The OpenGL® Graphics System:
A Specification
(Version 3.0 (with changes marked) - August 11, 2008)

Mark Segal
Kurt Akeley

Editor (version 1.1): Chris Frazier
Editor (versions 1.2-3.0): Jon Leech
Editor (version 2.0): Pat Brown
# Contents

## 1 Introduction

1.1 Formatting of Optional Features .......................... 1
1.2 What is the OpenGL Graphics System? ......................... 2
1.3 Programmer’s View of OpenGL ............................... 2
1.4 Implementor’s View of OpenGL ............................... 2
1.5 Our View ............................................... 3
1.6 The Deprecation Model ..................................... 3
1.7 Companion Documents ....................................... 3
  1.7.1 OpenGL Shading Language ............................ 3
  1.7.2 Window System Bindings ............................. 4

## 2 OpenGL Operation

2.1 OpenGL Fundamentals ..................................... 5
  2.1.1 Floating-Point Computation .......................... 7
  2.1.2 16-Bit Floating-Point Numbers ....................... 8
  2.1.3Unsigned 11-Bit Floating-Point Numbers ................ 9
  2.1.4Unsigned 10-Bit Floating-Point Numbers ............... 9
2.2 GL State ................................................. 10
  2.2.1 Shared Object State ................................. 11
2.3 GL Command Syntax ..................................... 11
2.4 Basic GL Operation ...................................... 13
2.5 GL Errors ................................................ 15
2.6 Begin/End Paradigm ..................................... 16
  2.6.1 Begin and End .................................... 19
  2.6.2 Polygon Edges .................................... 23
  2.6.3 GL Commands within Begin/End ...................... 24
2.7 Vertex Specification ..................................... 24
2.8 Vertex Arrays .......................................... 28
2.9 Buffer Objects ......................................... 38
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.9.1</td>
<td>Vertex Arrays in Buffer Objects</td>
<td>45</td>
</tr>
<tr>
<td>2.9.2</td>
<td>Array Indices in Buffer Objects</td>
<td>46</td>
</tr>
<tr>
<td>2.9.3</td>
<td>Buffer Object State</td>
<td>47</td>
</tr>
<tr>
<td>2.10</td>
<td>Vertex Array Objects</td>
<td>47</td>
</tr>
<tr>
<td>2.11</td>
<td>Rectangles</td>
<td>48</td>
</tr>
<tr>
<td>2.12</td>
<td>Coordinate Transformations</td>
<td>49</td>
</tr>
<tr>
<td>2.12.1</td>
<td>Controlling the Viewport</td>
<td>50</td>
</tr>
<tr>
<td>2.12.2</td>
<td>Matrices</td>
<td>51</td>
</tr>
<tr>
<td>2.12.3</td>
<td>Normal Transformation</td>
<td>57</td>
</tr>
<tr>
<td>2.12.4</td>
<td>Generating Texture Coordinates</td>
<td>59</td>
</tr>
<tr>
<td>2.13</td>
<td>Asynchronous Queries</td>
<td>61</td>
</tr>
<tr>
<td>2.14</td>
<td>Conditional Rendering</td>
<td>62</td>
</tr>
<tr>
<td>2.15</td>
<td>Transform Feedback</td>
<td>63</td>
</tr>
<tr>
<td>2.16</td>
<td>Primitive Queries</td>
<td>67</td>
</tr>
<tr>
<td>2.17</td>
<td>Clipping</td>
<td>67</td>
</tr>
<tr>
<td>2.18</td>
<td>Current Raster Position</td>
<td>70</td>
</tr>
<tr>
<td>2.19</td>
<td>Colors and Coloring</td>
<td>73</td>
</tr>
<tr>
<td>2.19.1</td>
<td>Lighting</td>
<td>75</td>
</tr>
<tr>
<td>2.19.2</td>
<td>Lighting Parameter Specification</td>
<td>79</td>
</tr>
<tr>
<td>2.19.3</td>
<td><strong>ColorMaterial</strong></td>
<td>83</td>
</tr>
<tr>
<td>2.19.4</td>
<td>Lighting State</td>
<td>83</td>
</tr>
<tr>
<td>2.19.5</td>
<td>Color Index Lighting</td>
<td>84</td>
</tr>
<tr>
<td>2.19.6</td>
<td>Clamping or Masking</td>
<td>85</td>
</tr>
<tr>
<td>2.19.7</td>
<td>Flatshading</td>
<td>85</td>
</tr>
<tr>
<td>2.19.8</td>
<td>Color and Associated Data Clipping</td>
<td>86</td>
</tr>
<tr>
<td>2.19.9</td>
<td>Final Color Processing</td>
<td>87</td>
</tr>
<tr>
<td>2.20</td>
<td>Vertex Shaders</td>
<td>87</td>
</tr>
<tr>
<td>2.20.1</td>
<td>Shader Objects</td>
<td>88</td>
</tr>
<tr>
<td>2.20.2</td>
<td>Program Objects</td>
<td>89</td>
</tr>
<tr>
<td>2.20.3</td>
<td>Shader Variables</td>
<td>91</td>
</tr>
<tr>
<td>2.20.4</td>
<td>Shader Execution</td>
<td>103</td>
</tr>
<tr>
<td>2.20.5</td>
<td>Required State</td>
<td>109</td>
</tr>
<tr>
<td>3</td>
<td>Rasterization</td>
<td>111</td>
</tr>
<tr>
<td>3.1</td>
<td>Discarding Primitives Before Rasterization</td>
<td>113</td>
</tr>
<tr>
<td>3.2</td>
<td>Invariance</td>
<td>113</td>
</tr>
<tr>
<td>3.3</td>
<td>Antialiasing</td>
<td>113</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Multisampling</td>
<td>114</td>
</tr>
<tr>
<td>3.4</td>
<td>Points</td>
<td>116</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Basic Point Rasterization</td>
<td>118</td>
</tr>
</tbody>
</table>

