If a framebuffer object is bound to **DRAW_FRAMEBUFFER** or **READ_FRAMEBUFFER**, it becomes the target for rendering or readback operations, respectively, until it is deleted or another framebuffer is bound to the corresponding bind point. Calling **BindFramebuffer** with **target** set to **FRAMEBUFFER** binds **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 6.46 and 6.25 are derived from the framebuffer object bound to **DRAW_FRAMEBUFFER**.

The initial state of **DRAW_FRAMEBUFFER** and **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,
4.4. FRAMEBUFFER OBJECTS

- 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, 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* are silently ignored, as is the value zero.

The command

```c
void GenFramebuffers( sizei n, uint *ids );
```
returns \( n \) previously unused framebuffer object names in \( ids \). These names are marked as used, for the purposes of \texttt{GenFramebuffers} only, but they acquire state and type only when they are first bound, just as if they were unused.

The names bound to the draw and read framebuffer bindings can be queried by calling \texttt{GetIntegerv} with the symbolic constants \texttt{DRAW_FRAMEBUFFER_BINDING} and \texttt{READ_FRAMEBUFFER_BINDING}, respectively. \texttt{FRAMEBUFFER_BINDING} is equivalent to \texttt{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 6.23.

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 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 or three-dimensional texture, which is treated as a two-dimensional image.

Additionally, an entire level of a three-dimensional texture, cube map texture, 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).
4.4. FRAMEBUFFER OBJECTS

Renderbuffer Objects

A renderbuffer is a data storage object containing a single image of a renderable internal format. GL provides the methods described below to allocate and delete a renderbuffer’s image, and to attach a renderbuffer’s image to a framebuffer object.

The name space for renderbuffer objects is the unsigned integers, with zero reserved for 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 the state values listed in table 6.25. 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`.

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
4.4. FRAMEBUFFER OBJECTS

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 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, just as if they were unused.

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, or if samples is greater than the value of MAX_SAMPLES, then the error INVALID_VALUE is generated. If internalformat is a signed or unsigned integer format and samples is greater than the value of MAX_INTEGER_SAMPLES, then the error INVALID_OPERATION is generated (see “Required Renderbuffer Formats” below). 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 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
4.4. FRAMEBUFFER OBJECTS

Table 4.10: Correspondence of sized internal formats to base internal formats for formats that can be used only with renderbuffers.

<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>

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 3.8.1, 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.

OpenGL 3.2 (Core Profile) - July 24, 2009
4.4. FRAMEBUFFER OBJECTS

Attaching Renderbuffer Images to a Framebuffer

A renderbuffer can be attached as one of the logical buffers of the 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 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 6.23. 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 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 6.23.

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.

If a renderbuffer object is deleted while its image is attached to one or more attachment points in the currently bound framebuffer, then it is as if `FramebufferRenderbuffer` had been called, with a `renderbuffer` of 0, for each attachment point to which this image was attached in the currently bound framebuffer. In other words, this renderbuffer image is first detached from all attachment points in
the currently bound framebuffer. Note that the renderbuffer image is specifically not detached from any non-bound framebuffers. Detaching the image from any non-bound framebuffers is the responsibility of the application.

<table>
<thead>
<tr>
<th>Name of attachment</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLOR_ATTACHMENT&lt;i&gt; (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 \( \text{MAX}_\text{COLOR}_\text{ATTACHMENTS} - 1 \).

Attaching Texture Images to a Framebuffer

GL supports copying the rendered contents of the framebuffer into the images of a texture object through the use of the routines CopyTexImage* and CopyTexImageSubImage*. Additionally, 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 *level* is not a supported texture level number for textures of the type corresponding to *target*. An INVALID_OPERATION error is generated if *texture* is the name of a buffer texture.

If *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 the currently bound framebuffer object by calling one of the following routines, depending on the type of the texture:

```c
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 );
```

In all three routines, `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 not zero, then `texture` must either name an existing texture object with a target of `textarget`, or `texture` must name an existing cube map texture and `textarget` must be one of `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`, or `TEXTURE_CUBE_MAP_NEGATIVE_Z`. Otherwise, an `INVALID_OPERATION` error is generated.

`level` specifies the mipmap level of the texture image to be attached to the framebuffer.

If `textarget` is `TEXTURE_RECTANGLE`, `TEXTURE_2D_MULTISAMPLE`, or `TEXTURE_2D_MULTISAMPLE_ARRAY`, then `level` must be zero. If `textarget` is `TEXTURE_3D`, then `level` must be greater than or equal to 0 and less than or equal to `log2` of the value of `MAX_3D_TEXTURE_SIZE`. If `textarget` is one of `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`, or `TEXTURE_CUBE_MAP_NEGATIVE_Z`, then `level` must be greater than or equal to zero and less than or equal to `log2` of the value of `MAX_CUBE_MAP_TEXTURE_SIZE`. For all other values of `textarget`, `level` must be greater than or equal to zero and no larger than `log2` of the value of `MAX_TEXTURE_SIZE`. Otherwise, an `INVALID_VALUE` error is generated.

`layer` specifies the layer of a 2-dimensional image within a 3-dimensional texture. An `INVALID_VALUE` error is generated if `layer` is larger than the value of `MAX_3D_TEXTURE_SIZE-1`.  

OpenGL 3.2 (Core Profile) - July 24, 2009
For **FramebufferTexture1D**, if `texture` is not zero, then `textarget` must be `TEXTURE_1D`.

For **FramebufferTexture2D**, if `texture` is not zero, then `textarget` 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 `textarget` must be `TEXTURE_3D`.

The command

```c
voidFramebufferTextureLayer( 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 texture, or two-dimensional multisample array texture. `layer` is an integer indicating the layer number, and is treated identically to the `layer` parameter in **FramebufferTexture3D**. 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, or one-or two-dimensional array texture. Unlike **FramebufferTexture3D**, no `textarget` 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 6.23.

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 currently bound to `target`. State values of the specified attachment point are set as follows:
4.4. FRAMEBUFFER OBJECTS

- 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 `TEXTURE_CUBE_MAP_POSITIVE_X`.
- 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 6.23. 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.

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 the currently bound framebuffer, 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 the currently bound framebuffer. In other words, this texture image is first detached from all attachment points in the currently bound framebuffer. Note that the texture image is specifically **not** detached from any other framebuffer objects. Detaching the texture image from any other 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 3.8.9) and texture copying feedback loops (see section 3.8.2) 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 3.8.9. 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 framebuffer 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 3.8.9.

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 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 $T$. 

 OpenGL 3.2 (Core Profile) - July 24, 2009  

OpenGL 3.2 (Core Profile) - July 24, 2009
4.4. FRAMEBUFFER OBJECTS

- the value of \texttt{TEXTURE_MIN_FILTER} for texture object \( T \) is one of \texttt{NEAREST_MIPMAP_NEAREST}, \texttt{NEAREST_MIPMAP_LINEAR}, \texttt{LINEAR_MIPMAP_NEAREST}, or \texttt{LINEAR_MIPMAP_LINEAR}, and the value of \texttt{FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL} for attachment point \( A \) is within the range specified by the current values of \texttt{TEXTURE_BASE_LEVEL} to \( q \), inclusive, for the texture object \( T \). (\( q \) is defined in the Mipmapping discussion of section 3.8.9).

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 \texttt{TEXTURE_BASE_LEVEL} and \texttt{TEXTURE_MAX_LEVEL} exclude any levels containing image(s) attached to the currently bound framebuffer, 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 read framebuffer while the same texture image is the destination of a \texttt{CopyTexImage*} operation, as described under “Texture Copying Feedback Loops” in section 3.8.2. 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 \texttt{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 at attachment point \( A \)
- the selected read buffer is attachment point \( A \)
- \( T \) is bound to the texture target of a \texttt{CopyTexImage*} operation
- the \texttt{level} argument of the copying operation selects the same image that is attached to \( A \)
4.4. FRAMEBUFFER OBJECTS

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.

- 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:
4.4. FRAMEBUFFER OBJECTS

- \textit{image} is a component of an existing object with the name specified by the value of \texttt{FRAMEBUFFER\_ATTACHMENT\_OBJECT\_NAME}, and of the type specified by the value of \texttt{FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPE}.

- The width and height of \textit{image} are non-zero.

- If the value of \texttt{FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPE} is \texttt{TEXTURE} and the value of \texttt{FRAMEBUFFER\_ATTACHMENT\_OBJECT\_NAME} names a three-dimensional texture, then the value of \texttt{FRAMEBUFFER\_ATTACHMENT\_TEXTURE\_LAYER} must be smaller than the depth of the texture.

- If the value of \texttt{FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPE} is \texttt{TEXTURE} and the value of \texttt{FRAMEBUFFER\_ATTACHMENT\_OBJECT\_NAME} names a one-or two-dimensional array texture, then the value of \texttt{FRAMEBUFFER\_ATTACHMENT\_TEXTURE\_LAYER} must be smaller than the number of layers in the texture.

- If \textit{attachment} is \texttt{COLOR\_ATTACHMENT\_i}, then \textit{image} must have a color-renderable internal format.

- If \textit{attachment} is \texttt{DEPTH\_ATTACHMENT}, then \textit{image} must have a depth-renderable internal format.

- If \textit{attachment} is \texttt{STENCIL\_ATTACHMENT}, then \textit{image} must have a stencil-renderable internal format.

\textbf{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 \textit{target} is said to be \textit{framebuffer complete} if all the following conditions are true:

- \textit{target} is the default framebuffer, and the default framebuffer exists.

\{ \texttt{FRAMEBUFFER\_UNDEFINED} \}

- All framebuffer attachment points are \textit{framebuffer attachment complete}.

\{ \texttt{FRAMEBUFFER\_INCOMPLETE\_ATTACHMENT} \}
There is at least one image attached to the framebuffer.

\{ FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT \}

The value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE must not be NONE for any color attachment point(s) named by DRAW_BUFFER_i.

\{ FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER \}

If READ_BUFFER is not NONE, then the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE must not be NONE for the color attachment point named by READ_BUFFER.

\{ FRAMEBUFFER_INCOMPLETE_READ_BUFFER \}

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.

\{ 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 framebuffer object that is bound to the framebuffer.
- Changing the read buffer or one of the draw buffers.
- 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`.

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 \texttt{target} can be queried by calling

\begin{verbatim}
enum CheckFramebufferStatus( enum target );
\end{verbatim}

\texttt{target} must be \texttt{DRAW_FRAMEBUFFER}, \texttt{READ_FRAMEBUFFER}, or \texttt{FRAMEBUFFER}. \texttt{FRAMEBUFFER} is equivalent to \texttt{DRAW_FRAMEBUFFER}. If \texttt{CheckFramebufferStatus} generates an error, zero is returned.

Otherwise, a value is returned that identifies whether or not the framebuffer bound to \texttt{target} is complete, and if not complete the value identifies one of the rules of framebuffer completeness that is violated. If the framebuffer is complete, then \texttt{FRAMEBUFFER_COMPLETE} is returned.

The values of \texttt{SAMPLE_BUFFERS} and \texttt{SAMPLES} are derived from the attachments of the currently bound framebuffer object. If the current \texttt{DRAW_FRAMEBUFFER_BINDING} is not framebuffer complete, then both \texttt{SAMPLE_BUFFERS} and \texttt{SAMPLES} are undefined. Otherwise, \texttt{SAMPLES} is equal to the value of \texttt{RENDERBUFFER_SAMPLES} for the attached images (which all must have the same value for \texttt{RENDERBUFFER_SAMPLES}). Further, \texttt{SAMPLE_BUFFERS} is one if \texttt{SAMPLES} is non-zero. Otherwise, \texttt{SAMPLE_BUFFERS} is zero.

\textbf{Required Framebuffer Formats}

Implementations must support framebuffer objects with up to \texttt{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 3.8.1 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.

There must be at least one default framebuffer format allowing creation of a default framebuffer supporting front-buffered rendering.

\textbf{Effects of Framebuffer Completeness on Framebuffer Operations}

Attempting to render to or read from a framebuffer which is not framebuffer complete will generate an \texttt{INVALID_FRAMEBUFFER_OPERATION} error. This means that rendering commands such as \texttt{DrawArrays} or one of the other drawing commands defined in section 2.8.2, as well as commands that read the framebuffer such as \texttt{ReadPixels}, \texttt{CopyTexImage}, and \texttt{CopyTexSubImage}, will generate the error \texttt{INVALID_FRAMEBUFFER_OPERATION} if called while the framebuffer is not complete.
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 \texttt{RASTERIZER\_DISCARD} is enabled.

### 4.4.5 Effects of Framebuffer State on Framebuffer Dependent Values

The values of the state variables listed in table 6.46 may change when a change is made to \texttt{DRAW\_FRAMEBUFFER\_BINDING}, to the state of the currently bound framebuffer object, or to an image attached to the currently bound framebuffer object.

When \texttt{DRAW\_FRAMEBUFFER\_BINDING} is zero, the values of the state variables listed in table 6.46 are implementation defined.

When \texttt{DRAW\_FRAMEBUFFER\_BINDING} is non-zero, if the currently bound framebuffer object is not framebuffer complete, then the values of the state variables listed in table 6.46 are undefined.

When \texttt{DRAW\_FRAMEBUFFER\_BINDING} is non-zero and the currently bound framebuffer object is framebuffer complete, then the values of the state variables listed in table 6.46 are completely determined by \texttt{DRAW\_FRAMEBUFFER\_BINDING}, the state of the currently bound framebuffer object, and the state of the images attached to the currently bound framebuffer object. The actual sizes of the color, depth, or stencil bit planes can be obtained by querying an attachment point using \texttt{GetFramebufferAttachmentParameteriv}, or querying the object attached to that point. If the value of \texttt{FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPE} at a particular attachment point is \texttt{RENDERBUFFER}, the sizes may be determined by calling \texttt{GetRenderbufferParameteriv} as described in section 6.1.3. If the value of \texttt{FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPE} at a particular attachment point is \texttt{TEXTURE}, the sizes may be determined by calling \texttt{GetTexParameter}, as described in section 6.1.3.

### 4.4.6 Mapping between Pixel and Element in Attached Image

When \texttt{DRAW\_FRAMEBUFFER\_BINDING} is non-zero, an operation that writes to the framebuffer modifies the image attached to the selected logical buffer, and an operation 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 3.6 as follows:

\[
i = (x_w - b) \\
j = (y_w - b) \\
k = (layer - b)
\]

where \(b\) is the texture image’s border width and \(layer\) is the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER for the selected logical buffer. For a two-dimensional texture, \(k\) and \(layer\) are irrelevant; for a one-dimensional texture, \(j\), \(k\), and \(layer\) are irrelevant.

\((x_w, y_w)\) corresponds to a border texel if \(x_w, y_w,\) or \(layer\) is less than the border width, or if \(x_w, y_w,\) or \(layer\) 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 or is used as the source of a logical operation or blending while the read framebuffer binding is non-zero, the components of the framebuffer-attachable image that is attached to the logical buffer selected by READ_BUFFER are first converted to R, G, B, and A values according to table 3.20 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 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.

<table>
<thead>
<tr>
<th>Layer Number</th>
<th>Cube Map Face</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>TEXTURE_CUBE_MAP_POSITIVE_X</code></td>
</tr>
<tr>
<td>1</td>
<td><code>TEXTURE_CUBE_MAP_NEGATIVE_X</code></td>
</tr>
<tr>
<td>2</td>
<td><code>TEXTURE_CUBE_MAP_POSITIVE_Y</code></td>
</tr>
<tr>
<td>3</td>
<td><code>TEXTURE_CUBE_MAP_NEGATIVE_Y</code></td>
</tr>
<tr>
<td>4</td>
<td><code>TEXTURE_CUBE_MAP_POSITIVE_Z</code></td>
</tr>
<tr>
<td>5</td>
<td><code>TEXTURE_CUBE_MAP_NEGATIVE_Z</code></td>
</tr>
</tbody>
</table>

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.
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 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.

5.2 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: \textit{signaled} and \textit{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 \textit{Finish}.

The command

\begin{verbatim}
sync FenceSync(enum condition, bitfield flags);
\end{verbatim}

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 \textit{condition} of the sync object is satisfied by the fence command, the sync object is signaled by the GL, causing any \texttt{ClientWaitSync} or \texttt{WaitSync} commands (see below) blocking on \texttt{sync} to unblock. No other state is affected by \texttt{FenceSync} or by execution of the associated fence command.

\textit{condition} must be \texttt{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 after the fence command completes.

\textit{flags} must be 0\footnote{\textit{flags} is a placeholder for anticipated future extensions of fence sync object capabilities.}.

Each sync object contains a number of \textit{properties} which determine the state of the object and the behavior of any commands associated with it. Each property has a \textit{property name} and \textit{property value}. The initial property values for a sync object created by \texttt{FenceSync} are shown in table 5.1.

Properties of a sync object may be queried with \texttt{GetSynciv} (see section 6.1.7). The \texttt{SYNC_STATUS} property will be changed to \texttt{SIGNALED} when \textit{condition} is satisfied.

If \texttt{FenceSync} fails to create a sync object, zero will be returned and a GL error will be generated as described. An \texttt{INVALID_ENUM} error is generated if \textit{condition} is not \texttt{SYNC_GPU_COMMANDS_COMPLETE}.
5.2. 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.

A sync object can be deleted by passing its name to the command

```c
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.

5.2.1 Waiting for Sync Objects

The command

```c
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 5.2.2.

`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 value `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.
5.2. SYNC OBJECTS AND FENCES

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{\texttt{flags} and \texttt{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.}. 

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.

5.2.2 Signalling

A fence sync object enters the signaled state only once the corresponding fence command has completed and signaled the sync object.

If the sync object being blocked upon will not be signaled 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 signaled 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 signaled.
5.3. **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.

<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 dFdx, dFdy and fwidth</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.

OpenGL 3.2 (Core Profile) - July 24, 2009
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 four commands for obtaining simple state variables:

```c
void GetBooleanv( enum value, boolean *data );
void GetIntegerv( enum value, int *data );
void GetInteger64v( enum value, int64 *data );
void GetFloatv( enum value, float *data );
void GetDoublev( enum value, double *data );
```

The commands obtain boolean, integer, 64-bit integer, floating-point, or double-precision state variables. value 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 );
```
6.1. QUERYING GL STATE

```
void GetInteger64i_v( enum target, uint index, int64 *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,

```
boolean IsEnabled( enum value );
```

can be used to determine if value is currently enabled (as with Enable) or disabled, and

```
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 GetBooleanv 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 4.9; 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.

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 $f$.  

OpenGL 3.2 (Core Profile) - July 24, 2009
6.1. QUERYING GL STATE

If fragment color clamping is enabled, querying of the texture border color, blend color, and RGBA clear color will clamp the corresponding state values to \([0, 1]\) before returning them. This behavior provides compatibility with previous versions of the GL that clamped these values when specified.

Most texture state variables are qualified by the value of `ACTIVE_TEXTURE` to determine which server texture state vector is queried. Table 6.13 indicates those state variables which are qualified by `ACTIVE_TEXTURE` during state queries. All other texture state queries will result in an `INVALID_OPERATION` error if the value of `ACTIVE_TEXTURE` is greater than or equal to `MAX_COMBINED_TEXTURE_IMAGE_UNITS`.

Vertex array state variables are qualified by the value of `VERTEX_ARRAY_BINDING` to determine which vertex array object is queried. Tables 6.4 and 6.5 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 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`, or `TEXTURE_CUBE_MAP`, indicating the currently bound one-, two-, three-dimensional, one- or two-dimensional array, rectangular, or cube map 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);
```
6.1. QUERYING GL STATE

places information about texture image parameter \textit{value} for level-of-detail \textit{lod} of the specified \textit{target} into \textit{data}. \textit{value} must be one of the symbolic values in table 6.16.

\textit{target} may be one of \texttt{TEXTURE_1D}, \texttt{TEXTURE_2D}, \texttt{TEXTURE_3D}, \texttt{TEXTURE_1D_ARRAY}, \texttt{TEXTURE_2D_ARRAY}, \texttt{TEXTURE_RECTANGLE}, \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_2D_MULTISAMPLE}, \texttt{TEXTURE_2D_MULTISAMPLE_ARRAY}, \texttt{PROXY_TEXTURE_1D}, \texttt{PROXY_TEXTURE_2D}, \texttt{PROXY_TEXTURE_3D}, \texttt{PROXY_TEXTURE_1D_ARRAY}, \texttt{PROXY_TEXTURE_2D_ARRAY}, \texttt{PROXY_TEXTURE_RECTANGLE}, \texttt{PROXY_TEXTURE_CUBE_MAP}, \texttt{PROXY_TEXTURE_2D_MULTISAMPLE}, or \texttt{PROXY_TEXTURE_2D_MULTISAMPLE_ARRAY}, indicating the one-, two-, or three-dimensional texture, one-or two-dimensional 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, rectangular, cube map, two-dimensional multisample, or two-dimensional multisample array proxy state vector.

\textit{target} may also be \texttt{TEXTURE_BUFFER}, indicating the texture buffer. In the case \textit{lod} must be zero or an \texttt{INVALID_VALUE} error is generated.

Note that \texttt{TEXTURE_CUBE_MAP} is not a valid \textit{target} parameter for \texttt{GetTexLevelParameter}, because it does not specify a particular cube map face.

\textit{lod} determines which level-of-detail’s state is returned. If \textit{lod} is less than zero or larger than the maximum allowable level-of-detail, then an \texttt{INVALID_VALUE} error is generated.

For texture images with uncompressed internal formats, queries of \textit{value} \texttt{TEXTURE_RED_TYPE}, \texttt{TEXTURE_GREEN_TYPE}, \texttt{TEXTURE_BLUE_TYPE}, \texttt{TEXTURE_ALPHA_TYPE}, and \texttt{TEXTURE_DEPTH_TYPE} return the data type used to store the component. Types \texttt{NONE}, \texttt{SIGNED_NORMALIZED}, \texttt{UNSIGNED_NORMALIZED}, \texttt{FLOAT}, \texttt{INT}, and \texttt{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 \textit{value} \texttt{TEXTURE_RED_SIZE}, \texttt{TEXTURE_GREEN_SIZE}, \texttt{TEXTURE_BLUE_SIZE}, \texttt{TEXTURE_ALPHA_SIZE}, \texttt{TEXTURE_DEPTH_SIZE}, \texttt{TEXTURE_STENCIL_SIZE}, and \texttt{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 a compressed internal format, 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 ques-
6.1. QUERYING GL STATE

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 bytes) 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_WIDTH, TEXTURE_HEIGHT, TEXTURE_DEPTH, and TEXTURE_BORDER return the width, height, depth, and border as specified when the image array was created. The internal format of the image array is queried as TEXTURE_INTERNAL_FORMAT.

6.1.4 Texture Queries

The command

```c
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, and TEXTURE_RECTANGLE indicate a one-, two-, or three-dimensional, one- or two-dimensional 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.