Version 3.0 (with changes marked - August 11, 2008)
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.2</td>
<td>Point Rasterization State</td>
<td>122</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Point Multisample Rasterization</td>
<td>122</td>
</tr>
<tr>
<td>3.5</td>
<td>Line Segments</td>
<td>122</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Basic Line Segment Rasterization</td>
<td>123</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Other Line Segment Features</td>
<td>125</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Line Rasterization State</td>
<td>128</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Line Multisample Rasterization</td>
<td>128</td>
</tr>
<tr>
<td>3.6</td>
<td>Polygons</td>
<td>129</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Basic Polygon Rasterization</td>
<td>129</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Stippling</td>
<td>131</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Anti-aliasing</td>
<td>132</td>
</tr>
<tr>
<td>3.6.4</td>
<td>Options Controlling Polygon Rasterization</td>
<td>132</td>
</tr>
<tr>
<td>3.6.5</td>
<td>Depth Offset</td>
<td>133</td>
</tr>
<tr>
<td>3.6.6</td>
<td>Polygon Multisample Rasterization</td>
<td>134</td>
</tr>
<tr>
<td>3.6.7</td>
<td>Polygon Rasterization State</td>
<td>135</td>
</tr>
<tr>
<td>3.7</td>
<td>Pixel Rectangles</td>
<td>135</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Pixel Storage Modes and Pixel Buffer Objects</td>
<td>135</td>
</tr>
<tr>
<td>3.7.2</td>
<td>The Imaging Subset</td>
<td>137</td>
</tr>
<tr>
<td>3.7.3</td>
<td>Pixel Transfer Modes</td>
<td>137</td>
</tr>
<tr>
<td>3.7.4</td>
<td>Rasterization of Pixel Rectangles</td>
<td>148</td>
</tr>
<tr>
<td>3.7.5</td>
<td>Pixel Transfer Operations</td>
<td>162</td>
</tr>
<tr>
<td>3.7.6</td>
<td>Pixel Rectangle Multisample Rasterization</td>
<td>172</td>
</tr>
<tr>
<td>3.8</td>
<td>Bitmaps</td>
<td>173</td>
</tr>
<tr>
<td>3.9</td>
<td>Texturing</td>
<td>175</td>
</tr>
<tr>
<td>3.9.1</td>
<td>Texture Image Specification</td>
<td>176</td>
</tr>
<tr>
<td>3.9.2</td>
<td>Alternate Texture Image Specification Commands</td>
<td>191</td>
</tr>
<tr>
<td>3.9.3</td>
<td>Compressed Texture Images</td>
<td>196</td>
</tr>
<tr>
<td>3.9.4</td>
<td>Texture Parameters</td>
<td>200</td>
</tr>
<tr>
<td>3.9.5</td>
<td>Depth Component Textures</td>
<td>202</td>
</tr>
<tr>
<td>3.9.6</td>
<td>Cube Map Texture Selection</td>
<td>202</td>
</tr>
<tr>
<td>3.9.7</td>
<td>Texture Minification</td>
<td>203</td>
</tr>
<tr>
<td>3.9.8</td>
<td>Texture Magnification</td>
<td>212</td>
</tr>
<tr>
<td>3.9.9</td>
<td>Combined Depth/Stencil Textures</td>
<td>212</td>
</tr>
<tr>
<td>3.9.10</td>
<td>Texture Completeness</td>
<td>213</td>
</tr>
<tr>
<td>3.9.11</td>
<td>Texture State and Proxy State</td>
<td>214</td>
</tr>
<tr>
<td>3.9.12</td>
<td>Texture Objects</td>
<td>216</td>
</tr>
<tr>
<td>3.9.13</td>
<td>Texture Environments and Texture Functions</td>
<td>219</td>
</tr>
<tr>
<td>3.9.14</td>
<td>Texture Comparison Modes</td>
<td>224</td>
</tr>
<tr>
<td>3.9.15</td>
<td>sRGB Texture Color Conversion</td>
<td>225</td>
</tr>
<tr>
<td>3.9.16</td>
<td>Shared Exponent Texture Color Conversion</td>
<td>226</td>
</tr>
</tbody>
</table>

Version 3.0 (with changes marked - August 11, 2008)
3.9.17 Texture Application ........................................... 226
3.10 Color Sum .......................................................... 229
3.11 Fog .................................................................. 229
3.12 Fragment Shaders .................................................. 231
  3.12.1 Shader Variables .............................................. 231
  3.12.2 Shader Execution .............................................. 232
3.13 Antialiasing Application ......................................... 237
3.14 Multisample Point Fade ........................................... 237

4 Per-Fragment Operations and the Framebuffer ............. 238
  4.1 Per-Fragment Operations .......................................... 240
    4.1.1 Pixel Ownership Test ......................................... 240
    4.1.2 Scissor Test ................................................... 241
    4.1.3 Multisample Fragment Operations ......................... 241
    4.1.4 Alpha Test .................................................... 242
    4.1.5 Stencil Test .................................................... 243
    4.1.6 Depth Buffer Test ............................................. 245
    4.1.7 Occlusion Queries ............................................. 245
    4.1.8 Blending ........................................................ 246
    4.1.9 sRGB Conversion .............................................. 251
    4.1.10 Dithering ..................................................... 251
    4.1.11 Logical Operation ............................................ 252
    4.1.12 Additional Multisample Fragment Operations ........... 253
  4.2 Whole Framebuffer Operations ................................... 254
    4.2.1 Selecting a Buffer for Writing ............................... 254
    4.2.2 Fine Control of Buffer Updates ............................. 258
    4.2.3 Clearing the Buffers .......................................... 260
    4.2.4 The Accumulation Buffer .................................... 263
  4.3 Drawing, Reading, and Copying Pixels ....................... 264
    4.3.1 Writing to the Stencil or Depth/Stencil Buffers ........ 265
    4.3.2 Reading Pixels ............................................... 265
    4.3.3 Copying Pixels ............................................... 273
    4.3.4 Pixel Draw/Read State ....................................... 277
  4.4 Framebuffer Objects .............................................. 277
    4.4.1 Binding and Managing Framebuffer Objects ................ 278
    4.4.2 Attaching Images to Framebuffer Objects ................. 281
    4.4.3 Rendering When an Image of a Bound Texture Object is Also Attached to the Framebuffer 288
    4.4.4 Framebuffer Completeness .................................. 289