- format is DEPTH_COMPONENT and the base internal format is not DEPTH_COMPONENT or DEPTH_STENCIL.
6.1. QUERYING GL STATE

- **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 textures are treated as two- 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 and two-dimensional 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 3.8.1.

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 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
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.

```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. All pixel storage modes are ignored when returning a compressed texture image.

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 `INVALID_OPERATION` error results.

The command

```c
boolean IsTexture( uint texture );
```

returns `TRUE` if `texture` is the name of a texture object. If `texture` is zero, or is a non-zero value that is not the name of a texture object, or if an error condition occurs, `IsTexture` returns `FALSE`. A name returned by `GenTextures`, but not yet bound, is not the name of a texture object.
6.1.5 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 SHADING_LANGUAGE_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 `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 **GetIntegerv** 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 );
```

`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 `index`th supported extension should be returned. `index`
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.

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.

6.1.6 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 a query target can be queried with the command

```java
void GetQueryiv(enum target, enum pname, int *params);
```

target identifies the query target, and must be one of SAMPLES_PASSED for occlusion queries or PRIMITIVES_GENERATED and TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN for primitive queries.

If pname is CURRENT_QUERY, the name of the currently active query for target, or zero if no query is active, will be placed in params.

If pname is QUERY_COUNTER_BITS, 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 (SAMPLES_PASSED), if the number of bits is non-zero, the minimum number of bits allowed is a function of the implementation’s maximum viewport dimensions (MAX_VIEWPORT_DIMS). The counter must be able to
6.1. QUERYING GL STATE

represent at least two overdrawing for every pixel in the viewport. The formula to compute the allowable minimum value (where \( n \) is the minimum number of bits) is

\[
n = \min\{32, \lceil \log_2(\text{maxViewportWidth} \times \text{maxViewportHeight} \times 2) \rceil \}.
\]

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);
```

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.

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 \( TRUE \).

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[u]iv`, 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 \( id \).

6.1.7 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);
```

OpenGL 3.2 (Core Profile) - July 24, 2009
6.1. QUERYING GL STATE

The value or values being queried are returned in the parameters \textit{length} and \textit{values}.

On success, \texttt{GetSynciv} replaces up to \texttt{bufSize} integers in \textit{values} with the corresponding property values of the object being queried. The actual number of integers replaced is returned in \*\textit{length}. If \textit{length} is \texttt{NULL}, no length is returned.

If \textit{pname} is \texttt{OBJECT_TYPE}, a single value representing the specific type of the sync object is placed in \textit{values}. The only type supported is \texttt{SYNC_FENCE}.

If \textit{pname} is \texttt{SYNC_STATUS}, a single value representing the status of the sync object (\texttt{SIGNALED} or \texttt{UNSIGNALED}) is placed in \textit{values}.

If \textit{pname} is \texttt{SYNC_CONDITION}, a single value representing the condition of the sync object is placed in \textit{values}. The only condition supported is \texttt{SYNC_GPU_COMMANDS_COMPLETE}.

If \textit{pname} is \texttt{SYNC_FLAGS}, a single value representing the flags with which the sync object was created is placed in \textit{values}. No flags are currently supported.

If \textit{sync} is not the name of a sync object, an \texttt{INVALID_VALUE} error is generated.

If \textit{pname} is not one of the values described above, an \texttt{INVALID_ENUM} error is generated. If an error occurs, nothing will be written to \textit{values} or \textit{length}.

The command

\begin{verbatim}
boolean IsSync( sync sync );
\end{verbatim}

returns \texttt{TRUE} if \textit{sync} is the name of a sync object. If \textit{sync} is not the name of a sync object, or if an error condition occurs, \texttt{IsSync} returns \texttt{FALSE} (note that zero is not the name of a sync object).

Sync object names immediately become invalid after calling \texttt{DeleteSync}, as discussed in sections 5.2 and D.2, but the underlying sync object will not be deleted until it is no longer associated with any fence command and no longer blocking any \*\texttt{WaitSync} command.

6.1.8 Buffer Object Queries

The command

\begin{verbatim}
boolean IsBuffer( uint buffer );
\end{verbatim}

returns \texttt{TRUE} if \textit{buffer} is the name of an buffer object. If \textit{buffer} is zero, or if \textit{buffer} is a non-zero value that is not the name of an buffer object, \texttt{IsBuffer} returns \texttt{FALSE}.

The commands

\begin{verbatim}
void GetBufferParameteriv( enum target, enum pname, int *data );
\end{verbatim}
6.1. QUERYING GL STATE

```c
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 2.6, and pname must be one of the buffer object parameters in table 2.7, 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 2.6. 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 2.6 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.

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 GetIntegeriv with param set to UNIFORM_BUFFER_BINDING. index must be in the range zero to the value of MAX_UNIFORM_BUFFER_BINDINGS - 1. 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.

To query the starting offset or size of the range of each buffer object binding used for uniform buffers, call GetInteger64iv with param set to UNIFORM_BUFFER_START or UNIFORM_BUFFER_SIZE respectively. index must be in the range zero to the value of MAX_UNIFORM_BUFFER_BINDINGS - 1. If the parameter (starting offset or size) was not specified when the buffer object was bound, zero is returned. If no buffer object is bound to index, -1 is returned.
6.1. QUERYING GL STATE

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 `GetIntegeriv` with `param` set to `TRANSFORM_FEEDBACK_BUFFER_BINDING`. `index` must be in the range zero to the value of `MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS - 1`. 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`.

To query the starting offset or size of the range of each buffer object binding used for transform feedback, call `GetInteger64iv` with `param` set to `TRANSFORM_FEEDBACK_BUFFER_START` or `TRANSFORM_FEEDBACK_BUFFER_SIZE` respectively. `index` must be in the range 0 to the value of `MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS - 1`. If the parameter (starting offset or size) was not specified when the buffer object was bound, zero is returned. If no buffer object is bound to `index`, -1 is returned.

6.1.9 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.

6.1.10 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 `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

```c
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
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, GEOMETRY_SHADER, or FRAGMENT_SHADER is returned if shader is a vertex, 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

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

void GetProgramiv(uint program, enum pname, int *params);

returns properties of the program object named program in params. The parameter value to return is specified by pname.

If pname is DELETE_STATUS, TRUE is returned if the program has been flagged for deletion, and FALSE is returned otherwise. If pname is LINK_STATUS, TRUE is returned if the program was last compiled successfully, and FALSE is returned otherwise. If pname is VALIDATE_STATUS, TRUE is returned if the last call to ValidateProgram with program was successful, 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 ATTACHED_SHADERs, the number of objects attached is returned. If pname is ACTIVE_ATTRIBUTES, the number of active attributes in program is returned. If no active attributes exist, zero is returned. If pname is ACTIVE_ATTRIBUTE_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 pname is ACTIVE_UNIFORMs, the number of active uniforms is returned. If no active uniforms exist, zero is returned. If pname is 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 *pname* is `TRANSFORM_FEEDBACK_BUFFER_MODE`, the buffer mode used when transform feedback is active is returned. It can be one of `SEPARATE_ATTRIBS` or `INTERLEAVED_ATTRIBS`. If *pname* is `TRANSFORM_FEEDBACK_VARYINGS`, the number of varying variables to capture in transform feedback mode for the program is returned. If *pname* is `TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH`, the length of the longest varying name specified to be used for transform feedback, including a null terminator, is returned. If no varyings are used for transform feedback, zero is returned. If *pname* is `ACTIVE_UNIFORM_BLOCKS`, the number of uniform blocks for program containing active uniforms is returned. If *pname* is `ACTIVE_UNIFORM_BLOCK_MAX_NAME_LENGTH`, the length of the longest active uniform block name, including the null terminator, is returned. If *pname* is `GEOMETRY_VERTICES_OUT`, the maximum number of vertices the geometry shader will output is returned. If *pname* is `GEOMETRY_INPUT_TYPE`, the geometry shader input type, which must be one of `POINTS`, `LINES`, `LINES_ADJACENCY`, `TRIANGLES` or `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 `GEOMETRY_VERTICES_OUT`, `GEOMETRY_INPUT_TYPE`, or `GEOMETRY_OUTPUT_TYPE` 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.

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 or last link or validation attempt on a program 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 );
```

OpenGL 3.2 (Core Profile) - July 24, 2009
These commands return the info log string 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 can be queried with GetShaderiv or GetProgramiv with INFO_LOG_LENGTH. If shader is a shader object, the returned info log will either be an empty string or it will contain information about the last compilation attempt for that object. If program is a program object, the returned info log will either be an empty string or it will contain information about the last link attempt or 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, 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 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 );
```
obtain the vertex attribute state named by \textit{pname} for the generic vertex attribute numbered \textit{index} and places the information in the array \textit{params}. \textit{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, or CURRENT_VERTEX_ATTRIB. Note that all the queries except \texttt{CURRENT_VERTEX_ATTRIB} return values stored in the currently bound vertex array object (the value of VERTEX_ARRAY_BINDING). If the zero object is bound, these values are client state. The error INVALID_VALUE is generated if \textit{index} is greater than or equal to \texttt{MAX_VERTEX_ATTRIBS}.

All but \texttt{CURRENT_VERTEX_ATTRIB} return information about generic vertex attribute arrays. The enable state of a generic vertex attribute array is set by the command \texttt{EnableVertexAttribArray} and cleared by \texttt{DisableVertexAttribArray}. The size, stride, type, normalized flag, and unconverted integer flag are set by the commands \texttt{VertexAttribPointer} and \texttt{VertexAttribIPointer}. The normalized flag is always set to \texttt{FALSE} by \texttt{VertexAttribIPointer}. The unconverted integer flag is always set to \texttt{FALSE} by \texttt{VertexAttribPointer} and \texttt{TRUE} by \texttt{VertexAttribIPointer}.

The query \texttt{CURRENT_VERTEX_ATTRIB} returns the current value for the generic attribute \textit{index}. \texttt{GetVertexAttribdv} and \texttt{GetVertexAttribfv} read and return the current attribute values as floating-point values; \texttt{GetVertexAttribiv} reads them as floating-point values and converts them to integer values; \texttt{GetVertexAttribIiv} reads and returns them as integers; \texttt{GetVertexAttribIuiv} reads and returns them as unsigned integers. 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

\begin{verbatim}
void GetVertexAttribPointerv( uint index, enum pname, 
    void **pointer ) ;
\end{verbatim}

obtains the pointer named \textit{pname} for the vertex attribute numbered \textit{index} and places the information in the array \textit{pointer}. \textit{pname} must be VERTEX_ATTRIB_ARRAY_POINTER. The value returned is queried from the currently bound vertex array object. If the zero object is bound, the value is queried from client state. An INVALID_VALUE error is generated if \textit{index} is greater than or equal to the value of \texttt{MAX_VERTEX_ATTRIBS}.

The commands

\begin{verbatim}
void GetUniformfv( uint program, int location, 
    float *params ) ;
\end{verbatim}
6.1. QUERYING GL STATE

```c
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.

### 6.1.11 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` return `FALSE`.

The command

```c
void GetFramebufferAttachmentParameteriv(uint 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 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 3.8.16.

If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is `RENDERBUFFER`, then
6.1. QUERYING GL STATE

- 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.