Version 3.0 (with changes marked - August 11, 2008)
CONTENTS

B Corollaries 396

C Compressed Texture Image Formats 399
  C.1 RGTC Compressed Texture Image Formats 399
    C.1.1 Format COMPRESSED_RED_RGTC1 400
    C.1.2 Format COMPRESSED_SIGNED_RED_RGTC1 401
    C.1.3 Format COMPRESSED_RG_RGTC2 402
    C.1.4 Format COMPRESSED_SIGNED_RG_RGTC2 402

D Shared Objects and Multiple Contexts 403
  D.1 Object Deletion Behavior 403

E The Deprecation Model 404
  E.1 Profiles and Deprecated Features of OpenGL 3.0 404

F Version 1.1 410
  F.1 Vertex Array 410
  F.2 Polygon Offset 411
  F.3 Logical Operation 411
  F.4 Texture Image Formats 411
  F.5 Texture Replace Environment 411
  F.6 Texture Proxies 412
  F.7 Copy Texture and Subtexture 412
  F.8 Texture Objects 412
  F.9 Other Changes 412
  F.10 Acknowledgements 413

G Version 1.2 415
  G.1 Three-Dimensional Texturing 415
  G.2 BGRA Pixel Formats 415
  G.3 Packed Pixel Formats 416
  G.4 Normal Rescaling 416
  G.5 Separate Specular Color 416
  G.6 Texture Coordinate Edge Clamping 416
  G.7 Texture Level of Detail Control 417
  G.8 Vertex Array Draw Element Range 417
  G.9 Imaging Subset 417
    G.9.1 Color Tables 417
    G.9.2 Convolution 418
    G.9.3 Color Matrix 418

Version 3.0 (with changes marked - August 11, 2008)
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>G.9.4</td>
<td>Pixel Pipeline Statistics</td>
<td>419</td>
</tr>
<tr>
<td>G.9.5</td>
<td>Constant Blend Color</td>
<td>419</td>
</tr>
<tr>
<td>G.9.6</td>
<td>New Blending Equations</td>
<td>419</td>
</tr>
<tr>
<td>G.10</td>
<td>Acknowledgements</td>
<td>419</td>
</tr>
</tbody>
</table>

### H Version 1.2.1

#### I Version 1.3

1.1 Compressed Textures | 424
1.2 Cube Map Textures | 424
1.3 Multisample | 425
1.4 Multitexture | 425
1.5 Texture Add Environment Mode | 426
1.6 Texture Combine Environment Mode | 426
1.7 Texture Dot3 Environment Mode | 426
1.8 Texture Border Clamp | 426
1.9 Transpose Matrix | 427
1.10 Acknowledgements | 427

### J Version 1.4

J.1 Automatic Mipmap Generation | 432
J.2 Blend Squaring | 432
J.3 Changes to the Imaging Subset | 433
J.4 Depth Textures and Shadows | 433
J.5 Fog Coordinate | 433
J.6 Multiple Draw Arrays | 433
J.7 Point Parameters | 434
J.8 Secondary Color | 434
J.9 Separate Blend Functions | 434
J.10 Stencil Wrap | 434
J.11 Texture Crossbar Environment Mode | 434
J.12 Texture LOD Bias | 435
J.13 Texture Mirrored Repeat | 435
J.14 Window Raster Position | 435
J.15 Acknowledgements | 435