6.1.12 Renderbuffer Object Queries

The command

```c
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} return \texttt{FALSE}.

The command
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 6.25. 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.

In the tables that follow, a type is indicated 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.

### 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 GetBooleanv, GetIntegerv, GetFloatv, or 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 IsEnabled. However, state variables for which IsEnabled is listed as the query command can also be obtained using GetBooleanv, GetIntegerv, GetFloatv, and 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-
### 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, Z_{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

Point state is returned as integer values or integer state is returned as floating-point values it is converted in the fashion described in section 6.1.2.
<table>
<thead>
<tr>
<th>Command</th>
<th>Get</th>
<th>Command</th>
<th>Get</th>
<th>Command</th>
<th>Get</th>
<th>Command</th>
<th>Get</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex attrib array enable</td>
<td>FALSE</td>
<td>Vertex attrib array size</td>
<td>4</td>
<td>Vertex attrib array stride</td>
<td>0</td>
<td>Vertex attrib array type</td>
<td>FLOAT</td>
</tr>
<tr>
<td>Vertex attrib array has unconverted integers</td>
<td>FALSE</td>
<td>Vertex attrib array pointer</td>
<td>NULL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4. Vertex Array Object State (cont.)
### 6.2. State Tables

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Type</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetIntegerv</td>
<td>ELEMENT</td>
<td>0</td>
<td>Element array binding</td>
<td>2.9.7</td>
</tr>
<tr>
<td>GetVertexAttrib</td>
<td>BUFFER</td>
<td>0</td>
<td>Attribute array binding</td>
<td>2.9</td>
</tr>
</tbody>
</table>

Table 6.5. Vertex Array 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>ARRAY_BUFFER_BINDING</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Current buffer binding</td>
<td>2.9</td>
</tr>
<tr>
<td>VERTEX_ARRAY_BINDING</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Current vertex array object binding</td>
<td>2.10</td>
</tr>
<tr>
<td>PRIMITIVE_RESTART</td>
<td>$B$</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Primitive restart enable</td>
<td>2.8</td>
</tr>
<tr>
<td>PRIMITIVE_RESTART_INDEX</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Primitive restart index</td>
<td>2.8</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>-</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>$n \times Z^+$</td>
<td>GetBufferParameteriv</td>
<td>STATIC_DRAW</td>
<td>Buffer usage pattern</td>
<td>2.9</td>
</tr>
<tr>
<td>BUFFER_ACCESS</td>
<td>$n \times Z^+$</td>
<td>GetBufferParameteriv</td>
<td>READ_WRITE</td>
<td>Buffer access flag</td>
<td>2.9</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</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</td>
</tr>
<tr>
<td>BUFFER_MAP_POINTER</td>
<td>$n \times Y$</td>
<td>GetBufferPointerv</td>
<td>NULL</td>
<td>Mapped buffer pointer</td>
<td>2.9</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</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</td>
</tr>
</tbody>
</table>

Table 6.7. Buffer Object State

OpenGL 3.2 (Core Profile) - July 24, 2009
Table 6.8. 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>Viewport</td>
<td>$4 \times Z$</td>
<td>GetIntegerv</td>
<td>see 2.13.1</td>
<td>Viewport origin &amp; extent</td>
<td>2.13.1</td>
</tr>
<tr>
<td>Depth range</td>
<td>$2 \times R^+$</td>
<td>GetFloatv</td>
<td>0.1</td>
<td>Depth range near &amp; far</td>
<td>2.13.1</td>
</tr>
<tr>
<td>Clip distance i</td>
<td>$6 \times B$</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>$i^{th}$ user clipping plane enabled</td>
<td>2.19</td>
</tr>
<tr>
<td>Depth clamp</td>
<td>$B$</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Depth clamping enabled</td>
<td>2.19</td>
</tr>
</tbody>
</table>
### 6.2. STATE TABLES

<table>
<thead>
<tr>
<th>Get value</th>
<th>Get Command</th>
<th>Type</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLAMP, READ, COLOR</td>
<td>GetIntegerv</td>
<td>$Z_3$</td>
<td>FIXED_ONLY</td>
<td>Read color clamping</td>
<td>4.3.1</td>
</tr>
<tr>
<td>PROVOKING, VERTEX</td>
<td>GetIntegerv</td>
<td>$Z_2$</td>
<td>LAST VERTEX_CONVENTION</td>
<td>Provoking vertex convention</td>
<td>2.18</td>
</tr>
</tbody>
</table>
## 6.2. STATE TABLES

<table>
<thead>
<tr>
<th>Command</th>
<th>Get Command</th>
<th>Type</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>POINT SIZE</td>
<td>GetFloatv</td>
<td>R^+</td>
<td>1.0</td>
<td>Point size</td>
<td>3.4</td>
</tr>
<tr>
<td>POINT FAKE</td>
<td>GetFloatv</td>
<td>R</td>
<td>1.0</td>
<td>Threshold for alpha attenuation</td>
<td>3.4</td>
</tr>
<tr>
<td>POINT FAKE</td>
<td>GetIntegerv</td>
<td>Z^+</td>
<td>Upper, left</td>
<td>Origin orientation for point sprites</td>
<td>3.4</td>
</tr>
<tr>
<td>LINE WIDTH</td>
<td>GetIntegerv</td>
<td>B</td>
<td>1.0</td>
<td>Line width</td>
<td>3.5</td>
</tr>
<tr>
<td>LINE ANTI</td>
<td>GetFloatv</td>
<td>B</td>
<td>FA, SE</td>
<td>Line antialiasing on</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Table 6.10. Rasterization
Table 6.11. Rasterization (cont.)
<table>
<thead>
<tr>
<th>Command</th>
<th>Initial Value</th>
<th>Get value</th>
<th>Type</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTISAMPLE</td>
<td>TRUE</td>
<td>B</td>
<td>MULTISAMPLE</td>
<td>Multisample rasterization</td>
<td>3.3.1</td>
</tr>
<tr>
<td>SAMPLE ALPHA TO COVERAGE</td>
<td>FALSE</td>
<td>B</td>
<td>SAMPLE ALPHA TO COVERAGE</td>
<td>Modify coverage from alpha</td>
<td>4.1.3</td>
</tr>
<tr>
<td>SAMPLE ALPHA TO ONE</td>
<td>FALSE</td>
<td>B</td>
<td>SAMPLE ALPHA TO ONE</td>
<td>Set alpha to maximum</td>
<td>4.1.3</td>
</tr>
<tr>
<td>SAMPLE COVERAGE</td>
<td>FALSE</td>
<td>B</td>
<td>SAMPLE COVERAGE</td>
<td>Mask to modify coverage</td>
<td>4.1.3</td>
</tr>
<tr>
<td>SAMPLE COVERAGE INVERT</td>
<td>FALSE</td>
<td>B</td>
<td>SAMPLE COVERAGE INVERT</td>
<td>Invert coverage mask value</td>
<td>4.1.3</td>
</tr>
<tr>
<td>SAMPLE MASK</td>
<td>FALSE</td>
<td>B</td>
<td>SAMPLE MASK</td>
<td>Sample mask enable</td>
<td>4.1.3</td>
</tr>
<tr>
<td>SAMPLE MASK VALUE</td>
<td>1</td>
<td>B</td>
<td>SAMPLE MASK VALUE</td>
<td>Sample mask words</td>
<td>4.1.3</td>
</tr>
</tbody>
</table>

Table 6.12. Multisampling
### Table 6.13. Textures (state per texture unit and binding point)

<table>
<thead>
<tr>
<th>Type</th>
<th>Get Command</th>
<th>Minimum Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXTURE_BINDING_1D</td>
<td>GetInteger</td>
<td>0</td>
<td>Texture object bound to TEXTURE_1D</td>
<td>3.8.14</td>
</tr>
<tr>
<td>TEXTURE_BINDING_2D_ARRAY</td>
<td>GetInteger</td>
<td>0</td>
<td>Texture object bound to TEXTURE_2D_ARRAY</td>
<td>3.8.14</td>
</tr>
<tr>
<td>TEXTURE_BINDING_1D_ARRAY</td>
<td>GetInteger</td>
<td>0</td>
<td>Texture object bound to TEXTURE_1D_ARRAY</td>
<td>3.8.14</td>
</tr>
</tbody>
</table>

Get value:
- \( 4S \times 3 \times Z^+ \)
- \( 4S \times Z^+ \)
- \( 4S \times Z^+ \)
<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>TEXTURE BINDING RECTANGLE</td>
<td>48 × Z⁺</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Texture object bound to TEXTURE_RECTANGLE</td>
<td>3.8.14</td>
</tr>
<tr>
<td>TEXTURE BINDING BUFFER</td>
<td>48 × Z⁺</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Texture object bound to TEXTURE_BUFFER</td>
<td>3.8.14</td>
</tr>
<tr>
<td>TEXTURE BINDING CUBE_MAP</td>
<td>48 × Z⁺</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Texture object bound to TEXTURE_CUBE_MAP</td>
<td>3.8.14</td>
</tr>
<tr>
<td>TEXTURE BINDING 2D MULTISAMPLE</td>
<td>48 × Z⁺</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Texture object bound to TEXTURE_2D_MULTISAMPLE</td>
<td>3.8.13</td>
</tr>
<tr>
<td>TEXTURE BINDING 2D MULTISAMPLE ARRAY</td>
<td>48 × Z⁺</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Texture object bound to TEXTURE_2D_MULTISAMPLE_ARRAY</td>
<td>3.8.13</td>
</tr>
<tr>
<td>TEXTURE 1D</td>
<td>n × I</td>
<td>GetTexImage</td>
<td>see 3.8</td>
<td>xD texture image at l.o.d. i</td>
<td>3.8</td>
</tr>
<tr>
<td>TEXTURE 1D ARRAY</td>
<td>n × I</td>
<td>GetTexImage</td>
<td>see 3.8</td>
<td>1D texture array image at row i</td>
<td>3.8</td>
</tr>
<tr>
<td>TEXTURE 2D ARRAY</td>
<td>n × I</td>
<td>GetTexImage</td>
<td>see 3.8</td>
<td>2D texture array image at slice i</td>
<td>3.8</td>
</tr>
<tr>
<td>TEXTURE RECTANGLE</td>
<td>n × I</td>
<td>GetTexImage</td>
<td>see 3.8</td>
<td>Rectangular texture image at l.o.d. zero</td>
<td>3.8</td>
</tr>
<tr>
<td>TEXTURE CUBE MAP POSITIVE X</td>
<td>n × I</td>
<td>GetTexImage</td>
<td>see 3.8.1</td>
<td>+x face cube map texture image at l.o.d. i</td>
<td>3.8.1</td>
</tr>
<tr>
<td>TEXTURE CUBE MAP NEGATIVE X</td>
<td>n × I</td>
<td>GetTexImage</td>
<td>see 3.8.1</td>
<td>-x face cube map texture image at l.o.d. i</td>
<td>3.8.1</td>
</tr>
<tr>
<td>TEXTURE CUBE MAP POSITIVE Y</td>
<td>n × I</td>
<td>GetTexImage</td>
<td>see 3.8.1</td>
<td>+y face cube map texture image at l.o.d. i</td>
<td>3.8.1</td>
</tr>
<tr>
<td>TEXTURE CUBE MAP NEGATIVE Y</td>
<td>n × I</td>
<td>GetTexImage</td>
<td>see 3.8.1</td>
<td>-y face cube map texture image at l.o.d. i</td>
<td>3.8.1</td>
</tr>
<tr>
<td>TEXTURE CUBE MAP POSITIVE Z</td>
<td>n × I</td>
<td>GetTexImage</td>
<td>see 3.8.1</td>
<td>+z face cube map texture image at l.o.d. i</td>
<td>3.8.1</td>
</tr>
<tr>
<td>TEXTURE CUBE MAP NEGATIVE Z</td>
<td>n × I</td>
<td>GetTexImage</td>
<td>see 3.8.1</td>
<td>-z face cube map texture image at l.o.d. i</td>
<td>3.8.1</td>
</tr>
</tbody>
</table>
### Table 6.15. 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_BORDER_COLOR</td>
<td>$n \times C$</td>
<td>GetTexParameter</td>
<td>0,0,0,0</td>
<td>Border color</td>
<td>3.8</td>
</tr>
<tr>
<td>TEXTURE_MIN_FILTER</td>
<td>$n \times Z_6$</td>
<td>GetTexParameter</td>
<td>see sec. 3.8.13</td>
<td>Minification function</td>
<td>3.8.9</td>
</tr>
<tr>
<td>TEXTURE_MAG_FILTER</td>
<td>$n \times Z_2$</td>
<td>GetTexParameter</td>
<td>LINEAR</td>
<td>Magnification function</td>
<td>3.8.10</td>
</tr>
<tr>
<td>TEXTURE_WRAP_S</td>
<td>$n \times Z_4$</td>
<td>GetTexParameter</td>
<td>see sec. 3.8.13</td>
<td>Texcoord $s$ wrap mode</td>
<td>3.8.9</td>
</tr>
<tr>
<td>TEXTURE_WRAP_T</td>
<td>$n \times Z_4$</td>
<td>GetTexParameter</td>
<td>see sec. 3.8.13</td>
<td>Texcoord $t$ wrap mode (2D, 3D, cube map textures only)</td>
<td>3.8.9</td>
</tr>
<tr>
<td>TEXTURE_WRAP_R</td>
<td>$n \times Z_4$</td>
<td>GetTexParameter</td>
<td>see sec. 3.8.13</td>
<td>Texcoord $r$ wrap mode (3D textures only)</td>
<td>3.8.9</td>
</tr>
<tr>
<td>TEXTURE_MIN_LOD</td>
<td>$n \times R$</td>
<td>GetTexParameterfv</td>
<td>-1000</td>
<td>Minimum level of detail</td>
<td>3.8</td>
</tr>
<tr>
<td>TEXTURE_MAX_LOD</td>
<td>$n \times R$</td>
<td>GetTexParameterfv</td>
<td>1000</td>
<td>Maximum level of detail</td>
<td>3.8</td>
</tr>
<tr>
<td>TEXTURE_BASE_LEVEL</td>
<td>$n \times Z^+$</td>
<td>GetTexParameterfv</td>
<td>0</td>
<td>Base texture array</td>
<td>3.8</td>
</tr>
<tr>
<td>TEXTURE_MAX_LEVEL</td>
<td>$n \times Z^+$</td>
<td>GetTexParameterfv</td>
<td>1000</td>
<td>Max. texture array level</td>
<td>3.8</td>
</tr>
<tr>
<td>TEXTURE_LOD_BIAS</td>
<td>$n \times R$</td>
<td>GetTexParameterfv</td>
<td>0.0</td>
<td>Texture level of detail bias ($bias_{texobj}$)</td>
<td>3.8.9</td>
</tr>
<tr>
<td>TEXTURE_COMPARE_MODE</td>
<td>$n \times Z_2$</td>
<td>GetTexParameteriv</td>
<td>NONE</td>
<td>Comparison mode</td>
<td>3.8.15</td>
</tr>
<tr>
<td>TEXTURE_COMPARE_FUNC</td>
<td>$n \times Z_8$</td>
<td>GetTexParameteriv</td>
<td>LEQUAL</td>
<td>Comparison function</td>
<td>3.8.15</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>TEXTURE_WIDTH</td>
<td>$n \times Z^+$</td>
<td>GetTexImageParameter</td>
<td>0</td>
<td>Specified width</td>
<td>3.8</td>
</tr>
<tr>
<td>TEXTURE_HEIGHT</td>
<td>$n \times Z^+$</td>
<td>GetTexImageParameter</td>
<td>0</td>
<td>Specified height (2D/3D)</td>
<td>3.8</td>
</tr>
<tr>
<td>TEXTURE_DEPTH</td>
<td>$n \times Z^+$</td>
<td>GetTexImageParameter</td>
<td>0</td>
<td>Specified depth (3D)</td>
<td>3.8</td>
</tr>
<tr>
<td>TEXTURE_SAMPLES</td>
<td>$Z^+$</td>
<td>GetTexImageParameter</td>
<td>0</td>
<td>Number of samples per texel</td>
<td>3.8.4</td>
</tr>
<tr>
<td>TEXTURE_FIXED_SAMPLE_LOCATIONS</td>
<td>$B$</td>
<td>GetTexImageParameter</td>
<td>TRUE</td>
<td>Whether the image uses a fixed sample pattern</td>
<td>3.8.4</td>
</tr>
<tr>
<td>TEXTURE_INTERNAL_FORMAT</td>
<td>$n \times Z_{68^*}$</td>
<td>GetTexImageParameter</td>
<td>RGBA or R8</td>
<td>Internal format (see section 3.8.13)</td>
<td>3.8</td>
</tr>
<tr>
<td>TEXTURE_SIZE</td>
<td>$n \times 6 \times Z^+$</td>
<td>GetTexImageParameter</td>
<td>0</td>
<td>Component resolution ($x$ is RED, GREEN, BLUE, ALPHA, DEPTH, or STENCIL)</td>
<td>3.8</td>
</tr>
<tr>
<td>TEXTURE_SHARED_SIZE</td>
<td>$n \times Z^+$</td>
<td>GetTexImageParameter</td>
<td>0</td>
<td>Shared exponent field resolution</td>
<td>3.8</td>
</tr>
<tr>
<td>TEXTURE_TYPE</td>
<td>$n \times Z_5$</td>
<td>GetTexImageParameter</td>
<td>NONE</td>
<td>Component type ($x$ is RED, GREEN, BLUE, ALPHA, or DEPTH)</td>
<td>6.1.3</td>
</tr>
<tr>
<td>TEXTURE_COMPRESSED</td>
<td>$n \times B$</td>
<td>GetTexImageParameter</td>
<td>FALSE</td>
<td>True if image has a compressed internal format</td>
<td>3.8.3</td>
</tr>
<tr>
<td>TEXTURE_COMPRESSED_IMAGE_SIZE</td>
<td>$n \times Z^+$</td>
<td>GetTexImageParameter</td>
<td>0</td>
<td>Size (in ubytes) of compressed image</td>
<td>3.8.3</td>
</tr>
<tr>
<td>TEXTURE_BUFFER_DATA_STORE_BINDING</td>
<td>$n \times Z^+$</td>
<td>GetTexImageParameter</td>
<td>0</td>
<td>Buffer object bound as the data store for the active image unit’s buffer texture</td>
<td>3.8.14</td>
</tr>
</tbody>
</table>
### Table 6.17. Texture Environment and Generation

<table>
<thead>
<tr>
<th>Get value Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVE TEXTURE</td>
<td>GetIntegerv</td>
<td>TEXTURE0</td>
<td>Active texture unit selector</td>
</tr>
</tbody>
</table>

*Sec.: 2.7*
<table>
<thead>
<tr>
<th>Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCISSOR</td>
<td>FALSE</td>
<td>Scissoring enabled</td>
<td>4.1.2</td>
</tr>
<tr>
<td>BOX</td>
<td>4 × Z</td>
<td>Scissor box</td>
<td>4.1.2</td>
</tr>
<tr>
<td>STENCIL</td>
<td>FALSE</td>
<td>Stenciling enabled</td>
<td>4.1.4</td>
</tr>
<tr>
<td>TEST</td>
<td>B</td>
<td>Front stencil function</td>
<td>4.1.4</td>
</tr>
<tr>
<td>B</td>
<td>Z&lt;sub&gt;z&lt;/sub&gt;</td>
<td>Front stencil mask</td>
<td>4.1.4</td>
</tr>
<tr>
<td>Z</td>
<td>KEEP</td>
<td>Front stencil reference value</td>
<td>4.1.4</td>
</tr>
<tr>
<td>Z&lt;sub&gt;z&lt;/sub&gt;</td>
<td>KEEP</td>
<td>Front stencil fail action</td>
<td>4.1.4</td>
</tr>
<tr>
<td>Z&lt;sub&gt;z&lt;/sub&gt;</td>
<td>KEEP</td>
<td>Front stencil depth buffer fail action</td>
<td>4.1.4</td>
</tr>
<tr>
<td>STENCIL</td>
<td>KEEP</td>
<td>Back stencil function</td>
<td>4.1.4</td>
</tr>
<tr>
<td>BACK</td>
<td>KEEP</td>
<td>Back stencil mask</td>
<td>4.1.4</td>
</tr>
<tr>
<td>STENCIL</td>
<td>KEEP</td>
<td>Back stencil reference value</td>
<td>4.1.4</td>
</tr>
<tr>
<td>BACK</td>
<td>KEEP</td>
<td>Back stencil fail action</td>
<td>4.1.4</td>
</tr>
<tr>
<td>STENCIL</td>
<td>KEEP</td>
<td>Back stencil depth buffer fail action</td>
<td>4.1.4</td>
</tr>
<tr>
<td>BACK</td>
<td>KEEP</td>
<td>Back stencil depth buffer pass action</td>
<td>4.1.4</td>
</tr>
</tbody>
</table>

Table 6.18. Pixel Operations
6.2. STATE TABLES

<table>
<thead>
<tr>
<th>Command Type</th>
<th>Get Value</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPTH</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>Depth buffer enabled</td>
<td>4.1.5</td>
</tr>
<tr>
<td>DEPTH_TEST</td>
<td>glIsEnabled</td>
<td>TRUE</td>
<td>Depth test enabled</td>
<td>4.1.5</td>
</tr>
<tr>
<td>BLEND</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>Blending enabled</td>
<td>4.1.5</td>
</tr>
<tr>
<td>BLEND_SRC</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>Blending enabled for source</td>
<td>4.1.5</td>
</tr>
<tr>
<td>BLEND_DST</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>Blending enabled for destination</td>
<td>4.1.5</td>
</tr>
<tr>
<td>BLEND_EQUATION</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>Blending equation enabling</td>
<td>4.1.5</td>
</tr>
<tr>
<td>BLEND_COLOR</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>Color blending enabled</td>
<td>4.1.5</td>
</tr>
<tr>
<td>LOGIC_OP</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>Logic operation enabled</td>
<td>4.1.5</td>
</tr>
<tr>
<td>DITHER</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>Dithering enabled</td>
<td>4.1.5</td>
</tr>
<tr>
<td>FRAMEBUFFER_SRC</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>RGB update and blending enabled</td>
<td>4.1.5</td>
</tr>
<tr>
<td>FRAMEBUFFER_DST</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>RGB blending enabled</td>
<td>4.1.5</td>
</tr>
<tr>
<td>COLOR_ATTACHMENT_BLEND</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>Constant blend color</td>
<td>4.1.5</td>
</tr>
<tr>
<td>OBJECT_BLEND</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>Object blending enabled</td>
<td>4.1.5</td>
</tr>
<tr>
<td>OBJECT_BLEND_RESULT</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>Object blend result enabled</td>
<td>4.1.5</td>
</tr>
<tr>
<td>OBJECT_BLEND_SOURCE</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>Object blend source enabled</td>
<td>4.1.5</td>
</tr>
<tr>
<td>OBJECT_BLEND_RESULT</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>Object blend destination enabled</td>
<td>4.1.5</td>
</tr>
<tr>
<td>OBJECT_BLEND_SOURCE</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>Object blend equation enabled</td>
<td>4.1.5</td>
</tr>
<tr>
<td>OBJECT_BLEND_RESULT</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>Object blend equation enabled</td>
<td>4.1.5</td>
</tr>
<tr>
<td>OBJECT_BLEND_SOURCE</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>Object blend color enabled</td>
<td>4.1.5</td>
</tr>
<tr>
<td>OBJECT_BLEND_RESULT</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>Object blend logic enabled</td>
<td>4.1.5</td>
</tr>
<tr>
<td>OBJECT_BLEND_SOURCE</td>
<td>glIsEnabled</td>
<td>FALSE</td>
<td>Object blend logic enabled</td>
<td>4.1.5</td>
</tr>
</tbody>
</table>

Table 6.19. Pixel Operations (cont.)
### Table 6.20. Framebuffer Control

<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>$1 \times 4 \times B$</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</td>
<td>Color buffer clear value</td>
<td>4.2.3</td>
</tr>
<tr>
<td>DEPTH_CLEAR_VALUE</td>
<td>$R^+$</td>
<td>GetIntegerv</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 6.21. Framebuffer (state per target binding point)

<table>
<thead>
<tr>
<th>Get value</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAW_FRAMEBUFFER BINDING</td>
<td>GetIntegery</td>
<td>0</td>
<td>Framebuffer object bound to DRAW_FRAMEBUFFER</td>
</tr>
<tr>
<td>READ_FRAMEBUFFER BINDING</td>
<td>GetIntegery</td>
<td>0</td>
<td>Framebuffer object bound to READ_FRAMEBUFFER</td>
</tr>
</tbody>
</table>

Type: \(Z^+\)
<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_BUFFER</td>
<td>1 * Z11*</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>Z11*</td>
<td>GetIntegerv</td>
<td>see 4.3.1</td>
<td>Read source buffer</td>
<td>4.3.1</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>FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE</td>
<td>Z</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_ATTACHMENT_OBJECT_NAME</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_ATTACHMENT_TEXTURE_LEVEL</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_ATTACHMENT_TEXTURE_CUBE_MAP</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_ATTACHMENT_TEXTURE_LAYER</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_ATTACHMENT_LAYERED</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_ATTACHMENT_COLOR_ENCODING</td>
<td>$Z_2$</td>
<td>GetFramebufferAttachmentParameteriv</td>
<td>-</td>
<td>Encoding of components in the attached image</td>
<td>6.1.3</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE</td>
<td>$Z_4$</td>
<td>GetFramebufferAttachmentParameteriv</td>
<td>-</td>
<td>Data type of components in the attached image</td>
<td>6.1.3</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_x_SIZE</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>6.1.3</td>
</tr>
</tbody>
</table>
### Table 6.24. Renderbuffer (state per target and binding point)

<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>Z</td>
<td>RENDERBUFFER</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 6.25. Renderbuffer (state per renderbuffer object)

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Type</th>
<th>Initial Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetRenderbufferParameteriv</td>
<td>Z[+0]</td>
<td>Z[+0]</td>
<td>Width of renderbuffer</td>
</tr>
<tr>
<td></td>
<td>RGBA[+0]</td>
<td>0</td>
<td>Height of renderbuffer</td>
</tr>
<tr>
<td></td>
<td>Z[+0]</td>
<td>RGBA[+0]</td>
<td>Internal format of renderbuffer image’s red component</td>
</tr>
<tr>
<td></td>
<td>Z[+0]</td>
<td>Z[+0]</td>
<td>Size in bits of renderbuffer image’s green component</td>
</tr>
<tr>
<td></td>
<td>Z[+0]</td>
<td>RGBA[+0]</td>
<td>Size in bits of renderbuffer image’s blue component</td>
</tr>
<tr>
<td></td>
<td>Z[+0]</td>
<td>RGBA[+0]</td>
<td>Size in bits of renderbuffer image’s alpha component</td>
</tr>
<tr>
<td></td>
<td>Z[+0]</td>
<td>RGBA[+0]</td>
<td>Size in bits of renderbuffer image’s depth component</td>
</tr>
<tr>
<td></td>
<td>Z[+0]</td>
<td>RGBA[+0]</td>
<td>Size in bits of renderbuffer image’s stencil component</td>
</tr>
<tr>
<td></td>
<td>Z[+0]</td>
<td>RGBA[+0]</td>
<td>Number of samples</td>
</tr>
</tbody>
</table>

**Type**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**

**Initial Value**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**
- **Z[+0]**

**Description**
- Width of renderbuffer
- Height of renderbuffer
- Internal format of renderbuffer image’s red component
- Size in bits of renderbuffer image’s green component
- Size in bits of renderbuffer image’s blue component
- Size in bits of renderbuffer image’s alpha component
- Size in bits of renderbuffer image’s depth component
- Size in bits of renderbuffer image’s stencil component
- Number of samples
<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>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>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>PIXEL_UNPACK_BUFFER_BINDING</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Pixel unpack buffer binding</td>
<td>6.1.8</td>
</tr>
</tbody>
</table>
### Shader Object State

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Command</th>
<th>Description</th>
<th>Initial Value</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetShaderiv</td>
<td>GetShaderiv</td>
<td>Type of shader (vertex, geometry, or fragment)</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td></td>
<td>GetShaderiv</td>
<td>Shader flagged for deletion</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td></td>
<td>GetShaderiv</td>
<td>Last compile succeeded</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td></td>
<td>GetShaderInfoLog</td>
<td>Info log for shader objects</td>
<td>empty string</td>
<td>empty string</td>
</tr>
<tr>
<td></td>
<td>GetShaderSource</td>
<td>Source code for a shader</td>
<td>empty string</td>
<td>empty string</td>
</tr>
<tr>
<td></td>
<td>GetShaderSource</td>
<td>Length of source code</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Description</th>
<th>Initial Value</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHADER-TYPE</td>
<td>Zi</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BINT-STATES</td>
<td>Bi</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>COMPILE-STATES</td>
<td>Bi</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>INFO-LENGTH</td>
<td>Zi+</td>
<td>empty string</td>
<td>empty string</td>
<td>empty string</td>
</tr>
<tr>
<td>SOURCE-LENGTH</td>
<td>Zi+</td>
<td>empty string</td>
<td>empty string</td>
<td>empty string</td>
</tr>
</tbody>
</table>

Table 6.27. Shader Object State
### Table 6.28. Program 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>CURRENT_PROGRAM</td>
<td>Z⁺</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Name of current program object</td>
<td>2.11.2</td>
</tr>
<tr>
<td>DELETE_STATUS</td>
<td>B</td>
<td>GetProgramiv</td>
<td>FALSE</td>
<td>Program object deleted</td>
<td>2.11.2</td>
</tr>
<tr>
<td>LINK_STATUS</td>
<td>B</td>
<td>GetProgramiv</td>
<td>FALSE</td>
<td>Last link attempt succeeded</td>
<td>2.11.2</td>
</tr>
<tr>
<td>VALIDATE_STATUS</td>
<td>B</td>
<td>GetProgramiv</td>
<td>FALSE</td>
<td>Last validate attempt succeeded</td>
<td>2.11.2</td>
</tr>
<tr>
<td>ATTACHED_SHADERS</td>
<td>Z⁺</td>
<td>GetProgramiv</td>
<td>0</td>
<td>Number of attached shader objects</td>
<td>6.1.10</td>
</tr>
<tr>
<td></td>
<td>0⁺ × Z⁺</td>
<td>GetAttachedShaders</td>
<td>empty</td>
<td>Shader objects attached</td>
<td>6.1.10</td>
</tr>
<tr>
<td>INFO_LOG_LENGTH</td>
<td>Z⁺</td>
<td>GetProgramiv</td>
<td>0</td>
<td>Length of info log</td>
<td>2.11.4</td>
</tr>
<tr>
<td>ACTIVE_UNIFORMS</td>
<td>Z⁺</td>
<td>GetProgramiv</td>
<td>0</td>
<td>Number of active uniforms</td>
<td>2.11.4</td>
</tr>
<tr>
<td></td>
<td>0⁺ × Z⁺</td>
<td>GetUniformLocation</td>
<td>–</td>
<td>Location of active uniforms</td>
<td>6.1.10</td>
</tr>
<tr>
<td></td>
<td>0⁺ × Z⁺</td>
<td>GetActiveUniform</td>
<td>–</td>
<td>Size of active uniform</td>
<td>2.11.4</td>
</tr>
<tr>
<td></td>
<td>0⁺ × char</td>
<td>GetActiveUniform</td>
<td>empty</td>
<td>Name of active uniform</td>
<td>2.11.4</td>
</tr>
<tr>
<td>ACTIVE_UNIFORM_MAX_LENGTH</td>
<td>Z⁺</td>
<td>GetProgramiv</td>
<td>0</td>
<td>Maximum active uniform name length</td>
<td>6.1.10</td>
</tr>
<tr>
<td></td>
<td>512⁺ × R</td>
<td>GetUniform</td>
<td>0</td>
<td>Uniform value</td>
<td>2.11.4</td>
</tr>
<tr>
<td>ACTIVE_ATTRIBUTES</td>
<td>Z⁺</td>
<td>GetProgramiv</td>
<td>0</td>
<td>Number of active attributes</td>
<td>2.11.3</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>-</td>
<td>(0 \times Z)</td>
<td>GetAttribLocation</td>
<td>-</td>
<td>Location of active generic attribute</td>
<td>2.11.3</td>
</tr>
<tr>
<td>-</td>
<td>(0 \times Z^+)</td>
<td>GetActiveAttrib</td>
<td>-</td>
<td>Size of active attribute</td>
<td>2.11.3</td>
</tr>
<tr>
<td>-</td>
<td>(0 \times Z^+)</td>
<td>GetActiveAttrib</td>
<td>-</td>
<td>Type of active attribute</td>
<td>2.11.3</td>
</tr>
<tr>
<td>-</td>
<td>(0 \times \text{char})</td>
<td>GetActiveAttrib</td>
<td>empty</td>
<td>Name of active attribute</td>
<td>2.11.3</td>
</tr>
<tr>
<td>ACTIVE_ATTRIB_MAX_LENGTH</td>
<td>(Z^+)</td>
<td>GetProgramiv</td>
<td>0</td>
<td>Maximum active attribute name length</td>
<td>6.1.10</td>
</tr>
<tr>
<td>GEOMETRY_VERTICES_OUT</td>
<td>(Z^+)</td>
<td>GetProgramiv</td>
<td>0</td>
<td>Maximum number of output vertices</td>
<td>2.12.4</td>
</tr>
<tr>
<td>GEOMETRY_INPUT_TYPE</td>
<td>(Z_5)</td>
<td>GetProgramiv</td>
<td>TRIANGLES</td>
<td>Primitive input type</td>
<td>2.12.1</td>
</tr>
<tr>
<td>GEOMETRY_OUTPUT_TYPE</td>
<td>(Z_3)</td>
<td>GetProgramiv</td>
<td>TRIANGLE_STRIP</td>
<td>Primitive output type</td>
<td>2.12.2</td>
</tr>
<tr>
<td>TRANSFORM_FEEDBACK_BUFFER_MODE</td>
<td>(Z_2)</td>
<td>GetProgramiv</td>
<td>INTERLEAVED_-ATTRIBS</td>
<td>Transform feedback mode for the program</td>
<td>6.1.10</td>
</tr>
<tr>
<td>TRANSFORM_FEEDBACK_VARYINGS</td>
<td>(Z^+)</td>
<td>GetProgramiv</td>
<td>0</td>
<td>Number of varyings to stream to buffer object(s)</td>
<td>6.1.10</td>
</tr>
<tr>
<td>TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH</td>
<td>(Z^+)</td>
<td>GetProgramiv</td>
<td>0</td>
<td>Maximum transform feedback varying name length</td>
<td>6.1.10</td>
</tr>
<tr>
<td>-</td>
<td>(Z^+)</td>
<td>GetTransformFeedbackVarying</td>
<td>-</td>
<td>Size of each transform feedback varying variable</td>
<td>2.11.6</td>
</tr>
<tr>
<td>-</td>
<td>(Z^+)</td>
<td>GetTransformFeedbackVarying</td>
<td>-</td>
<td>Type of each transform feedback varying variable</td>
<td>2.11.6</td>
</tr>
<tr>
<td>-</td>
<td>(0^+ \times \text{char})</td>
<td>GetTransformFeedbackVarying</td>
<td>-</td>
<td>Name of each transform feedback varying variable</td>
<td>2.11.6</td>
</tr>
</tbody>
</table>
### 6.2. STATE TABLES

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Type</th>
<th>Initial Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetIntegerv</td>
<td>$Z^+$</td>
<td>0</td>
<td>Uniform buffer object bound to the context for buffer object manipulation</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>$n \times Z^+$</td>
<td>0</td>
<td>Uniform buffer object bound to the specified context binding point</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>$n \times Z^+$</td>
<td>0</td>
<td>Size of bound uniform buffer region</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>$n \times Z^+$</td>
<td>0</td>
<td>Number of active uniform blocks in a program</td>
</tr>
<tr>
<td>GetProgramiv</td>
<td>$Z^+$</td>
<td>0</td>
<td>Start of bound uniform buffer region</td>
</tr>
<tr>
<td>GetProgramiv</td>
<td>$0 \times Z^+$</td>
<td>0</td>
<td>Length of longest active uniform block name</td>
</tr>
<tr>
<td>GetActiveUniformsiv</td>
<td>$0 \times Z^+$</td>
<td>0</td>
<td>Type of active uniform</td>
</tr>
<tr>
<td>GetActiveUniformsiv</td>
<td>$0 \times Z^+$</td>
<td>0</td>
<td>Size of active uniform</td>
</tr>
<tr>
<td>GetActiveUniformsiv</td>
<td>$0 \times Z^+$</td>
<td>0</td>
<td>Uniform name length</td>
</tr>
<tr>
<td>GetActiveUniformsiv</td>
<td>$0 \times Z^+$</td>
<td>0</td>
<td>Uniform block index</td>
</tr>
<tr>
<td>GetActiveUniformsiv</td>
<td>$0 \times Z^+$</td>
<td>0</td>
<td>Uniform buffer offset</td>
</tr>
</tbody>
</table>

Table 6.30. Program Object State (cont.)
### Table 6.31. Program Object State (cont.)

| Uniform Buffer Array | Description | Uniform Buffer intra-matrix stride | Uniform Buffer intra-matrix row-major matrix size | Uniform Buffer intra-matrix row-major matrix size of the specified uniform block | Uniform Buffer intra-matrix row-major matrix size of the specified uniform block's data | Uniform Buffer intra-matrix row-major matrix count of active uniforms in the specified uniform block | Uniform Buffer intra-matrix row-major matrix array of active uniform indices of the specified uniform block | Uniform Buffer binding points associated with the specified uniform block's data | Uniform Buffer binding points associated with the specified uniform block’s data | Uniform Buffer binding points associated with the specified uniform block’s data of the vertex stage | Uniform Buffer binding points associated with the specified uniform block’s data of the geometry stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the vertex stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the geometry stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the fragment stage |
|----------------------|-------------|-----------------------------------|-----------------------------------------------|--------------------------------------------------------------------------------|--------------------------------------------------------------------------------|---------------------------------------------------------------------------|--------------------------------------------------------------------------|-------------------------------------------------------------------------|------------------------------------------------------------------------|------------------------------------------------------------------------|------------------------------------------------------------------------|------------------------------------------------------------------------|-----------------------------------------------------------------------|------------------------------------------------------------------------|------------------------------------------------------------------------|------------------------------------------------------------------------|
| UNIFORM ARRAY STRIDE | Uniform Buffer array | Uniform Buffer intra-matrix stride | Uniform Buffer intra-matrix row-major matrix size | Uniform Buffer intra-matrix row-major matrix size of the specified uniform block | Uniform Buffer intra-matrix row-major matrix size of the specified uniform block’s data | Uniform Buffer intra-matrix row-major matrix count of active uniforms in the specified uniform block | Uniform Buffer intra-matrix row-major matrix array of active uniform indices of the specified uniform block | Uniform Buffer binding points associated with the specified uniform block’s data | Uniform Buffer binding points associated with the specified uniform block’s data | Uniform Buffer binding points associated with the specified uniform block’s data of the vertex stage | Uniform Buffer binding points associated with the specified uniform block’s data of the geometry stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the vertex stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the geometry stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the fragment stage |
| UNIFORM MATH STRIDE | Uniform Buffer array | Uniform Buffer intra-matrix stride | Uniform Buffer intra-matrix row-major matrix size | Uniform Buffer intra-matrix row-major matrix size of the specified uniform block | Uniform Buffer intra-matrix row-major matrix size of the specified uniform block’s data | Uniform Buffer intra-matrix row-major matrix count of active uniforms in the specified uniform block | Uniform Buffer intra-matrix row-major matrix array of active uniform indices of the specified uniform block | Uniform Buffer binding points associated with the specified uniform block’s data | Uniform Buffer binding points associated with the specified uniform block’s data | Uniform Buffer binding points associated with the specified uniform block’s data of the vertex stage | Uniform Buffer binding points associated with the specified uniform block’s data of the geometry stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the vertex stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the geometry stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the fragment stage |
| UNIFORM BLOCK BINDING | Uniform Buffer array | Uniform Buffer intra-matrix stride | Uniform Buffer intra-matrix row-major matrix size | Uniform Buffer intra-matrix row-major matrix size of the specified uniform block | Uniform Buffer intra-matrix row-major matrix size of the specified uniform block’s data | Uniform Buffer intra-matrix row-major matrix count of active uniforms in the specified uniform block | Uniform Buffer intra-matrix row-major matrix array of active uniform indices of the specified uniform block | Uniform Buffer binding points associated with the specified uniform block’s data | Uniform Buffer binding points associated with the specified uniform block’s data | Uniform Buffer binding points associated with the specified uniform block’s data of the vertex stage | Uniform Buffer binding points associated with the specified uniform block’s data of the geometry stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the vertex stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the geometry stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the fragment stage |
| UNIFORM BLOCK DATA SIZE | Uniform Buffer array | Uniform Buffer intra-matrix stride | Uniform Buffer intra-matrix row-major matrix size | Uniform Buffer intra-matrix row-major matrix size of the specified uniform block | Uniform Buffer intra-matrix row-major matrix size of the specified uniform block’s data | Uniform Buffer intra-matrix row-major matrix count of active uniforms in the specified uniform block | Uniform Buffer intra-matrix row-major matrix array of active uniform indices of the specified uniform block | Uniform Buffer binding points associated with the specified uniform block’s data | Uniform Buffer binding points associated with the specified uniform block’s data | Uniform Buffer binding points associated with the specified uniform block’s data of the vertex stage | Uniform Buffer binding points associated with the specified uniform block’s data of the geometry stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the vertex stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the geometry stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the fragment stage |
| UNIFORM BLOCK ACTIVE UNIFORMS | Uniform Buffer array | Uniform Buffer intra-matrix stride | Uniform Buffer intra-matrix row-major matrix size | Uniform Buffer intra-matrix row-major matrix size of the specified uniform block | Uniform Buffer intra-matrix row-major matrix size of the specified uniform block’s data | Uniform Buffer intra-matrix row-major matrix count of active uniforms in the specified uniform block | Uniform Buffer intra-matrix row-major matrix array of active uniform indices of the specified uniform block | Uniform Buffer binding points associated with the specified uniform block’s data | Uniform Buffer binding points associated with the specified uniform block’s data | Uniform Buffer binding points associated with the specified uniform block’s data of the vertex stage | Uniform Buffer binding points associated with the specified uniform block’s data of the geometry stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the vertex stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the geometry stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the fragment stage |
| UNIFORM BLOCK REFERENCED BY VERTEX SHADER | Uniform Buffer array | Uniform Buffer intra-matrix stride | Uniform Buffer intra-matrix row-major matrix size | Uniform Buffer intra-matrix row-major matrix size of the specified uniform block | Uniform Buffer intra-matrix row-major matrix size of the specified uniform block’s data | Uniform Buffer intra-matrix row-major matrix count of active uniforms in the specified uniform block | Uniform Buffer intra-matrix row-major matrix array of active uniform indices of the specified uniform block | Uniform Buffer binding points associated with the specified uniform block’s data | Uniform Buffer binding points associated with the specified uniform block’s data | Uniform Buffer binding points associated with the specified uniform block’s data of the vertex stage | Uniform Buffer binding points associated with the specified uniform block’s data of the geometry stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the vertex stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the geometry stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the fragment stage |