### K Version 1.5

K.1 Buffer Objects | 438
K.2 Occlusion Queries | 439
K.3 Shadow Functions | 439

Version 3.0 (with changes marked - August 11, 2008)
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>O.9</td>
<td>Texture Border Clamp</td>
<td>463</td>
</tr>
<tr>
<td>O.10</td>
<td>Point Parameters</td>
<td>463</td>
</tr>
<tr>
<td>O.11</td>
<td>Vertex Blend</td>
<td>463</td>
</tr>
<tr>
<td>O.12</td>
<td>Matrix Palette</td>
<td>464</td>
</tr>
<tr>
<td>O.13</td>
<td>Texture Combine Environment Mode</td>
<td>464</td>
</tr>
<tr>
<td>O.14</td>
<td>Texture Crossbar Environment Mode</td>
<td>464</td>
</tr>
<tr>
<td>O.15</td>
<td>Texture Dot3 Environment Mode</td>
<td>464</td>
</tr>
<tr>
<td>O.16</td>
<td>Texture Mirrored Repeat</td>
<td>464</td>
</tr>
<tr>
<td>O.17</td>
<td>Depth Texture</td>
<td>464</td>
</tr>
<tr>
<td>O.18</td>
<td>Shadow</td>
<td>464</td>
</tr>
<tr>
<td>O.19</td>
<td>Shadow Ambient</td>
<td>465</td>
</tr>
<tr>
<td>O.20</td>
<td>Window Raster Position</td>
<td>465</td>
</tr>
<tr>
<td>O.21</td>
<td>Low-Level Vertex Programming</td>
<td>465</td>
</tr>
<tr>
<td>O.22</td>
<td>Low-Level Fragment Programming</td>
<td>465</td>
</tr>
<tr>
<td>O.23</td>
<td>Buffer Objects</td>
<td>465</td>
</tr>
<tr>
<td>O.24</td>
<td>Occlusion Queries</td>
<td>466</td>
</tr>
<tr>
<td>O.25</td>
<td>Shader Objects</td>
<td>466</td>
</tr>
<tr>
<td>O.26</td>
<td>High-Level Vertex Programming</td>
<td>466</td>
</tr>
<tr>
<td>O.27</td>
<td>High-Level Fragment Programming</td>
<td>466</td>
</tr>
<tr>
<td>O.28</td>
<td>OpenGL Shading Language</td>
<td>466</td>
</tr>
<tr>
<td>O.29</td>
<td>Non-Power-Of-Two Textures</td>
<td>466</td>
</tr>
<tr>
<td>O.30</td>
<td>Point Sprites</td>
<td>467</td>
</tr>
<tr>
<td>O.31</td>
<td>Fragment Program Shadow</td>
<td>467</td>
</tr>
<tr>
<td>O.32</td>
<td>Multiple Render Targets</td>
<td>467</td>
</tr>
<tr>
<td>O.33</td>
<td>Rectangular Textures</td>
<td>467</td>
</tr>
<tr>
<td>O.34</td>
<td>Floating-Point Color Buffers</td>
<td>467</td>
</tr>
<tr>
<td>O.35</td>
<td>Half-Precision Floating Point</td>
<td>468</td>
</tr>
<tr>
<td>O.36</td>
<td>Floating-Point Textures</td>
<td>468</td>
</tr>
<tr>
<td>O.37</td>
<td>Pixel Buffer Objects</td>
<td>468</td>
</tr>
</tbody>
</table>