| UNIFORM BLOCK REFERENCED BY GEOMETRY SHADER | Uniform Buffer array | Uniform Buffer intra-matrix stride | Uniform Buffer intra-matrix row-major matrix size | Uniform Buffer intra-matrix row-major matrix size of the specified uniform block | Uniform Buffer intra-matrix row-major matrix size of the specified uniform block’s data | Uniform Buffer intra-matrix row-major matrix count of active uniforms in the specified uniform block | Uniform Buffer intra-matrix row-major matrix array of active uniform indices of the specified uniform block | Uniform Buffer binding points associated with the specified uniform block’s data | Uniform Buffer binding points associated with the specified uniform block’s data | Uniform Buffer binding points associated with the specified uniform block’s data of the vertex stage | Uniform Buffer binding points associated with the specified uniform block’s data of the geometry stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the vertex stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the geometry stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the fragment stage |
| UNIFORM BLOCK REFERENCED BY FRAGMENT SHADER | Uniform Buffer array | Uniform Buffer intra-matrix stride | Uniform Buffer intra-matrix row-major matrix size | Uniform Buffer intra-matrix row-major matrix size of the specified uniform block | Uniform Buffer intra-matrix row-major matrix size of the specified uniform block’s data | Uniform Buffer intra-matrix row-major matrix count of active uniforms in the specified uniform block | Uniform Buffer intra-matrix row-major matrix array of active uniform indices of the specified uniform block | Uniform Buffer binding points associated with the specified uniform block’s data | Uniform Buffer binding points associated with the specified uniform block’s data | Uniform Buffer binding points associated with the specified uniform block’s data of the vertex stage | Uniform Buffer binding points associated with the specified uniform block’s data of the geometry stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the vertex stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the geometry stage | Uniform Buffer binding points associated with the specified uniform block’s data of the fragment stage of the fragment stage |

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Type</th>
<th>Initial Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetActiveUniformiv</td>
<td>UNIFORM</td>
<td>0 × Z</td>
<td>Uniform buffer array</td>
</tr>
<tr>
<td>GetActiveUniformBlockiv</td>
<td>BLOCK</td>
<td>0 × Z</td>
<td>Uniform buffer intra-matrix stride</td>
</tr>
<tr>
<td>GetActiveUniformBlockiv</td>
<td>BLOCK</td>
<td>0 × Z</td>
<td>Uniform buffer intra-matrix row-major matrix size</td>
</tr>
<tr>
<td>GetActiveUniformBlockiv</td>
<td>BLOCK</td>
<td>0 × Z</td>
<td>Uniform buffer intra-matrix row-major matrix size of the specified uniform block</td>
</tr>
<tr>
<td>GetActiveUniformBlockiv</td>
<td>BLOCK</td>
<td>0 × Z</td>
<td>Uniform buffer intra-matrix row-major matrix size of the specified uniform block’s data</td>
</tr>
<tr>
<td>GetActiveUniformBlockiv</td>
<td>BLOCK</td>
<td>0 × Z</td>
<td>Uniform buffer intra-matrix row-major matrix count of active uniforms in the specified uniform block</td>
</tr>
<tr>
<td>GetActiveUniformBlockiv</td>
<td>BLOCK</td>
<td>0 × Z</td>
<td>Uniform buffer intra-matrix row-major matrix array of active uniform indices of the specified uniform block</td>
</tr>
<tr>
<td>GetActiveUniformBlockiv</td>
<td>BLOCK</td>
<td>0 × Z</td>
<td>Uniform buffer binding points associated with the specified uniform block’s data</td>
</tr>
<tr>
<td>GetActiveUniformBlockiv</td>
<td>BLOCK</td>
<td>0 × Z</td>
<td>Uniform buffer binding points associated with the specified uniform block’s data of the vertex stage</td>
</tr>
<tr>
<td>GetActiveUniformBlockiv</td>
<td>BLOCK</td>
<td>0 × Z</td>
<td>Uniform buffer binding points associated with the specified uniform block’s data of the geometry stage</td>
</tr>
<tr>
<td>GetActiveUniformBlockiv</td>
<td>BLOCK</td>
<td>0 × Z</td>
<td>Uniform buffer binding points associated with the specified uniform block’s data of the fragment stage</td>
</tr>
</tbody>
</table>

**Notes:**
- The table contains various fields related to uniform buffer management, including initial values and descriptions of commands and type specifications for different objects such as uniform buffers, matrices, and active uniform blocks.
- Each row represents a specific uniform buffer attribute with its corresponding command, initial value, and description.
- The table is comprehensive, covering aspects like buffer array, intra-matrix stride, row-major matrix size, and various references to vertex, geometry, and fragment stages.
### Table 6.32. Vertex and Geometry Shader 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>CURRENT_VERTEX_ATTRIB</td>
<td>$16 \times R^4$</td>
<td>GetVertexAttribfv</td>
<td>0.0,0.0,0.0,1.0</td>
<td>Current generic vertex attribute values</td>
<td>2.7</td>
</tr>
<tr>
<td>PROGRAM_POINT_SIZE</td>
<td>$B$</td>
<td>isEnabled</td>
<td>FALSE</td>
<td>Point size mode</td>
<td>3.4</td>
</tr>
</tbody>
</table>
### Table 6.33. Query Object State

<table>
<thead>
<tr>
<th>Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetQueryObjectAVAILABLE</td>
<td>FALSE</td>
<td>Is the query object result available?</td>
<td>6.1.6</td>
</tr>
<tr>
<td>GetQueryObjectRESULT</td>
<td>0</td>
<td>Query object result</td>
<td>6.1.6</td>
</tr>
<tr>
<td>GetQueryObject</td>
<td></td>
<td>Query object result</td>
<td>6.1.6</td>
</tr>
<tr>
<td>SETRESULT</td>
<td></td>
<td>Query object result</td>
<td>6.1.6</td>
</tr>
</tbody>
</table>

OpenGL 3.2 (Core Profile) - July 24, 2009
<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>6.1.8</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>6.1.8</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>6.1.8</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>6.1.8</td>
</tr>
<tr>
<td>MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>64</td>
<td>Max number of components to write to a single buffer in interleaved mode</td>
<td>2.16</td>
</tr>
<tr>
<td>MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>4</td>
<td>Max number of separate attributes or varyings that can be captured in transform feedback</td>
<td>2.16</td>
</tr>
<tr>
<td>MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>4</td>
<td>Max number of components per attribute or varying in separate mode</td>
<td>2.16</td>
</tr>
<tr>
<td>Get Command</td>
<td>Get Sync</td>
<td>Type</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>GetSync</td>
<td>Z_1</td>
<td>Z</td>
<td>GetSynciv</td>
<td>Type of sync object</td>
<td></td>
</tr>
<tr>
<td>GetSync</td>
<td>Z_2</td>
<td>Z</td>
<td>Unsignal</td>
<td>Sync object status</td>
<td></td>
</tr>
<tr>
<td>GetSync</td>
<td>Z_1</td>
<td>Z</td>
<td>SYNC_GPU_COMMANDS_COMPLETE</td>
<td>Sync object condition</td>
<td></td>
</tr>
<tr>
<td>GetSync</td>
<td>Z</td>
<td>Z</td>
<td>Sync flags</td>
<td>Sync object flags</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.35. 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>DONT_CARE</td>
<td>Line smooth hint</td>
<td>5.3</td>
</tr>
<tr>
<td>POLYGON_SMOOTH_HINT</td>
<td>$Z_3$</td>
<td>GetIntegerv</td>
<td>DONT_CARE</td>
<td>Polygon smooth hint</td>
<td>5.3</td>
</tr>
<tr>
<td>TEXTURE_COMPRESSION_HINT</td>
<td>$Z_3$</td>
<td>GetIntegerv</td>
<td>DONT_CARE</td>
<td>Texture compression quality hint</td>
<td>5.3</td>
</tr>
<tr>
<td>FRAGMENT_SHADER_DERIVATIVE_HINT</td>
<td>$Z_3$</td>
<td>GetIntegerv</td>
<td>DONT_CARE</td>
<td>Fragment shader derivative accuracy hint</td>
<td>5.3</td>
</tr>
</tbody>
</table>
### Table 6.37. Implementation Dependent Values

<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_CLIP_DISTANCES</td>
<td>Z⁺</td>
<td>GetIntegerv</td>
<td>8</td>
<td>Maximum number of user clipping planes</td>
<td>2.19</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>256</td>
<td>Maximum 3D texture image dimension</td>
<td>3.8.1</td>
</tr>
<tr>
<td>MAX_TEXTURE_SIZE</td>
<td>Z⁺</td>
<td>GetIntegerv</td>
<td>1024</td>
<td>Maximum 2D/1D texture image dimension</td>
<td>3.8.1</td>
</tr>
<tr>
<td>MAX_ARRAY_TEXTURE_LAYERS</td>
<td>Z⁺</td>
<td>GetIntegerv</td>
<td>256</td>
<td>Maximum number of layers for texture arrays</td>
<td>3.8.1</td>
</tr>
<tr>
<td>MAX_TEXTURE_1D_BIAS</td>
<td>R⁺</td>
<td>GetFloatv</td>
<td>2.0</td>
<td>Maximum absolute texture level of detail bias</td>
<td>3.8.9</td>
</tr>
<tr>
<td>MAX_CUBE_MAP_TEXTURE_SIZE</td>
<td>Z⁺</td>
<td>GetIntegerv</td>
<td>1024</td>
<td>Maximum cube map texture image dimension</td>
<td>3.8.1</td>
</tr>
<tr>
<td>MAX_RENDERBUFFER_SIZE</td>
<td>Z⁺</td>
<td>GetIntegerv</td>
<td>1024</td>
<td>Maximum width and height of renderbuffers</td>
<td>4.4.2</td>
</tr>
<tr>
<td>Get value</td>
<td>Get Command</td>
<td>Minimum Value</td>
<td>Description</td>
<td>Sec.</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>---------------</td>
<td>---------------------------------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>MAX_VIEWPORT_DIMS</td>
<td>GetIntegerv</td>
<td>see 2.13.1</td>
<td>Maximum viewport dimensions</td>
<td>2.13.1</td>
<td></td>
</tr>
<tr>
<td>POINT_SIZE_RANGE</td>
<td>GetFloatv</td>
<td>1.1</td>
<td>Range (lo to hi) of point sprite sizes</td>
<td>3.4</td>
<td></td>
</tr>
<tr>
<td>POINT_SIZE_GRANULARITY</td>
<td>GetFloatv</td>
<td>-</td>
<td>Point sprite size granularity</td>
<td>3.4</td>
<td></td>
</tr>
<tr>
<td>2 × Z+</td>
<td>GetIntegerv</td>
<td>2 × R+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 × R+</td>
<td>GetFloatv</td>
<td>R+</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.38. 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>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 (v1.1: 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 (v1.1: 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>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>COMPRESSED_TEXTURE_FORMATS</td>
<td>$4 \times Z^+$</td>
<td>GetIntegerv</td>
<td>–</td>
<td>Enumerated compressed texture formats</td>
<td>3.8.3</td>
</tr>
<tr>
<td>NUM_COMPRESSED_TEXTURE_FORMATS</td>
<td>$Z$</td>
<td>GetIntegerv</td>
<td>4</td>
<td>Number of compressed texture formats</td>
<td>3.8.3</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>3.8.5</td>
</tr>
<tr>
<td>MAX_RECTANGLE_TEXTURE_SIZE</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>1024</td>
<td>Max. width &amp; height of rectangular textures</td>
<td>3.8.1</td>
</tr>
</tbody>
</table>
## 6.2. State Tables

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Minimum Value</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetStringI</td>
<td>–</td>
<td>Z+</td>
<td>Supported individual extension names</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>–</td>
<td>Z+</td>
<td>Number of individual extension names</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>3</td>
<td>Z+</td>
<td>Major version number supported</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>–</td>
<td>Z+</td>
<td>Minor version number supported</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>–</td>
<td>Z+</td>
<td>Context full/forward-compatible flag</td>
</tr>
<tr>
<td>GetString</td>
<td>–</td>
<td>S</td>
<td>Renderer string</td>
</tr>
<tr>
<td>GetString</td>
<td>–</td>
<td>S</td>
<td>Shading Language version supported</td>
</tr>
<tr>
<td>GetString</td>
<td>–</td>
<td>S</td>
<td>Vendor string</td>
</tr>
<tr>
<td>GetString</td>
<td>–</td>
<td>S</td>
<td>OpenGL version supported</td>
</tr>
</tbody>
</table>

Table 6.40. Implementation Dependent Version and Extension Support
### Table 6.41. Implementation Dependent Vertex Shader Limits

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Type</th>
<th>Get value</th>
<th>Minimum Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetIntegerv</td>
<td>Z+</td>
<td>MAX_VERTEX_ATTRIBS</td>
<td>16</td>
<td>Number of active vertex attributes</td>
<td>2.7</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>Z+</td>
<td>MAX_VERTEX_UNIFORM_COMPONENTS</td>
<td>1024</td>
<td>Number of components for vertex shader uniform variables</td>
<td>2.11.4</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>Z+</td>
<td>MAX_VERTEX_UNIFORM_BLOCKS</td>
<td>12</td>
<td>Max number of vertex uniform buffers per program</td>
<td>2.11.4</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>Z+</td>
<td>MAX_VERTEX_OUTPUT_COMPONENTS</td>
<td>64</td>
<td>Max number of components of outputs written by a vertex shader</td>
<td>2.11.6</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>Z+</td>
<td>MAX_VERTEX_TEXTURE_IMAGE_UNITS</td>
<td>16</td>
<td>Number of texture image units accessible by a vertex shader</td>
<td>2.11.7</td>
</tr>
</tbody>
</table>

OpenGL 3.2 (Core Profile) - July 24, 2009
### Table 6.42. Implementation Dependent Geometry Shader Limits

<table>
<thead>
<tr>
<th>Type</th>
<th>Get Command</th>
<th>Get Value</th>
<th>Minimum Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_GEOMETRY_UNIFORM_BLOCKS</td>
<td>GetIntegerv</td>
<td>Z+</td>
<td>12</td>
<td>Max number of geometry uniform buffers per program</td>
</tr>
<tr>
<td>MAX_GEOMETRY_INPUT_COMPONENTS</td>
<td>GetIntegerv</td>
<td>Z+</td>
<td>64</td>
<td>Max number of components of inputs read by a geometry shader</td>
</tr>
<tr>
<td>MAX_GEOMETRY_OUTPUT_COMPONENTS</td>
<td>GetIntegerv</td>
<td>Z+</td>
<td>128</td>
<td>Max number of components of outputs written by a geometry shader</td>
</tr>
<tr>
<td>MAX_GEOMETRY_INPUT_OUTPUT_COMPONENTS</td>
<td>GetIntegerv</td>
<td>Z+</td>
<td>256</td>
<td>Maximum number of vertices that any geometry shader can emit</td>
</tr>
<tr>
<td>MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS</td>
<td>GetIntegerv</td>
<td>Z+</td>
<td>1024</td>
<td>Maximum number of total components (all vertices) of active varyings that a geometry shader can emit</td>
</tr>
<tr>
<td>MAX_GEOMETRY_TEXTURE_IMAGE_UNITS</td>
<td>GetIntegerv</td>
<td>Z+</td>
<td>16</td>
<td>Number of texture image units accessible by a geometry shader</td>
</tr>
</tbody>
</table>
### Table 6.43. Implementation Dependent Fragment Processing 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><code>MAX_FRAGMENT_UNIFORM_COMPONENTS</code></td>
<td>$Z^+$</td>
<td><code>GetIntegerv</code></td>
<td>1024</td>
<td>Number of components for frag. shader uniform variables</td>
<td>3.9.1</td>
</tr>
<tr>
<td><code>MAX_FRAGMENT_UNIFORM_BLOCKS</code></td>
<td>$Z^+$</td>
<td><code>GetIntegerv</code></td>
<td>12</td>
<td>Max number of fragment uniform buffers per program</td>
<td>2.11.4</td>
</tr>
<tr>
<td><code>MAX_FRAGMENT_INPUT_COMPONENTS</code></td>
<td>$Z^+$</td>
<td><code>GetIntegerv</code></td>
<td>128</td>
<td>Max number of components of inputs read by a fragment shader</td>
<td>3.9.2</td>
</tr>
<tr>
<td><code>MAX_TEXTURE_IMAGE_UNITS</code></td>
<td>$Z^+$</td>
<td><code>GetIntegerv</code></td>
<td>16</td>
<td>Number of texture image units accessible by a fragment shader</td>
<td>2.11.7</td>
</tr>
<tr>
<td><code>MIN_PROGRAM_TEXEL_OFFSET</code></td>
<td>$Z$</td>
<td><code>GetIntegerv</code></td>
<td>-8</td>
<td>Minimum texel offset allowed in lookup</td>
<td>2.11.7</td>
</tr>
<tr>
<td><code>MAX_PROGRAM_TEXEL_OFFSET</code></td>
<td>$Z$</td>
<td><code>GetIntegerv</code></td>
<td>7</td>
<td>Maximum texel offset allowed in lookup</td>
<td>2.11.7</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_UNIFORM_BUFFER_BINDINGS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>24</td>
<td>Max number of uniform buffer binding points on the context</td>
<td>2.11.4</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>2.11.4</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>2.11.4</td>
</tr>
<tr>
<td>MAX_COMBINED_UNIFORM_BLOCKS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>24</td>
<td>Max number of uniform buffers per program</td>
<td>2.11.4</td>
</tr>
<tr>
<td>MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>$\dagger$</td>
<td>Number of words for vertex shader uniform variables in all uniform blocks (including default)</td>
<td>2.11.4</td>
</tr>
<tr>
<td>MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>$\dagger$</td>
<td>Number of words for fragment shader uniform variables in all uniform blocks (including default)</td>
<td>2.11.4</td>
</tr>
<tr>
<td>MAX_COMBINED_GEOMETRY_UNIFORM_COMPONENTS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>$\dagger$</td>
<td>Number of words for geometry shader uniform variables in all uniform blocks (including default)</td>
<td>2.11.4</td>
</tr>
<tr>
<td>MAX_VARYING_COMPONENTS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>60</td>
<td>Number of components for varying variables</td>
<td>2.11.6</td>
</tr>
<tr>
<td>MAX_COMBINED_TEXTURE_IMAGE_UNITS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>48</td>
<td>Total number of texture units accessible by the GL</td>
<td>2.11.7</td>
</tr>
</tbody>
</table>

† The minimum value for each stage is \( \text{MAX}_{\text{stage}} \_\text{UNIFORM\_BLOCK\_SIZE} \times \text{MAX}_{\text{stage}} \_\text{UNIFORM\_BLOCK\_SIZE} \) + \( \text{MAX}_{\text{stage}} \_\text{UNIFORM\_COMPONENTS} \).
<table>
<thead>
<tr>
<th>Get value</th>
<th>Command</th>
<th>Minimum Value</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXSAMPLEMASKWORDS</td>
<td>GetIntegerv</td>
<td>1</td>
<td>$Z^+$</td>
<td>Maximum number of sample mask words</td>
</tr>
<tr>
<td>MAXCOLORTEXTURESAMPLES</td>
<td>GetIntegerv</td>
<td>1</td>
<td>$Z^+$</td>
<td>Maximum number of samples in a color multisample texture</td>
</tr>
<tr>
<td>MAXDEPTHTEXTURESAMPLES</td>
<td>GetIntegerv</td>
<td>1</td>
<td>$Z^+$</td>
<td>Maximum number of samples in a depth/stencil multisample texture</td>
</tr>
<tr>
<td>MAXINTEGERSAMPLES</td>
<td>GetIntegerv</td>
<td>1</td>
<td>$Z^+$</td>
<td>Maximum number of samples in integer format multisample buffers</td>
</tr>
<tr>
<td>QUADCOLORINDEXPRIMITIVES</td>
<td>GetBooleanv</td>
<td>–</td>
<td></td>
<td>Whether quads follow provoking vertex convention</td>
</tr>
<tr>
<td>QUERYCOUNTERBITS</td>
<td>GetQueryiv</td>
<td>see 6.1.6</td>
<td></td>
<td>Asynchronous counter bits</td>
</tr>
<tr>
<td>MAXSERVERWAITTIMEOUT</td>
<td>GetIntegerv$64v$</td>
<td>0</td>
<td>$Z^+$</td>
<td>Maximum WaitSync timeout interval</td>
</tr>
</tbody>
</table>

Table 6.45. Implementation Dependent Values (cont.)
### Table 6.46. Framebuffer Dependent Values

<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_DRAW_BUFFERS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>8</td>
<td>Maximum number of active draw buffers</td>
<td>4.2.1</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>GetMultisamplev</td>
<td>–</td>
<td>Explicit sample positions</td>
<td>3.3.1</td>
</tr>
<tr>
<td>MAX_COLOR_ATTACHMENTS</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>8</td>
<td>Maximum number of FBO attachment points for color buffers</td>
<td>4.4.2</td>
</tr>
<tr>
<td>MAX_SAMPLES</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>4</td>
<td>Maximum number of samples supported for multisampling</td>
<td>4.4.2</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>-</td>
<td>$n \times Z_5$</td>
<td>GetError</td>
<td>0</td>
<td>Current error code(s)</td>
<td>2.5</td>
</tr>
<tr>
<td>-</td>
<td>$n \times B$</td>
<td>–</td>
<td>FALSE</td>
<td>True if there is a corresponding error</td>
<td>2.5</td>
</tr>
<tr>
<td>B</td>
<td>–</td>
<td>FALSE</td>
<td>Occlusion query active</td>
<td>4.1.6</td>
<td></td>
</tr>
<tr>
<td>CURRENTQUERY</td>
<td>$3 \times Z^+$</td>
<td>GetQueryiv</td>
<td>0</td>
<td>Active query object names</td>
<td>6.1.6</td>
</tr>
<tr>
<td>COPY,READ,BUFFER</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Buffer object bound to copy buffer “read” bind point</td>
<td>2.9.5</td>
</tr>
<tr>
<td>COPY,WRITE,BUFFER</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Buffer object bound to copy buffer “write” bind point</td>
<td>2.9.5</td>
</tr>
<tr>
<td>TEXTURE,CUBE,MAP,SEAMLESS</td>
<td>$B$</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Seamless cube map filtering enable</td>
<td>3.8.8</td>
</tr>
</tbody>
</table>

Table 6.47. 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.

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.4 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 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 con-
sistent 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, poly-
gons 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 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 \(\text{block-size}\) (8 or 16 bytes) is decoded, the corresponding image size (in bytes) is:

\[
\left\lfloor \frac{w}{4} \right\rfloor \times \left\lfloor \frac{h}{4} \right\rfloor \times \text{blocksize}.
\]

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:

\[
\text{blocksize} \times \left( \left\lfloor \frac{w}{4} \right\rfloor \times \left\lfloor \frac{y}{4} \right\rfloor + \left\lfloor \frac{x}{4} \right\rfloor \right) .
\]

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:
C.1.1 Format COMPRESSED_RED_RGTC1

Each 4 × 4 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):

\[ red_0, red_1, bits_0, bits_1, bits_2, bits_3, bits_4, bits_5 \]

The 6 \( bits \) bytes of the block are decoded into a 48-bit bit vector:

\[ bits = bits_0 + 256 \times (bits_1 + 256 \times (bits_2 + 256 \times (bits_3 + 256 \times (bits_4 + 256 \times bits_5)))) \]

\( red_0 \) and \( red_1 \) are 8-bit unsigned integers that are unpacked to red values \( RED_0 \) and \( RED_1 \).

\( bits \) 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:

\[ code(x, y) = bits \left[ 3 \times (4 \times y + x) + 2 \ldots 3 \times (4 \times y + x) + 0 \right] \]

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:

\[
R = \begin{cases} 
RED_0, & red_0 > red_1, \text{code}(x, y) = 0 \\
RED_1, & red_0 > red_1, \text{code}(x, y) = 1 \\
6RED_0 + RED_1, & red_0 > red_1, \text{code}(x, y) = 2 \\
5RED_0 + 2RED_1, & red_0 > red_1, \text{code}(x, y) = 3 \\
4RED_0 + 3RED_1, & red_0 > red_1, \text{code}(x, y) = 4 \\
3RED_0 + 4RED_1, & red_0 > red_1, \text{code}(x, y) = 5 \\
2RED_0 + 5RED_1, & red_0 > red_1, \text{code}(x, y) = 6 \\
RED_0 + 6RED_1, & red_0 > red_1, \text{code}(x, y) = 7 \\
RED_0, & red_0 \leq red_1, \text{code}(x, y) = 0 \\
RED_1, & red_0 \leq red_1, \text{code}(x, y) = 1 \\
4RED_0 + RED_1, & red_0 \leq red_1, \text{code}(x, y) = 2 \\
3RED_0 + 2RED_1, & red_0 \leq red_1, \text{code}(x, y) = 3 \\
2RED_0 + 3RED_1, & red_0 \leq red_1, \text{code}(x, y) = 4 \\
RED_{\text{min}}, & red_0 \leq red_1, \text{code}(x, y) = 5 \\
RED_{\text{max}}, & red_0 \leq red_1, \text{code}(x, y) = 6 \\
\end{cases}
\]
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 texel is \((R, 0, 0, 1)\).

C.1.2 Format \texttt{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 \texttt{COMPRESSED\_RED\_RGTC1} except \(\text{red}_0, \text{red}_1, \text{RED}_0, \text{RED}_1, \text{RED}_{min},\) and \(\text{RED}_{max}\) are signed values defined as follows:

- \(\text{red}_0\) and \(\text{red}_1\) are 8-bit signed (twos complement) integers.

\[
\text{RED}_0 = \begin{cases} \frac{\text{red}_0}{127}, & \text{red}_0 > -128 \\ -1.0, & \text{red}_0 = -128 \end{cases}
\]

\[
\text{RED}_1 = \begin{cases} \frac{\text{red}_1}{127}, & \text{red}_1 > -128 \\ -1.0, & \text{red}_1 = -128 \end{cases}
\]

\(\text{RED}_{min} = -1.0\)

\(\text{RED}_{max} = 1.0\)

CAVEAT for signed \(\text{red}_0\) and \(\text{red}_1\) values: the expressions \(\text{red}_0 > \text{red}_1\) and \(\text{red}_0 \leq \text{red}_1\) above are considered undefined (read: may vary by implementation) when \(\text{red}_0 = -127\) and \(\text{red}_1 = -128\). This is because if \(\text{red}_0\) 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 \(\text{red}_0 = -127\) and \(\text{red}_1 = -128\).

C.1.3 Format \texttt{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 \texttt{COMPRESSED\_RED\_RGTC1} above.

The second 64 bits of compressed green are decoded exactly like \texttt{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 \((R, G, 0, 1)\).
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 $(R, G, 0, 1)$. 
Appendix D

Shared Objects and Multiple Contexts

This appendix describes special considerations for objects shared between multiple OpenGL context, including deletion behavior and how changes to shared objects are propagated between contexts.

Objects that can be shared between contexts include pixel and vertex buffer objects, program and shader objects, renderbuffer objects, sync objects, and texture objects (except for the texture objects named zero).

Framebuffer, query, and vertex array objects 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 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. Bind points in other contexts are not affected.
D.1.2 Deleted Object and Object Name Lifetimes

When a buffer, texture, renderbuffer, 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.

When a program or shader 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 program object is in use while it is the current program in any context. A shader object is in use while it is attached to any program object.

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 State Changes

Data is information the GL implementation does not have to inspect, and does not
D.3. PROPAGATING STATE CHANGES

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 state of an object $T$ is changed, such changes are not always immediately visible, and do not always immediately affect GL operations involving that object. Changes to an object 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.

D.3.1 Determining Completion of Changes to an object

The object $T$ is considered to have been changed once a command such as described in section D.3 has completed. Completion of a command \footnote{The 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.} 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. PROPAGATING STATE CHANGES

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.

- 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 state of object \( T \) is changed in the current context while \( T \) is directly or indirectly attached, then all operations on \( T \) will use that new state in the current context.

  Note: The intent of this rule is to address state 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 \( C \)’s attachments in the current context are guaranteed to be seen. To guarantee seeing changes made in another context to objects attached to \( C \) must be completed in that other context (see section D.3.1) prior to \( C \) being bound. Changes made in another
D.3. PROPAGATING STATE CHANGES

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** State changes to shared objects are not automatically propagated between contexts. If the state of a shared object T is 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 state.

**Rule 4** If the state of a shared object T is changed in a context other than the current context, and T is already directly or indirectly attached to the current context at multiple attachment or bind points, it must be attached or re-attached to at least one binding point in the current context in order for the new state of T to be 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: This rule also applies to the pointer to the data store of an object. The pointer itself is state, while the content of the data store are data, not state. To guarantee that another context sees data updates to an object, you should attach or re-attach the object in that context, since the pointer to the data store could have changed.

Note: To be sure that a data update, as the result of 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 modified in another context, re-binding the texture at any one of the texture bind points is sufficient to cause the modifications 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 I.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, ShadeModel, and Enable/Disable targets LIGHTING, VERTEX_PROGRAM_TWO-SIDE, LIGHTi, 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 ClampColor 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 VertexAttribPointer 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 vertex 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 builtins 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 Enable/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.
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 3.8.15 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 $(r, r, r, 1)$ 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 3.8.1); all language in section 3.8 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 3.8.2).

- 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
E.2. DEPRECATED AND REMOVED FEATURES

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 5.3).

• 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_gsgpu_shader4).
- Conditional rendering (GL_NV_conditional_render).
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

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.

<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_DISTANCEi</td>
<td>CLIP_PLANEi</td>
</tr>
</tbody>
</table>

Table F.1: New token names and the old names they replace.

### 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 6.1.11 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 2.14 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 2.11.7 and note that only one of **gl_ClipVertex** and **gl_ClipDistance** should be written by a shader (bug 3898).
- 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 6.1.11 to accept `DEPTH_BUFFER / DEPTH_ATTACHMENT` and `STENCIL_BUFFER / STENCIL_ATTACHMENT` interchangeably (bug 3744).

- Add proper type suffix to query commands in tables 6.4 and 6.32 (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 2.7, 2.8, and 6.7 - 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 2.13.1, 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 2.16 - define “word-aligned” to be a multiple of 4 (e.g. 32 bits) (bug 3624).

- Section 6.1.8 - 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 6.1.11 - Moved `GetFramebufferAttachmentiv` query from section 6.1.3. Querying framebuffer attachment parameters other than object...
type and name when no attachment is present is an INVALID_ENUM error.
Querying texture parameters (level, cube map face, or layer) for a render-
buffer attachment is also an INVALID_ENUM error (note that this was allowed
in previous versions of the extension but the return values were not specified;
it should clearly be an error as are other parameters that don’t exist for the
type of attachment present). Also reorganized the description of this com-
mand quite a bit to improve readability and remove redundancy and internal
inconsistencies (bug 3697).

- Section 6.1.12 - Moved GetRenderbufferParameteriv query from sec-
tion 6.1.3 (side effect of bug 3697).
- Appendix D.1 - add language to clarify that attachments to an object affect
its reference count, and that object storage doesn’t go away until there are no
references remaining (bug 3725).
- Appendix E.2 - remove TEXTURE_BORDER_COLOR and CLAMP_TO_BORDER
mode from the deprecated feature list; they were put in by accident (bug
3750).
- Appendix F - Cite GL_EXT_texture_array instead of GL_EXT_-
geometry_shader4 as the source of 1D/2D array texture functionality. Fix
a typo. Add change log relative to initial 3.0 spec release.

F.5 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 to-
gether with their name, including specific functionality developed in the form of
new ARB extensions together with OpenGL 3.0. In addition, many people partic-
ipated 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 3.2 (Core Profile) - July 24, 2009
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; deprecation model)
John Rosasco, Apple
Jon Leech, Independent (Chair, ARB Ecosystem TSG; OpenGL API Specification 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
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 Freder-
icks, 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 2.19 and table 6.37, to match GLSL (bug 4803).

- Accept null pointers in CompressedTexImage* (section 3.8.3) and treat them the same as for the corresponding TexImage* commands (bug 4863).
G.4. CREDITS AND ACKNOWLEDGEMENTS

- Relax error conditions when specifying RGTC format texture images (section 3.8.2) and subimages (section 3.8.3) 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 3.8.10 (bug 4392).

- Clarify initial value of stencil value masks in section 4.1.4 and table 6.18 (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 6.23 to **NONE** (bug 4407).

- Brought extension list in appendix I.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.
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
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

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

Changes in the specification for public release on August 3, 2009:

- Public release of OpenGL 3.2. Add release date.

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
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 Freder-
icks, and Michelle Clark, and technical support from James Riordon, webmaster
of Khronos.org and OpenGL.org.
Appendix I

Extension Registry, Header Files, and ARB Extensions

I.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 GL 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 Specification. The Registry is available on the World Wide Web at URL

http://www.opengl.org/registry/

I.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 I.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 3.2, 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 OpenGL 3.2 compatibility profile (see appendices H 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.

### I.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.

#### I.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 6.1.4.

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.

I.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.

GL 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.

I.3.3 Multitexture

The name string for multitexture is GL_ARB_multitexture. It was promoted to a core feature in OpenGL 1.3.

I.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.
I.3.5 Multisample

The name string for multisample is GL_ARB_multisample. It was promoted to a core feature in OpenGL 1.3.

I.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.

I.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.

I.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.

I.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.

I.3.10 Point Parameters

The name string for point parameters is GL_ARB_point_parameters. It was promoted to a core feature in OpenGL 1.4.

I.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.
I.3. 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.

I.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.

I.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.

I.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.

I.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.

I.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.

I.3.18 Shadow

The name string for shadow is GL_ARB_shadow. It was promoted to a core feature in OpenGL 1.4.

I.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.

I.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.

I.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.

I.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.

I.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.

I.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.

I.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.
I.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.

I.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.

I.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.

I.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.

I.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.

I.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.

I.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.
I.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.

I.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.

I.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.
I.3. **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.

I.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.

I.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.

I.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.

I.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.

I.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 surface 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.

I.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.

I.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.

I.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 instance rendering is GL_ARB_instanced_arrays.

I.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.

I.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.
I.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.

I.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 equivalent to new core functionality introduced in OpenGL 3.0, and is provided to enable this functionality in older drivers.

I.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.

I.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.

I.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.

I.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 depreca-
tion 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 cor-
responding 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 $\text{GL}_{-}
\text{ARB_compatibility}$.

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.

I.3.53 Fast Buffer-to-Buffer Copies

The name string for fast buffer-to-buffer copies is $\text{GL_ARB_copy_buffer}$. This
extension is equivalent to new core functionality introduced in OpenGL 3.1 and is
provided to enable this functionality in older drivers.

I.3.54 Shader Texture Level of Detail Control

The name string for shader texture level of detail control is $\text{GL_ARB_shader_-
texture_lod}$. 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.

I.3.55 Depth Clamp Control

The name string for depth clamp control is $\text{GL_ARB_depth_clamp}$. This exten-
sion is equivalent to new core functionality introduced in OpenGL 3.2 and is pro-
vided to enable this functionality in older drivers.

I.3.56 Base Vertex Offset Drawing Commands

The name string for base vertex offset drawing commands is $\text{GL_ARB_draw_-
elements_base_vertex}$. This extension is equivalent to new core functionality
introduced in OpenGL 3.2 and is provided to enable this functionality in older
drivers.
I.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.

I.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.

I.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.

I.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.

I.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.

I.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.

I.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.

I.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.

I.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.

I.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.

I.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.

I.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.
# Index

*BaseVertex, 30  
*CopyBufferSubData, 43  
*GetString, 254  
*GetStringi, 254  
*MapBuffer, 41  
*MapBufferRange, 39  
*Pointer, 46  
*WaitSync, 257  
-292–294  
--312  
312

Accum, 333  
ACCUM_*_BITS, 333  
ACCUM_BUFFER_BIT, 333  
ACTIVE_ATTRIBUTE_MAX_LENGTH, 53, 260, 294  
ACTIVE_ATTRIBUTES, 52, 260, 293  
ACTIVE_TEXTURE, 131, 176, 249, 282  
ACTIVE_UNIFORM_BLOCK_MAX_NAME_LENGTH, 261, 295  
ACTIVE_UNIFORM_BLOCKS, 57, 261, 295  
ACTIVE_UNIFORM_MAX_LENGTH, 59, 60, 260, 293  
ACTIVE_UNIFORMS, 59, 60, 260, 293  
ActiveTexture, 71, 131  
ALIASED_LINE_WIDTH_RANGE, 304  
ALL_ATTRIB_BITS, 334  
ALPHA, 196, 212, 216, 281, 288, 332  
ALPHA_BITS, 333  
ALPHA_TEST, 333  
AlphaFunc, 333  
ALREADY_SIGNALED, 244  
ALWAYS, 159, 177, 190, 191, 283  
AND, 199  
AND_INVERTED, 199  
AND_REVERSE, 199  
Antialiasing, 111  
AreTexturesResident, 333  
ARRAY_BUFFER, 35, 44, 47  
ARRAY_BUFFER_BINDING, 44, 271  
ArrayElement, 337  
ATTACHED_SHADERS, 260, 261, 293  
AttachShader, 49  
ATTRIB_STACK_DEPTH, 334  
AUXi, 333  
BACK, 112, 190, 193, 201, 202, 204–206, 208, 212, 218, 276, 332  
BACK_LEFT, 202, 203, 264  
BACK_RIGHT, 202, 203, 264  
Begin, 330  
BeginConditionalRender, 92  
BeginQuery, 90, 91, 95, 192, 337  
BeginTransformFeedback, 93–95  
BGR, 121, 212, 216  
BGR_INTEGER, 121  
BGRA, 28, 29, 121, 124, 129, 212  
BGRA_INTEGER, 121

366
INDEX 367

BindAttribLocation, 53, 54
BindBuffer, 34–36, 45, 157
BindBufferBase, 36, 37, 69, 94, 95
BindBufferRange, 36, 37, 69, 70, 94, 95
BindFragDataLocation, 183
BindFramebuffer, 219–221, 236
BindRenderbuffer, 223, 224
BindTexture, 71, 131, 174, 175
BindVertexArray, 46
BITMAP, 332
Bitmap, 332
BLEND, 193, 197, 198, 284
BLEND_COLOR, 284
BLEND_DST_ALPHA, 284
BLEND_DST_RGB
  (v1.3: BLEND_DST), 284
BLEND_EQUATION_ALPHA, 284
BLEND_EQUATION_RGB
  (v1.5: BLEND_EQUATION), 284
BLEND_SRC_ALPHA, 284
BLEND_SRC_RGB
  (v1.3: BLEND_SRC), 284
BlendColor, 197
BlendEquation, 193, 194
BlendEquationSeparate, 194
BlendFunc, 195
BlendFuncSeparate, 195
BlitFramebuffer, 209, 216, 218, 232
BLUE, 121, 212, 216, 281, 288
BLUE_BITS, 333
BLUE_INTEGER, 121
BOOL, 61
bool, 61, 67
BOOL_VEC2, 61
BOOL_VEC3, 61
BOOL_VEC4, 61
BUFFER_ACCESS, 36, 38, 41, 272
BUFFER_ACCESS_FLAGS, 36, 38, 41,
  43, 272, 338
BUFFER_MAP_LENGTH, 36, 38, 41,
  43, 272, 338
BUFFER_MAP_OFFSET, 36, 38, 41,
  43, 272, 338
BUFFER_MAP_POINTER, 36, 38, 41,
  43, 258, 272
BUFFER_MAPPED, 36, 38, 41, 43, 272
BUFFER_SIZE, 36–39, 41, 42, 272
BUFFER_USAGE, 36, 38, 40, 272
BufferData, 37, 38, 55, 325
BufferSubData, 38, 55, 325
bvec2, 61, 65
bvec3, 61
bvec4, 61
BYTE, 28, 120, 214, 215

CallList, 333
CallLists, 333
CCW, 112, 276
CHANGED ITEMS, 3, 14–16, 23, 25,
  26, 28–30, 32–37, 42, 43, 45,
  47, 50, 55, 58, 60, 62, 63,
  66, 69–78, 81–88, 92–97, 100,
  103–105, 110, 111, 113, 117,
  122, 130, 141, 154, 156, 161,
  171–175, 180–182, 188, 189,
  192, 213, 217, 221, 222, 224,
  225, 227–230, 234, 235, 238–
  245, 247, 248, 250, 254, 256–
  261, 266, 269, 278, 279, 282,
  292, 297, 305–309, 311, 318,
  321, 323–329, 347–349, 352,
  362, 364

CheckFramebufferStatus, 236, 237
CLAMP, 332
CLAMP_FRAGMENT_COLOR, 331
CLAMP_READ_COLOR, 213, 274
CLAMP_TO_BORDER, 159, 161, 165,
  339
CLAMP_TO_EDGE, 159, 161, 165,
INDEX

173, 217
CLAMP_VERTEX_COLOR, 331
ClampColor, 213, 331
CLEAR, 199
ClearAccum, 333
ClearBuffer, 208, 209
ClearBuffer*, 92, 101, 240, 337, 338
ClearBuffer{if ui}v, 207, 208
ClearBufferfi, 208
ClearBufferfv, 208
ClearBufferiv, 208
ClearBufferuiv, 208
ClearColor, 207, 208
ClearDepth, 207, 208
ClearStencil, 207, 208
CLIENT_ALL_ATTRIB_BITS, 334
CLIENT_ATTRIB_STACK_DEPTH, 334
ClientActiveTexture, 330
ClientWaitSync, 242–245, 324
CLIP_DISTANCE, 97, 273, 337
CLIP_DISTANCE0, 97
CLIP_PLANE0i, 337
ClipPlane, 331
COLOR, 145, 207, 208
Color*, 330
COLOR_ATTACHMENT0i, 201, 202, 212, 227, 234
COLOR_ATTACHMENT0m, 201, 204
COLOR_ATTACHMENT0n, 221
COLOR_ATTACHMENT0TO, 201, 204, 212, 221
COLOR_BUFFER_BIT, 206, 209, 216, 217
COLOR_CLEAR_VALUE, 285
COLOR_INDEX, 330
COLOR_LOGIC_OP, 198, 284
COLOR_MATERIAL, 331
COLOR_SUM, 333
COLOR_WRITEMASK, 205, 285
ColorMask, 205, 206
ColorMaski, 205
ColorMaterial, 331
ColorPointer, 330
COMPARE_R_TO_TEXTURE, 337
COMPARE_REF_TO_TEXTURE, 159, 176, 337
COMPILING_STATUS, 48, 260, 292
CompileShader, 48, 49
COMPRESSED_RED, 140
COMPRESSED_RED_RGTC1, 135, 140, 320, 321
COMPRESSED_RG, 140
COMPRESSED_RG_RGTC2, 135, 140, 321
COMPRESSED_RGB, 140
COMPRESSED_RGBA, 140
COMPRESSED_SIGNED_RED_RGTC1, 135, 140, 321, 322
COMPRESSED_SIGNED_RG_RGTC2, 135, 140, 322
COMPRESSED_SRGB, 140, 177
COMPRESSED_SRGB_ALPHA, 140, 177
COMPRESSED_TEXTURE_FORMATS, 134, 304
CompressedTexImage, 153
CompressedTexImage2D, 151
CompressedTexImage*, 236, 343
CompressedTexImage1D, 150–153
CompressedTexImage2D, 150–153
CompressedTexImage3D, 150–153
CompressedTexSubImage2D, 152–154
CompressedTexSubImage1D, 152–154
CONDITION_SATISFIED, 244
CONSTANT_ALPHA, 196
<table>
<thead>
<tr>
<th>INDEX</th>
<th>369</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONSTANT_COLOR, 196</td>
<td></td>
</tr>
<tr>
<td>CONTEXT_COMPATIBILITY_PROFILE_BIT, 255</td>
<td></td>
</tr>
<tr>
<td>CONTEXT_CORE_PROFILE_BIT, 255</td>
<td></td>
</tr>
<tr>
<td>CONTEXT_FLAG_FORWARD_COMPATIBLE_BIT, 254</td>
<td></td>
</tr>
<tr>
<td>CONTEXT_FLAGS, 254, 305</td>
<td></td>
</tr>
<tr>
<td>CONTEXT_PROFILE_MASK, 254, 255</td>
<td></td>
</tr>
<tr>
<td>COPY, 199, 284</td>
<td></td>
</tr>
<tr>
<td>COPY_INVERTED, 199</td>
<td></td>
</tr>
<tr>
<td>COPY_READ_BUFFER, 35, 44, 312</td>
<td></td>
</tr>
<tr>
<td>COPY_WRITE_BUFFER, 35, 44, 312</td>
<td></td>
</tr>
<tr>
<td>CopyPixels, 333</td>
<td></td>
</tr>
<tr>
<td>CopyTexImage, 237, 333</td>
<td></td>
</tr>
<tr>
<td>CopyTexImage*, 227, 232, 236</td>
<td></td>
</tr>
<tr>
<td>CopyTexImage1D, 145–147, 150, 168</td>
<td></td>
</tr>
<tr>
<td>CopyTexImage2D, 143, 145–147, 150, 168</td>
<td></td>
</tr>
<tr>
<td>CopyTexImage3D, 147</td>
<td></td>
</tr>
<tr>
<td>CopyTexSubImage, 237</td>
<td></td>
</tr>
<tr>
<td>CopyTexSubImage*, 149, 154, 227</td>
<td></td>
</tr>
<tr>
<td>CopyTexSubImage1D, 146–150</td>
<td></td>
</tr>
<tr>
<td>CopyTexSubImage2D, 146–150</td>
<td></td>
</tr>
<tr>
<td>CopyTexSubImage3D, 146, 147, 149, 150</td>
<td></td>
</tr>
<tr>
<td>CreateProgram, 49</td>
<td></td>
</tr>
<tr>
<td>CreateShader, 48</td>
<td></td>
</tr>
<tr>
<td>CULL_FACE, 112, 276</td>
<td></td>
</tr>
<tr>
<td>CULL_FACE_MODE, 276</td>
<td></td>
</tr>
<tr>
<td>CullFace, 112, 116</td>
<td></td>
</tr>
<tr>
<td>CURRENT_PROGRAM, 293</td>
<td></td>
</tr>
<tr>
<td>CURRENT_QUERY, 255, 312</td>
<td></td>
</tr>
<tr>
<td>CURRENT_VERTEX_ATTRIB, 263, 297</td>
<td></td>
</tr>
<tr>
<td>CW, 112</td>
<td></td>
</tr>
<tr>
<td>DECR, 190</td>
<td></td>
</tr>
<tr>
<td>DECR_WRAP, 190</td>
<td></td>
</tr>
<tr>
<td>DELETE_STATUS, 49, 260, 292, 293</td>
<td></td>
</tr>
<tr>
<td>DeleteBuffers, 34, 35, 323</td>
<td></td>
</tr>
<tr>
<td>DeleteFramebuffers, 220, 221</td>
<td></td>
</tr>
<tr>
<td>DeleteLists, 333</td>
<td></td>
</tr>
<tr>
<td>DeleteProgram, 51</td>
<td></td>
</tr>
<tr>
<td>DeleteQueries, 90, 91</td>
<td></td>
</tr>
<tr>
<td>DeleteRenderbuffers, 223, 236, 323</td>
<td></td>
</tr>
<tr>
<td>DeleteShader, 49</td>
<td></td>
</tr>
<tr>
<td>DeleteSync, 243, 257</td>
<td></td>
</tr>
<tr>
<td>DeleteTextures, 174, 175, 236, 323</td>
<td></td>
</tr>
<tr>
<td>DeleteVertexArrays, 46</td>
<td></td>
</tr>
<tr>
<td>DEPTH, 145, 208, 264, 281, 337</td>
<td></td>
</tr>
<tr>
<td>DEPTH24_STENCIL8, 135, 139</td>
<td></td>
</tr>
<tr>
<td>DEPTH32F_STENCIL8, 135, 139</td>
<td></td>
</tr>
<tr>
<td>DEPTH_ATTACHMENT, 221, 227, 234, 337</td>
<td></td>
</tr>
<tr>
<td>DEPTH_BITS, 333</td>
<td></td>
</tr>
<tr>
<td>DEPTH_BUFFER, 338</td>
<td></td>
</tr>
<tr>
<td>DEPTH_BUFFER_BIT, 206, 209, 216–218</td>
<td></td>
</tr>
<tr>
<td>DEPTH_CLAMP, 98, 273</td>
<td></td>
</tr>
<tr>
<td>DEPTH_CLEAR_VALUE, 285</td>
<td></td>
</tr>
<tr>
<td>DEPTH_COMPONENT, 77, 121, 133, 139, 176, 180, 211, 213, 233, 251, 252</td>
<td></td>
</tr>
<tr>
<td>DEPTH_COMPONENT16, 135, 139</td>
<td></td>
</tr>
<tr>
<td>DEPTH_COMPONENT24, 135, 139</td>
<td></td>
</tr>
<tr>
<td>DEPTH_COMPONENT32, 139</td>
<td></td>
</tr>
<tr>
<td>DEPTH_COMPONENT32F, 135, 139</td>
<td></td>
</tr>
<tr>
<td>DEPTH_FUNC, 284</td>
<td></td>
</tr>
<tr>
<td>DEPTH_RANGE, 273</td>
<td></td>
</tr>
<tr>
<td>DEPTH_STENCIL, 77, 119, 121, 124, 129, 130, 132, 133, 139, 145, 171, 176, 180, 208, 211, 213, 226, 230, 233, 251, 252</td>
<td></td>
</tr>
<tr>
<td>DEPTH_STENCIL_ATTACHMENT, 226, 227, 230, 264</td>
<td></td>
</tr>
<tr>
<td>DEPTH_TEST, 191, 284</td>
<td></td>
</tr>
<tr>
<td>DEPTH_TEXTURE_MODE, 332</td>
<td></td>
</tr>
<tr>
<td>DEPTH_WRITEMASK, 285</td>
<td></td>
</tr>
<tr>
<td>DepthFunc, 191</td>
<td></td>
</tr>
</tbody>
</table>
INDEX

DepthMask, 205, 206
DepthRange, 89, 248
DetachShader, 50
dFdx, 246
dFdy, 246
DisableClientState, 330
Disablei, 193
DisableVertexAttribArray, 29, 263
DITHER, 198, 284
DONT_CARE, 246, 301
DOUBLE, 28
DOUBLEBUFFER, 311
DRAW_BUFFER, 201, 205, 212
DRAW_BUFFERi, 193, 205, 207, 235, 287
DRAW_BUFFER0, 205
DRAW_FRAMEBUFFER, 219–221, 226–228, 237, 264, 286
DRAW_FRAMEBUFFER_BINDING, 167, 201, 203, 218, 222, 237, 238, 286
DrawArrays, 19, 21, 29–32, 44–46, 78, 93, 237
DrawArraysInstanced, 32
DrawBuffer, 200–204, 206, 209
DrawBuffers, 200, 201, 203, 204
DrawElements, 29, 31–33, 45, 46
DrawElementsBaseVertex, 33, 34, 45
DrawElementsInstanced, 32, 45
DrawElementsInstancedBaseVertex, 33, 45
DrawPixels, 332
DrawRangeElements, 31, 33, 45, 304
DrawRangeElementsBaseVertex, 33, 45
DST_ALPHA, 196
DST_COLOR, 196
DYNAMIC_COPY, 36, 38
DYNAMIC_DRAW, 36, 38
DYNAMIC_READ, 36, 38
EdgeFlag*, 330
EdgeFlagPointer, 330
ELEMENT_ARRAY_BUFFER, 35, 45
ELEMENT_ARRAY_BUFFER_BINDING, 270
EnableClientState, 330
Enablei, 193
EnableVertexAttribArray, 29, 46, 263
DITHER, 198, 284
DONT_CARE, 246, 301
DOUBLE, 28
DOUBLEBUFFER, 311
DRAW_BUFFER, 201, 205, 212
DRAW_BUFFERi, 193, 205, 207, 235, 287
DRAW_BUFFER0, 205
DRAW_FRAMEBUFFER, 219–221, 226–228, 237, 264, 286
DRAW_FRAMEBUFFER_BINDING, 167, 201, 203, 218, 222, 237, 238, 286
DrawArrays, 19, 21, 29–32, 44–46, 78, 93, 237
DrawArraysInstanced, 32
DrawBuffer, 200–204, 206, 209
DrawBuffers, 200, 201, 203, 204
DrawElements, 29, 31–33, 45, 46
DrawElementsBaseVertex, 33, 34, 45
DrawElementsInstanced, 32, 45
DrawElementsInstancedBaseVertex, 33, 45
DrawPixels, 332
DrawRangeElements, 31, 33, 45, 304
DrawRangeElementsBaseVertex, 33, 45
DST_ALPHA, 196
DST_COLOR, 196
DYNAMIC_COPY, 36, 38
DYNAMIC_DRAW, 36, 38
DYNAMIC_READ, 36, 38
EdgeFlag*, 330
EdgeFlagPointer, 330
ELEMENT_ARRAY_BUFFER, 35, 45
ELEMENT_ARRAY_BUFFER_BINDING, 270
EnableClientState, 330
Enablei, 193
EnableVertexAttribArray, 29, 46, 263
End, 330
EndConditionalRender, 92
EndList, 333
EndPrimitive, 240
EndQuery, 91, 192
EndTransformFeedback, 93, 326, 327
EQUAL, 159, 177, 190, 191
EQUIV, 199
EvalCoord*, 333
EvalMesh*, 333
EvalPoint*, 333
EXTENSIONS, 254, 305, 334, 353
FASTEST, 246
FeedbackBuffer, 333
FenceSync, 242, 243, 246, 325
FILL, 114–117, 276, 317
Finish, 241, 242, 317, 325
FIRST_VERTEX_CONVENTION, 96
OpenGL 3.2 (Core Profile) - July 24, 2009
<table>
<thead>
<tr>
<th>Term</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIXED ONLY</td>
<td>213, 219, 274</td>
</tr>
<tr>
<td>flat</td>
<td>84, 96</td>
</tr>
<tr>
<td>FLOAT</td>
<td>28, 34, 53, 61, 119, 120, 133, 212, 213, 215, 250, 265, 269</td>
</tr>
<tr>
<td>float</td>
<td>51, 61, 67</td>
</tr>
<tr>
<td>FLOAT 32_UNSIGNED_INT_-24_8_REV</td>
<td>119, 120, 122–124, 128, 211, 214, 215</td>
</tr>
<tr>
<td>FLOAT_MAT2</td>
<td>53, 61</td>
</tr>
<tr>
<td>FLOAT_MAT2x3</td>
<td>53, 61</td>
</tr>
<tr>
<td>FLOAT_MAT2x4</td>
<td>53, 61</td>
</tr>
<tr>
<td>FLOAT_MAT3</td>
<td>53, 61</td>
</tr>
<tr>
<td>FLOAT_MAT3x2</td>
<td>53, 61</td>
</tr>
<tr>
<td>FLOAT_MAT3x4</td>
<td>53, 61</td>
</tr>
<tr>
<td>FLOAT_MAT4</td>
<td>53, 61</td>
</tr>
<tr>
<td>FLOAT_MAT4x2</td>
<td>53, 61</td>
</tr>
<tr>
<td>FLOAT_MAT4x3</td>
<td>53, 61</td>
</tr>
<tr>
<td>FLOAT_VEC2</td>
<td>53, 61</td>
</tr>
<tr>
<td>FLOAT_VEC3</td>
<td>53, 61</td>
</tr>
<tr>
<td>FLOAT_VEC4</td>
<td>53, 61</td>
</tr>
<tr>
<td>Flush</td>
<td>241, 245, 317</td>
</tr>
<tr>
<td>FlushMappedBufferRange</td>
<td>40, 42, 325</td>
</tr>
<tr>
<td>FOG</td>
<td>333</td>
</tr>
<tr>
<td>Fog</td>
<td>333</td>
</tr>
<tr>
<td>FOG_HINT</td>
<td>334</td>
</tr>
<tr>
<td>FogCoord*</td>
<td>330</td>
</tr>
<tr>
<td>FRAGMENT_SHADER</td>
<td>178, 260</td>
</tr>
<tr>
<td>FRAGMENT_SHADER_DERIVATIVE_HINT</td>
<td>246, 301</td>
</tr>
<tr>
<td>FRAMEBUFFER</td>
<td>220, 226–228, 237, 264</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_-x_SIZE</td>
<td>288</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_-ALPHA_SIZE</td>
<td>265</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_-BLUE_SIZE</td>
<td>265</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_-COLOR_ENCODING</td>
<td>194, 265, 288</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_-COMPONENT_TYPE</td>
<td>265, 288</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_-DEPTH_SIZE</td>
<td>265</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_-GREEN_SIZE</td>
<td>265</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_-LAYERED</td>
<td>230, 266, 288</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_-OBJECT_NAME</td>
<td>226, 230, 234, 235, 238, 265, 266, 288</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_-OBJECT_TYPE</td>
<td>226, 230, 233–235, 238, 265, 266, 288</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_-RED_SIZE</td>
<td>265</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_-STENCIL_SIZE</td>
<td>265</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_-TEXTURE_-CUBE_MAP_FACE</td>
<td>230, 266, 288</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_-TEXTURE_LAYER</td>
<td>229, 230, 234, 239, 266, 288</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_-TEXTURE_LEVEL</td>
<td>167, 230–232, 266, 288</td>
</tr>
<tr>
<td>FRAMEBUFFER_BINDING</td>
<td>222</td>
</tr>
<tr>
<td>FRAMEBUFFER_COMPLETE</td>
<td>237</td>
</tr>
<tr>
<td>FRAMEBUFFER_DEFAULT</td>
<td>265</td>
</tr>
<tr>
<td>FRAMEBUFFER_INCOMPLETE_ATTACHMENT</td>
<td>234</td>
</tr>
<tr>
<td>FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER</td>
<td>235</td>
</tr>
</tbody>
</table>

OpenGL 3.2 (Core Profile) - July 24, 2009
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAYER_TARGETS</td>
<td>235</td>
</tr>
<tr>
<td>FRAMEBUFFER_INCOMPLETE_-MISSING_ATTACHMENT</td>
<td>235</td>
</tr>
<tr>
<td>FRAMEBUFFER_INCOMPLETE_-MULTISAMPLE</td>
<td>235</td>
</tr>
<tr>
<td>FRAMEBUFFER_INCOMPLETE_-READ_BUFFER</td>
<td>235</td>
</tr>
<tr>
<td>FRAMEBUFFER_SRGB</td>
<td>194, 197, 284</td>
</tr>
<tr>
<td>FRAMEBUFFER_UNDEFINED</td>
<td>234</td>
</tr>
<tr>
<td>FRAMEBUFFER_UNSUPPORTED</td>
<td>235, 236</td>
</tr>
<tr>
<td>FramebufferRenderbuffer</td>
<td>226, 236</td>
</tr>
<tr>
<td>FramebufferTexture</td>
<td>227, 229, 230</td>
</tr>
<tr>
<td>FramebufferTexture*</td>
<td>229, 230, 236</td>
</tr>
<tr>
<td>FramebufferTexture1D</td>
<td>228, 229</td>
</tr>
<tr>
<td>FramebufferTexture2D</td>
<td>228–230</td>
</tr>
<tr>
<td>FramebufferTexture3D</td>
<td>228–230</td>
</tr>
<tr>
<td>FramebufferTextureLayer</td>
<td>229, 230, 344</td>
</tr>
<tr>
<td>FRONT, 112, 190, 193, 201, 202, 204–206, 208, 212, 218, 332</td>
<td></td>
</tr>
<tr>
<td>FRONT_AND_BACK, 112, 114, 190, 193, 202, 204–206, 208, 212</td>
<td></td>
</tr>
<tr>
<td>FRONT_FACE</td>
<td>276</td>
</tr>
<tr>
<td>FRONT_LEFT, 202, 203, 264</td>
<td></td>
</tr>
<tr>
<td>FRONT_RIGHT, 202, 203, 264</td>
<td></td>
</tr>
<tr>
<td>FrontFace, 112, 181, 331</td>
<td></td>
</tr>
<tr>
<td>Frustum, 330</td>
<td></td>
</tr>
<tr>
<td>FUNC_ADD, 194, 195, 197, 284</td>
<td></td>
</tr>
<tr>
<td>FUNC_REVERSE_SUBTRACT, 194, 195</td>
<td></td>
</tr>
<tr>
<td>FUNC_SUBTRACT</td>
<td>194, 195</td>
</tr>
<tr>
<td>fwidth, 246</td>
<td></td>
</tr>
<tr>
<td>Gen*, 324, 330</td>
<td></td>
</tr>
<tr>
<td>GenBuffers, 34, 35</td>
<td></td>
</tr>
<tr>
<td>GENERATE_MIPMAP</td>
<td>333</td>
</tr>
<tr>
<td>GENERATE_MIPMAP_HINT</td>
<td>334</td>
</tr>
<tr>
<td>GenerateMipmap</td>
<td>170</td>
</tr>
<tr>
<td>GenFramebuffers</td>
<td>219–222</td>
</tr>
<tr>
<td>GenLists, 333</td>
<td></td>
</tr>
<tr>
<td>GenQueries, 90, 91</td>
<td></td>
</tr>
<tr>
<td>GenRenderbuffers</td>
<td>223, 224</td>
</tr>
<tr>
<td>GenTextures, 174, 175, 253</td>
<td></td>
</tr>
<tr>
<td>GenVertexArrays</td>
<td>46</td>
</tr>
<tr>
<td>GEOMETRY_INPUT_TYPE, 82, 261, 294</td>
<td></td>
</tr>
<tr>
<td>GEOMETRY_OUTPUT_TYPE, 83, 261, 294</td>
<td></td>
</tr>
<tr>
<td>GEOMETRY_SHADER, 81, 260</td>
<td></td>
</tr>
<tr>
<td>GEOMETRY_VERTICES_OUT, 83, 86, 261, 294</td>
<td></td>
</tr>
<tr>
<td>GEQUAL, 159, 177, 190, 191</td>
<td></td>
</tr>
<tr>
<td>Get, 89, 247, 248</td>
<td></td>
</tr>
<tr>
<td>GetActiveUniformBlockiv, 296</td>
<td></td>
</tr>
<tr>
<td>GetActiveAttrib, 52, 53, 74, 294</td>
<td></td>
</tr>
<tr>
<td>GetActiveUniform, 59–61, 63, 65, 293</td>
<td></td>
</tr>
<tr>
<td>GetActiveUniformBlockiv, 57</td>
<td></td>
</tr>
<tr>
<td>GetActiveUniformBlockName, 56</td>
<td></td>
</tr>
<tr>
<td>GetActiveUniformName, 59</td>
<td></td>
</tr>
<tr>
<td>GetAttachedShaders, 261, 293</td>
<td></td>
</tr>
<tr>
<td>GetAttribLocation, 53, 54, 294</td>
<td></td>
</tr>
<tr>
<td>GetBooleani, 205, 247, 285</td>
<td></td>
</tr>
<tr>
<td>GetBooleanv, 189, 205, 247, 248, 267, 277, 285, 291, 310, 311</td>
<td></td>
</tr>
<tr>
<td>GetBufferParameteri64v, 258, 272</td>
<td></td>
</tr>
<tr>
<td>GetBufferParameteriv, 257, 272, 338</td>
<td></td>
</tr>
<tr>
<td>GetBufferPointerv, 258, 272</td>
<td></td>
</tr>
<tr>
<td>GetBufferSubData, 258, 272</td>
<td></td>
</tr>
<tr>
<td>GetCompressedTexImage, 151, 153, 246, 251, 253</td>
<td></td>
</tr>
<tr>
<td>GetDoublev, 247, 248, 267</td>
<td></td>
</tr>
<tr>
<td>GetError, 18, 312</td>
<td></td>
</tr>
<tr>
<td>GetFloatv, 13, 189, 247, 248, 267, 273, 275–277, 284, 285, 302–304</td>
<td></td>
</tr>
<tr>
<td>GetFragDataLocation, 183, 184</td>
<td></td>
</tr>
</tbody>
</table>
INDEX

GL_EXT_texture_compression_rgtc, 336, 361
GL_EXT_texture_integer, 336
GL_EXT_texture_shared_exponent, 336
GL_EXT_transform_feedback, 336
gl_FragColor, 182, 183, 204
gl_FragCoord, 181
gl_FragCoord.z, 315
gl_FragData, 183, 204
gl_FragData[n], 182
gl_FragDepth, 182, 183, 315
gl_FrontFacing, 181
gl_in[], 85
gl_InstanceID, 32, 77, 78
gl_Layer, 87, 240
GL_NV_conditional_render, 335
GL_NV_depth_buffer_float, 336, 359
GL_NV_half_float, 336, 360
GL_NV_primitive_restart, 342
gl_PointCoord, 106
gl_PointSize, 78, 85, 87, 105
gl_Position, 72, 78, 85, 87, 88, 318
gl_PrimitiveID, 87, 181, 182
gl_PrimitiveIDn, 85
gl_VertexID, 77, 78, 182
GLX_ARB_create_context, 361
GLX_ARB_create_context_profile, 329, 365
GLX_ARB_fboconfig_float, 358
GLX_ARB_framebuffer_sRGB, 360
GREATER, 159, 177, 190, 191
GREEN, 121, 212, 216, 281, 288
GREEN_BITS, 333
GREEN_INTEGER, 121
HALF_FLOAT, 28, 120, 133, 212, 213, 215
Hint, 246, 334
INCR, 190
INCR_WRAP, 190
Index*, 330
IndexPointer, 330
INFO_LOG_LENGTH, 260, 262, 292, 293
InitNames, 333
INT, 28, 53, 61, 120, 214, 215, 250, 265
int, 61, 67
INT_SAMPLER_1D, 62
INT_SAMPLER_1D_ARRAY, 62
INT_SAMPLER_2D, 62
INT_SAMPLER_2D_ARRAY, 62
INT_SAMPLER_2D_MULTISAMPLE, 62
INT_SAMPLER_2D_MULTISAMPLE_ARRAY, 62
INT_SAMPLER_2D_RECT, 62
INT_SAMPLER_3D, 62
INT_SAMPLER_BUFFER, 62
INT_SAMPLER_CUBE, 62
INT_VEC2, 53, 61
INT_VEC3, 53, 61
INT_VEC4, 53, 61
INTENSITY, 332
INTERLEAVED_ATTRIBS, 72, 73, 94, 261, 294
InterleavedArrays, 330
INVALID_ENUM, 18, 19, 42, 119, 131, 145, 151, 152, 154, 157, 160, 174, 201, 203, 208, 211, 242, 252, 257, 266, 267, 339
INVALID_FRAMEBUFFER_OPERATION, 19, 145, 150, 211, 218, 237
INVALID_INDEX, 56, 59
INDEX

183, 184, 201, 204, 209, 211, 212, 214, 217, 218, 220, 223, 224, 226–229, 249, 251–253, 256, 259, 261, 264, 265, 267, 329–331, 337, 338


INVERT, 190, 191, 283

isampler1D, 62
isampler1DArray, 62
isampler2D, 62
isampler2DArray, 62
isampler2DMS, 62
isampler2DMSArray, 62
isampler2DRect, 62
isampler3D, 62
isamplerBuffer, 62
isamplerCube, 62
IsBuffer, 257
IsEnabled, 188, 197, 248, 267, 271, 273, 275–277, 283, 284, 297, 312
IsEnabledi, 197, 248, 284
IsFramebuffer, 264
IsList, 333
IsProgram, 260
IsQuery, 255
IsRenderbuffer, 266
IsShader, 259
IsSync, 257
IsTexture, 253
IsVertexArray, 259
ivec2, 61
ivec3, 61
ivec4, 61

KEEP, 190, 191, 283

LAST_VERTEX_CONVENTION, 96, 274
Layered images, 222
layout, 68
LEFT, 193, 201, 202, 204, 205, 208, 212
LEQUAL, 159, 173, 177, 190, 191, 280
LESS, 159, 177, 190–192, 284
LIGHTi, 331
Light*, 331
LIGHTING, 331
LightModel*, 331
LINE, 114–117, 276
LINE_LOOP, 21, 82, 88, 93
LINE_SMOOTH, 107, 110, 275
LINE_SMOOTH_HINT, 246, 301
LINE_STIPPLE, 331
LINE_STRIP, 21, 82, 83, 88, 93, 261
LINE_STRIP_ADJACENCY, 24, 82, 88
LINE_WIDTH, 275
LINEAR, 75, 159–161, 166, 167, 169, 170, 173, 216–218, 231, 265, 280
LINEAR_MIPMAP_LINEAR, 159, 167–169, 232
LINEAR_MIPMAP_NEAREST, 159, 167–169, 232
LINES, 21, 82, 88, 93, 261
lines, 82
LINES_ADJACENCY, 23, 82, 88, 261
lines_adjacency, 82
LineStipple, 331
LineWidth, 107, 330, 331, 342
LINK_STATUS, 50, 260, 293
LinkProgram, 50–52, 54, 56–60, 70–73, 86, 95, 183
INDEX

ListBase, 333  
LoadIdentity, 330  
LoadMatrix, 330  
LoadName, 333  
LoadTransposeMatrix, 330  
LOGIC_OP_MODE, 284  
LogicOp, 199  
LOWER_LEFT, 105, 106  
LUMINANCE, 332  
LUMINANCE_ALPHA, 332  
MAJOR_VERSION, 254, 305  
Map*, 333  
MAP_FLUSH_EXPLICIT_BIT, 40–42  
MAP_INVALIDATE_BUFFER_BIT, 40, 41  
MAP_INVALIDATE_RANGE_BIT, 40, 41  
MAP_READ_BIT, 39–42  
MAP_UNS SYNCHRONIZED_BIT, 40, 41  
MAP_WRITE_BIT, 39–42  
MapBuffer, 36, 39, 42, 55, 95, 338  
MapBufferRange, 39–42, 338  
MapGrid*, 333  
matC, 67  
matCXR, 67  
mat2, 52, 61  
mat2x3, 52, 61  
mat2x4, 52, 61  
mat3, 52, 61  
mat3x2, 52, 61  
mat3x4, 52, 61  
mat4, 52, 61  
mat4x2, 52, 61  
mat4x3, 52, 61  
Material*, 331  
MatrixMode, 330  
MAX, 194, 195  
MAX_3D_TEXTURE_SIZE, 141, 228, 302  
MAX_ARRAY_TEXTURE_LAYERS, 141, 302  
MAX_ATTRIB_STACK_DEPTH, 334  
MAX_CLIENT_ATTRIB_STACK_DEPTH, 334  
MAX_CLIP_DISTANCES, 302, 337, 343  
MAX_CLIP_PLANES, 337  
MAX_COLOR_ATTACHMENTS, 201, 202, 204, 220, 227, 237, 311  
MAX_COLOR_TEXTURE_SAMPLES, 155, 310  
MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS, 178, 309  
MAX_COMBINED_GEOMETRY_UNIFORM_COMPONENTS, 309  
MAX_COMBINED_TEXTURE_IMAGE_UNITS, 76, 131, 249, 309  
MAX_COMBINED_UNIFORM_BLOCKS, 66, 309  
MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS, 55, 309  
MAX_CUBE_MAP_TEXTURE_SIZE, 141, 228, 302  
MAX_DEPTH_TEXTURE_SAMPLES, 155, 310  
MAX_DRAW_BUFFERS, 183, 193, 197, 203, 205, 208, 311  
MAX_ELEMENTS_INDICES, 32, 304  
MAX_ELEMENTS_VERTICES, 32, 304  
MAX_FRAGMENT_INPUT_COMPONENTS, 182, 308
<table>
<thead>
<tr>
<th><strong>INDEX</strong></th>
<th>378</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_FRAGMENT_UNIFORM_BLOCKS, 66, 308</td>
<td>MAX.TEXTURE_SIZE, 141, 155, 228, 302</td>
</tr>
<tr>
<td>MAX_FRAGMENT_UNIFORM_COMPONENTS, 178, 308</td>
<td>MAX.TEXTURE_UNITS, 333</td>
</tr>
<tr>
<td>MAX_GEOMETRY_INPUT_COMPONENTS, 86, 307</td>
<td>MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS, 73, 299</td>
</tr>
<tr>
<td>MAX_GEOMETRY_OUTPUT_COMPONENTS, 87, 307</td>
<td>MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS, 72, 94, 259, 299</td>
</tr>
<tr>
<td>MAX_GEOMETRY_OUTPUT_VERTICES, 86, 307</td>
<td>MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS, 73, 299</td>
</tr>
<tr>
<td>MAX_GEOMETRY_TEXTURE_IMAGE_UNITS, 76, 307</td>
<td>MAX_UNIFORM_BLOCK_SIZE, 58, 309</td>
</tr>
<tr>
<td>MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS, 86, 307</td>
<td>MAX_UNIFORM_BUFFER_BINDINGS, 69, 70, 258, 309</td>
</tr>
<tr>
<td>MAX_GEOMETRY_UNIFORM_BLOCKS, 66, 307</td>
<td>MAX_VARYING_COMPONENTS, 72, 309, 330, 337, 348</td>
</tr>
<tr>
<td>MAX_GEOMETRY_UNIFORM_COMPONENTS, 84</td>
<td>MAX_VARYING_FLOATS, 330, 337, 348</td>
</tr>
<tr>
<td>MAX_INTEGER_SAMPLES, 155, 224, 225, 310</td>
<td>MAX_VERTEX_ATTRIBS, 25, 27–29, 34, 52–54, 263, 306</td>
</tr>
<tr>
<td>MAX_PROGRAM_TEXEL_OFFSET, 163, 308</td>
<td>MAX_VERTEX_OUTPUT_COMPONENTS, 71, 72, 306</td>
</tr>
<tr>
<td>MAX_RECTANGLE_TEXTURE_SIZE, 141, 304</td>
<td>MAX_VERTEX_TEXTURE_IMAGE_UNITS, 76, 306</td>
</tr>
<tr>
<td>MAX_RENDERBUFFER_SIZE, 224, 302</td>
<td>MIN, 194, 195</td>
</tr>
<tr>
<td>MAX_SAMPLE_MASK_WORDS, 189, 310</td>
<td>MIN_PROGRAM_DIMS, 255, 303</td>
</tr>
<tr>
<td>MAX_SAMPLES, 155, 224, 225, 311</td>
<td>MIN_PROGRAM_TEXEL_OFFSET, 163, 308</td>
</tr>
<tr>
<td>MAX_SERVER_WAIT_TIMEOUT, 244, 310</td>
<td>MIRRORED_REPEAT, 159, 160, 165</td>
</tr>
<tr>
<td>MAX_TEXTURE_BUFFER_SIZE, 156, 304</td>
<td>MultiDrawArrays, 30</td>
</tr>
<tr>
<td>MAX_TEXTURE_COORDS, 333</td>
<td>MultiDrawElements, 31, 45</td>
</tr>
<tr>
<td>MAX_TEXTURE_IMAGE_UNITS, 76, 180, 308</td>
<td></td>
</tr>
<tr>
<td>MAX_TEXTURE_LOD_BIAS, 162, 302</td>
<td></td>
</tr>
</tbody>
</table>
INDEX

MultiDrawElementsBaseVertex, 33, 45
MULTISAMPLE, 104, 107, 110, 116, 188, 200, 277
MultMatrix, 330
MultTransposeMatrix, 330
NAND, 199
NEAREST_MIPMAP-, LINEAR, 159, 167–170, 173, 232
NEAREST_MIPMAP_NEAREST, 159, 167–171, 176, 232
NEVER, 159, 177, 190, 191
NewList, 333
NICEST, 246
NO_ERROR, 18
NOOP, 199
noperspective, 99
NOR, 199
Normal3*, 330
NORMALIZE, 331
NormalPointer, 330
NOTEQUAL, 159, 177, 190, 191
NULL, 34, 36, 38, 41, 45, 47, 48, 52, 57, 59, 60, 73, 257, 258, 261, 262, 268, 269, 272
NUM_COMPRESSED_TEXTURE_-FORMATS, 134, 304
NUM_EXTENSIONS, 255, 305
NV, 343
OBJECT_TYPE, 243, 257, 300
ONE, 196, 197, 284
ONE_MINUS_CONSTANT_ALPHA, 196
ONE_MINUS_CONSTANT_COLOR, 196
ONE_MINUS_DST_ALPHA, 196
ONE_MINUS_DST_COLOR, 196
ONE_MINUS_SRC_ALPHA, 196
ONE_MINUS_SRC_COLOR, 196
OR, 199
OR_INVERTED, 199
OR_REVERSE, 199
Ortho, 330
OUT_OF_MEMORY, 18, 19, 38, 41, 155, 224
PACK_ALIGNMENT, 210, 291
PACK_IMAGE_HEIGHT, 210, 252, 291
PACK_LSB_FIRST, 210, 291
PACK_ROW_LENGTH, 210, 291
PACK_SKIP_IMAGES, 210, 252, 291
PACK_SKIP_PIXELS, 210, 291
PACK_SKIP_ROWS, 210, 291
PACK_SWAP_BYTES, 210, 291
PassThrough, 333
PERSPECTIVE_CORRECTION_-HINT, 334
PIXEL_PACK_BUFFER, 35, 118, 209
PIXEL_PACK_BUFFER_BINDING, 214, 252, 291
PIXEL_UNPACK_BUFFER, 35, 118
PIXEL_UNPACK_BUFFER_BINDING, 121, 150, 291
PixelStore, 117, 118, 210, 218, 219
PixelZoom, 332
POINT, 114–117, 276
POINT_FADE_THRESHOLD_SIZE, 105, 275
POINT_SIZE, 275
POINT_SIZE_GRANULARITY, 303
INDEX 380

POINT_SIZE_RANGE, 303
POINT_SMOOTH, 331
POINT_SMOOTH_HINT, 334
POINT_SPRITE, 331
POINT_SPRITE_COORD_ORIGIN, 105, 106, 275
PointParameter, 105
PointParameter*, 105
POINTS, 21, 82, 83, 88, 93, 114, 261
points, 82
PointSize, 105
POLYGON, 331
POLYGON_OFFSET_FACTOR, 276
POLYGON_OFFSET_FILL, 116, 276
POLYGON_OFFSET_LINE, 116, 276
POLYGON_OFFSET_POINT, 116, 276
POLYGON_OFFSET_UNITS, 276
POLYGON_SMOOTH, 111, 116, 276
POLYGON_SMOOTH_HINT, 246, 301
POLYGON_STIPPLE, 332
PolygonMode, 114, 115, 117, 332
PolygonOffset, 115
PolygonStipple, 332
PopAttrib, 334
PopClientAttrib, 334
PopMatrix, 330
PopName, 333
PRIMITIVE_RESTART, 30, 271, 342, 343
PRIMITIVE_RESTART_INDEX, 271, 343
PrimitiveRestartIndex, 30
PRIMITIVES_GENERATED, 95, 255
PrioritizeTextures, 333
PROGRAM_POINT_SIZE, 85, 105, 297, 349
PROVOKING_VERTEX, 274
ProvokingVertex, 96
PROXY_TEXTURE_1D, 133, 143, 173, 250
PROXY_TEXTURE_1D_ARRAY, 133, 142, 173, 250
PROXY_TEXTURE_2D, 133, 142, 173, 250
PROXY_TEXTURE_2D_ARRAY, 132, 133, 173, 250
PROXY_TEXTURE_2D_MULTISAMPLE, 155, 173, 250
PROXY_TEXTURE_2D_MULTISAMPLE_ARRAY, 155, 173, 250
PROXY_TEXTURE_3D, 132, 173, 250
PROXY_TEXTURE_CUBE_MAP, 133, 142, 173, 174, 250
PROXY_TEXTURE_RECTANGLE, 133, 142, 151, 152, 173, 250
PushAttrib, 334
PushClientAttrib, 334
PushMatrix, 330
PushName, 333
QUAD_STRIP, 331
QUADS, 331
QUADS_FOLLOW_PROVOKING_VERTEX, 310
QUERY_BY_REGION_NO_WAIT, 92
QUERY_BY_REGION_WAIT, 92
QUERY_COUNTER_BITS, 255, 310
QUERY_NO_WAIT, 92
QUERY_RESULT, 256, 298
QUERY_RESULT_AVAILABLE, 256, 298
QUERY_WAIT, 92
R, 336
R11F_G11F_B10F, 135, 138
R16, 135, 137, 158
R16_SNORM, 135, 137
R16F, 135, 137, 158
R16I, 135, 138, 158
R16UI, 135, 138, 158
INDEX

RGB16_SNORM, 135, 137
RGB16F, 135, 137
RGB16L, 135, 138
RGB16UI, 135, 138
RGB32F, 135, 137
RGB32L, 135, 138
RGB32UI, 135, 138
RGBA, 137
RGB5, 137
RGB5_A1, 137
RGB8, 135, 137
RGB8_SNORM, 135, 137
RGB8I, 135, 138
RGB8UI, 135, 138
RGB9_E5, 135, 138, 178, 214
RGB_INTEGER, 121
RGBA, 121, 124, 129, 133, 137, 138, 140, 172, 180, 212, 233, 253, 281, 290, 332
RGB12, 137
RGB16, 135, 137, 158
RGB16_SNORM, 135, 137
RGB16F, 135, 137, 158
RGB16L, 135, 138, 158
RGB16UI, 135, 138, 158
RGB2, 137
RGB32F, 135, 138, 158
RGB32L, 135, 138, 158
RGB32UI, 135, 138, 158
RGBA, 137
RGB8, 135, 137, 158
RGB8_SNORM, 135, 137
RGB8I, 135, 138
RGB8UI, 135, 138
RGB_INTEGER, 121
RIGHT, 193, 201, 202, 204, 205, 208, 212
Rotate, 330
SAMPLE_ALPHA_TO_COVERAGE, 188, 277
SAMPLE_ALPHA_TO_ONE, 188, 189, 277
SAMPLE_BUFFERS, 103, 107, 110, 116, 145, 188, 192, 200, 206, 209, 211, 218, 237, 311
SAMPLE_COVERAGE, 188, 189, 277
SAMPLE_COVERAGE_INVERT, 188, 189, 277
SAMPLE_COVERAGE_VALUE, 188, 189, 277
SAMPLE_MASK, 188, 189, 277
SAMPLE_MASK_VALUE, 188, 189, 277
SAMPLE_POSITION, 103, 311
SampleCoverage, 189
SampleMaski, 189
sampler1D, 61
sampler1DArray, 61
sampler1DArrayShadow, 62
sampler1DShadow, 61, 77, 180
sampler2D, 61, 71
sampler2DArray, 62
sampler2DArrayShadow, 62
sampler2DMultiSample, 62
sampler2DMultiSampleArray, 62
sampler2DMultiSampleArrayShadow, 62
sampler2DRect, 62
sampler2DRectShadow, 62, 77, 180
sampler2DShadow, 61, 77, 180
sampler3D, 61
SAMPLER_1D, 61
SAMPLER_1D_ARRAY, 61
SAMPLER_1D_ARRAY_SHADOW, 62
SAMPLER_1D_SHADOW, 61
SAMPLER_2D, 61
SAMPLER_2D_ARRAY, 62
SAMPLER_2D_ARRAY_SHADOW, 62
SAMPLER_2D_MULTISAMPLE, 62
OpenGL 3.2 (Core Profile) - July 24, 2009
INDEX

SAMPLER_2D_MULTISAMPLE_ARRAY, 62
SAMPLER_2D_RECT, 62
SAMPLER_2D_RECT_SHADOW, 62
SAMPLER_2D_SHADOW, 61
SAMPLER_3D, 61
SAMPLER_BUFFER, 62
SAMPLER_CUBE, 61
SAMPLER_CUBE_SHADOW, 62
samplerBuffer, 62
samplerCube, 61
samplerCubeShadow, 62
SAMPLES, 103, 104, 192, 218, 237, 311
SAMPLES_PASSED, 92, 192, 255
Scale, 330
Scissor, 188
SCISSOR_BOX, 283
SCISSOR_TEST, 188, 283
SecondaryColor3*, 330
SecondaryColorPointer, 330
SelectBuffer, 333
SEPARATE_ATTRIBS, 72, 73, 94, 261
SET, 199
ShadeModel, 331
SHADER_SOURCE_LENGTH, 260, 262, 292
SHADER_TYPE, 80, 260, 292
ShaderSource, 48, 49, 262
SHADING_LANGUAGE_VERSION, 254, 305
SHORT, 28, 120, 214, 215
SIGNALED, 242, 257
SIGNED_NORMALIZED, 250, 265
SMOOTH_LINE_WIDTH_GRANULARITY
(v1.1: LINE_WIDTH_GRANULARITY), 304
SMOOTH_LINE_WIDTH_RANGE
(v1.1: LINE_WIDTH_RANGE), 304
SRC_ALPHA, 196
SRC_ALPHA_SATURATE, 196
SRC_COLOR, 196
SRGB, 177, 194, 197, 265
SRGB8, 135, 137, 177
SRGB8_ALPHA8, 135, 137, 177
SRGB_ALPHA, 177
STATIC_COPY, 36, 38
STATIC_DRAW, 36, 37, 272
STATIC_READ, 36, 37
std140, 57, 68
STENCIL, 208, 264, 281, 288, 337
STENCIL_ATTACHMENT, 221, 227, 234, 338
STENCIL_ATTACHMENT, 337, 338
STENCIL_BACK_FAIL, 283
STENCIL_BACK_FUNC, 283
STENCIL_BACK_PASS_DEPTH_FAIL, 283
STENCIL_BACK_PASS_DEPTH_PASS, 283
STENCIL_BACK_REF, 283
STENCIL_BACK_VALUE_MASK, 283
STENCIL_BACK_WRITEMASK, 285
STENCIL_BITS, 333
STENCIL_BUFFER, 338
STENCIL_BUFFER_BIT, 206, 209, 216–218
STENCIL_CLEAR_VALUE, 285
STENCIL_FAIL, 283
STENCIL_FUNC, 283
STENCIL_INDEX, 121, 132, 211, 213, 225, 233, 252
STENCIL_INDEX1, 225
STENCIL_INDEX16, 225
STENCIL_INDEX4, 225
STENCIL_INDEX8, 225
STENCIL_PASS_DEPTH_FAIL, 283
STENCIL_PASS_DEPTH_PASS, 283

OpenGL 3.2 (Core Profile) - July 24, 2009
INDEX

STENCIL_REF, 283
STENCIL_TEST, 190, 283
STENCIL_VALUE_MASK, 283
STENCIL_WRITE, 283
StencilFunc, 190, 191, 317
StencilFuncSeparate, 190, 191
StencilMask, 205, 206, 317
StencilMaskSeparate, 206
StencilOp, 190, 191
StencilOpSeparate, 190, 191
STEREO, 311
STREAM_COPY, 36, 37
STREAM_DRAW, 36, 37
STREAM_READ, 36, 37
SUBPIXEL_BITS, 302
SYNC_CONDITION, 243, 257, 300
SYNC_FENCE, 243, 257, 300
SYNC_FLAGS, 243, 257, 300
SYNC_FLUSH_COMMANDS_BIT, 244, 245
SYNC_GPU_COMMANDS_COMPLETE, 242, 243, 257, 300
SYNC_STATUS, 242, 243, 257, 300
TexBuffer, 156
TexCoord*, 330
TexCoordPointer, 330
TexEnv, 333
TexGen*, 331
TexImage, 131, 157, 325, 333
TexParameter, 131, 157, 325, 333
TexParameter*, 333
TexParameter[i], 162, 168
TexParameterI, 157
TexParameterIuiv, 157
TexParameterIuv, 157
TexSubImage, 147
TexSubImage*, 149, 154, 325
TexSubImage*D, 117
TexSubImage1D, 118, 146–149, 152
TexSubImage2D, 118, 146–149, 152
TexSubImage3D, 118, 146, 147, 149, 152
TEXTURE, 230, 234, 238, 265, 266
TEXTUREi, 131
TEXTURE0, 131, 282
TEXTURE_0_SIZE, 281
TEXTURE_0_TYPE, 281
TEXTURE_0, 131, 157, 170, 174, 175, 229, 249–251, 333
TEXTURE_1D_ARRAY, 133, 142, 145, 146, 157, 170, 174, 175, 249–251, 278, 279, 333
TEXTURE_2D, 71, 133, 142, 145, 146, 157, 170, 174, 175, 229, 249–251, 333
TEXTURE_2D_ARRAY, 132, 140, 146, 152, 154, 157, 170, 174, 175, 249–251, 278, 279, 333
TEXTURE_2D_MULTISAMPLE, 155, 174, 175, 228, 229, 250, 279
TEXTURE_2D_MULTISAMPLE_ARRAY, 155, 174, 175, 228, 250, 279

TexImage3DMultisample, 154, 155, 173
TexImage3D, 118, 131, 132, 139, 140, 142, 143, 147, 150, 152, 168,
INDEX

TEXTURE_3D, 132, 140, 146, 157, 170, 173–175, 228, 229, 249–251, 333
TEXTURE_ALPHA_SIZE, 250
TEXTURE_ALPHA_TYPE, 250
TEXTURE_BINDING_1D, 278
TEXTURE_BINDING_1D_ARRAY, 278
TEXTURE_BINDING_2D_ARRAY, 278
TEXTURE_BINDING_2D_MULTISAMPLE, 279
TEXTURE_BINDING_2D_MULTISAMPLE_ARRAY, 279
TEXTURE_BINDING_BUFFER, 279
TEXTURE_BINDING_CUBE_MAP, 279
TEXTURE_BINDING_RECTANGLE, 279
TEXTURE_BLUE_SIZE, 250
TEXTURE_BLUE_TYPE, 250
TEXTURE_BORDER, 151, 153, 251
TEXTURE_BORDER_COLOR, 157, 159, 165, 166, 172, 173, 249, 280, 339
TEXTURE_BUFFER, 35, 156, 157, 174, 175, 250, 279
TEXTURE_BUFFER_DATA_STORE_BINDING, 281
TEXTURE_COMPARE_FAIL_VALUE_ARB, 355
TEXTURE_COMPARE_FUNC, 159, 173, 176, 280
TEXTURE_COMPARE_MODE, 77, 159, 173, 176, 180, 280
TEXTURE_COMPONENTS, 332
TEXTURE_COMPRESSED, 281
TEXTURE_COMPRESSED_IMAGE_SIZE, 151, 153, 251, 253, 281
TEXTURE_COMPRESSION_HINT, 246, 301
TEXTURE_CUBE_MAP, 133, 142, 157, 170, 174, 175, 249, 250, 279, 333
TEXTURE_CUBE_MAP_, 142
TEXTURE_CUBE_MAP_NEGATIVE_X, 142, 145, 146, 160, 228, 229, 240, 250, 251, 279
TEXTURE_CUBE_MAP_NEGATIVE_Y, 142, 145, 146, 160, 228, 229, 240, 250, 251, 279
TEXTURE_CUBE_MAP_NEGATIVE_Z, 142, 145, 146, 160, 228, 229, 240, 250, 251, 279
TEXTURE_CUBE_MAP_POSITIVE_Y, 142, 145, 146, 160, 228, 229, 240, 250, 251, 279
TEXTURE_CUBE_MAP_POSITIVE_Z, 142, 145, 146, 160, 228, 229, 240, 250, 251, 279
TEXTURE_CUBE_MAP_SEAMLESS, 161, 312
TEXTURE_DEPTH, 151, 153, 154, 251, 281
TEXTURE_DEPTH_SIZE, 250
TEXTURE_DEPTH_TYPE, 250
TEXTURE_ENV, 333
TEXTURE_FILTER_CONTROL, 333
TEXTURE_FIXED_SAMPLE_LOCATIONS, 235, 281
TEXTURE_GEN_, 331
TEXTURE_GREEN_SIZE, 250
TEXTURE_GREEN_TYPE, 250
INDEX

TEXTURE_HEIGHT, 149, 151, 153, 154, 251, 281
TEXTURE_INTERNAL_FORMAT, 151, 153, 251, 281, 332
TEXTURE_LOD_BIAS, 159, 162, 280, 333
TEXTURE_MAG_FILTER, 159, 170, 171, 173, 176, 280
TEXTURE_MAX_LEVEL, 157, 159, 168, 173, 232, 280
TEXTURE_MAX_LOD, 159, 162, 173, 280
TEXTURE_MIN_LOD, 157, 159, 162, 173, 280
TEXTURE_PRIORITY, 333
TEXTURE_RECTANGLE, 133, 142, 145–147, 151, 152, 157, 174, 175, 228, 229, 249–252, 279
TEXTURE_RECTANGLE_ARB, 358
TEXTURE_RED_SIZE, 250
TEXTURE_RED_TYPE, 250
TEXTURE_SAMPLES, 235, 281
TEXTURE_SHARED_SIZE, 250, 281
TEXTURE_STENCIL_SIZE, 250
TEXTURE_WIDTH, 149, 151, 153, 154, 251, 281
TEXTURE_WRAP_R, 157, 159, 165, 280, 332
TEXTURE_WRAP_S, 157, 159, 165, 280, 332
TEXTURE_WRAP_T, 157, 159, 165, 280, 332
textureGather, 364
textureLOD, 364
TIMEOUT_EXPIRED, 244
TIMEOUT_IGNORED, 244, 245
TRANSFORM_FEEDBACK_BUFFER, 35, 36, 94, 95
TRANSFORM_FEEDBACK_BUFFER_BINDING, 259, 299
TRANSFORM_FEEDBACK_BUFFER_MODE, 261, 294
TRANSFORM_FEEDBACK_BUFFER_SIZE, 259, 299
TRANSFORM_FEEDBACK_BUFFER_START, 259, 299
TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, 95, 96, 255
TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH, 74, 261, 294
TRANSFORM_FEEDBACK_VARYINGS, 73, 261, 294
TransformFeedbackVaryings, 72, 73, 94
Translate, 330
TRIANGLE_FAN, 22, 25, 83, 88, 93
TRIANGLE_STRIP, 22, 25, 83, 88, 93, 261, 294
TRIANGLE_STRIP_ADJACENCY, 25, 83, 88
TRIANGLES, 23, 25, 83, 88, 93, 261, 294
triangles, 82
TRIANGLES_ADJACENCY, 24, 25, 83, 88, 261
triangles_adjacency, 83
uint, 67
Uniform, 14, 64
Uniform*, 55, 65, 66, 71
Uniform*f{v}, 64, 65
Uniform*i{v}, 64, 65
Uniform*ui{v}, 65
Uniform1f, 15
Uniform1i, 14
Uniform1i{v}, 65, 71
Uniform1iv, 65
Uniform2{if ui}*, 65
Uniform2f, 15
Uniform2i, 15
Uniform3f, 15
Uniform3i, 15
Uniform4f, 13, 15
Uniform4f{v}, 65
Uniform4i, 15
Uniform4i{v}, 65
UNIFORM_ARRAY_STRIDE, 63, 68, 296
UNIFORM_BLOCK_ACTIVE_UNIFORM_INDICES, 58, 296
UNIFORM_BLOCK_ACTIVE_UNIFORMS, 58, 296
UNIFORM_BLOCK_BINDING, 57, 296
UNIFORM_BLOCK_DATA_SIZE, 57, 70, 296
UNIFORM_BLOCK_INDEX, 63, 295
UNIFORM_BLOCK_NAME_LENGTH, 58
UNIFORM_BLOCK_REFERENCED_BY_FRAGMENT_SHADER, 58, 296
UNIFORM_BLOCK_REFERENCED_BY_GEOMETRY_SHADER, 58, 296
UNIFORM_BLOCK_REFERENCED_BY_VERTEX_SHADER, 58, 296
UNIFORM_BUFFER, 35, 36, 70
UNIFORM_BUFFER_BINDING, 258, 295
UNIFORM_BUFFER_OFFSET_-ALIGNMENT, 70, 309
UNIFORM_BUFFER_SIZE, 258, 295
UNIFORM_BUFFER_START, 258, 295
UNIFORM_MATRIX_STRIDE, 64, 67, 296
UNIFORM_NAME_LENGTH, 63, 295
UNIFORM_OFFSET, 63, 295
UNIFORM_SIZE, 63, 295
UNIFORM_TYPE, 63, 295
Uniform{1,2,3,4}ui, 64
Uniform{1,2,3,4}uiv, 64
UniformBlockBinding, 70
UniformMatrix2x4fv, 65
UniformMatrix3fv, 65
UniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}fv, 64, 65
UnmapBuffer, 40, 42, 43, 55, 325
UNPACK_ALIGNMENT, 118, 123, 132, 291
UNPACK_IMAGE_HEIGHT, 118, 132, 291
UNPACK_LSB_FIRST, 118, 291
UNPACK_ROW_LENGTH, 118, 122, 123, 132, 291
UNPACK_SKIP_IMAGES, 118, 132, 142, 291
UNPACK_SKIP_PIXELS, 118, 123, 291
UNPACK_SKIP_ROWS, 118, 123, 291
UNPACK_SWAP_BYTES, 118, 122, 291
UNSIGNED, 243, 257, 300
unsigned int, 61
UNSIGNED_BYTE, 28, 31, 120, 214, 215
UNSIGNED_BYTE_2_3_3_REV, 120,
INDEX

VERTEX_ARRAY_BINDING, 249, 263, 271

VERTEX_ATTRIB责任制、ARRAY_BUFFER_BINDING, 44, 263, 270

VERTEX_ATTRIB_ARRAY_ENABLED, 263, 269

VERTEX_ATTRIB_ARRAY_INTEGER, 263, 269

VERTEX_ATTRIB_ARRAY_NORMALIZED, 263, 269

VERTEX_ATTRIB_ARRAY_POINTER, 263, 269

VERTEX_ATTRIB_ARRAY_SIZE, 263, 269

VERTEX_ATTRIB_ARRAY_STRIDE, 263, 269

VERTEX_ATTRIB_ARRAY_TYPE, 263, 269

VERTEX_PROGRAM_POINT_SIZE, 349

VERTEX_PROGRAM_TWO_SIDE, 331

VERTEX_SHADER, 48, 260

VertexAttrib, 26, 92

VertexAttrib*, 27, 51, 330

VertexAttrib1*, 26

VertexAttrib2*, 26

VertexAttrib3*, 26

VertexAttrib4, 26

VertexAttrib4*, 26

VertexAttrib4N, 26

VertexAttrib4Nub, 26

VertexAttribI, 27

VertexAttribI4, 27

VertexAttribIPointer, 28, 29, 263

VertexAttribPointer, 27, 28, 44, 46, 263, 331

VertexPointer, 330

VIEWPORT, 273

Viewport, 89

WAIT_FAILED, 244

WaitSync, 242–245, 310, 324, 325

WGL_ARB_create_context, 361

WGL_ARB_create_context_profile, 329, 365

WGL_ARB_framebuffer_sRGB, 360

WGL_ARB_pixel_format_float, 358

WindowPos*, 331

WRITE_ONLY, 36, 41, 42

XOR, 199

ZERO, 190, 196, 197, 284

OpenGL 3.2 (Core Profile) - July 24, 2009