## Index

469
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Block diagram of the GL.</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Creation of a processed vertex from a transformed vertex and cur-</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>rent values.</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>Primitive assembly and processing</td>
<td>19</td>
</tr>
<tr>
<td>2.4</td>
<td>Triangle strips, fans, and independent triangles.</td>
<td>21</td>
</tr>
<tr>
<td>2.5</td>
<td>Quadrilateral strips and independent quadrilaterals.</td>
<td>22</td>
</tr>
<tr>
<td>2.6</td>
<td>Vertex transformation sequence.</td>
<td>49</td>
</tr>
<tr>
<td>2.7</td>
<td>Current raster position.</td>
<td>71</td>
</tr>
<tr>
<td>2.8</td>
<td>Processing of RGBA colors.</td>
<td>73</td>
</tr>
<tr>
<td>2.9</td>
<td>Processing of color indices.</td>
<td>73</td>
</tr>
<tr>
<td>2.10</td>
<td>ColorMaterial operation.</td>
<td>83</td>
</tr>
<tr>
<td>3.1</td>
<td>Rasterization.</td>
<td>111</td>
</tr>
<tr>
<td>3.2</td>
<td>Rasterization of non-antialiased wide points.</td>
<td>118</td>
</tr>
<tr>
<td>3.3</td>
<td>Rasterization of antialiased wide points.</td>
<td>118</td>
</tr>
<tr>
<td>3.4</td>
<td>Visualization of Bresenham’s algorithm.</td>
<td>123</td>
</tr>
<tr>
<td>3.5</td>
<td>Rasterization of non-antialiased wide lines.</td>
<td>126</td>
</tr>
<tr>
<td>3.6</td>
<td>The region used in rasterizing an antialiased line segment.</td>
<td>127</td>
</tr>
<tr>
<td>3.7</td>
<td>Operation of <code>DrawPixels</code>.</td>
<td>148</td>
</tr>
<tr>
<td>3.8</td>
<td>Selecting a subimage from an image</td>
<td>153</td>
</tr>
<tr>
<td>3.9</td>
<td>A bitmap and its associated parameters.</td>
<td>173</td>
</tr>
<tr>
<td>3.10</td>
<td>A texture image and the coordinates used to access it.</td>
<td>189</td>
</tr>
<tr>
<td>3.11</td>
<td>Multitexture pipeline.</td>
<td>227</td>
</tr>
<tr>
<td>4.1</td>
<td>Per-fragment operations.</td>
<td>240</td>
</tr>
<tr>
<td>4.2</td>
<td>Operation of <code>ReadPixels</code>.</td>
<td>265</td>
</tr>
<tr>
<td>4.3</td>
<td>Operation of <code>CopyPixels</code>.</td>
<td>273</td>
</tr>
<tr>
<td>5.1</td>
<td>Map Evaluation.</td>
<td>299</td>
</tr>
<tr>
<td>5.2</td>
<td>Feedback syntax.</td>
<td>308</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>GL command suffixes</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>GL data types</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Summary of GL errors</td>
<td>17</td>
</tr>
<tr>
<td>2.4</td>
<td>Vertex array sizes (values per vertex) and data types</td>
<td>30</td>
</tr>
<tr>
<td>2.5</td>
<td>Variables that direct the execution of InterleavedArrays</td>
<td>37</td>
</tr>
<tr>
<td>2.6</td>
<td>Buffer object parameters and their values</td>
<td>39</td>
</tr>
<tr>
<td>2.7</td>
<td>Buffer object initial state</td>
<td>41</td>
</tr>
<tr>
<td>2.8</td>
<td>Buffer object state set by MapBuffer</td>
<td>42</td>
</tr>
<tr>
<td>2.9</td>
<td>Transform feedback modes</td>
<td>64</td>
</tr>
<tr>
<td>2.10</td>
<td>Component conversions</td>
<td>74</td>
</tr>
<tr>
<td>2.11</td>
<td>Summary of lighting parameters</td>
<td>76</td>
</tr>
<tr>
<td>2.12</td>
<td>Correspondence of lighting parameter symbols to names</td>
<td>81</td>
</tr>
<tr>
<td>2.13</td>
<td>Polygon flatshading color selection</td>
<td>86</td>
</tr>
<tr>
<td>3.1</td>
<td>PixelStore parameters</td>
<td>136</td>
</tr>
<tr>
<td>3.2</td>
<td>PixelTransfer parameters</td>
<td>138</td>
</tr>
<tr>
<td>3.3</td>
<td>PixelMap parameters</td>
<td>139</td>
</tr>
<tr>
<td>3.4</td>
<td>Color table names</td>
<td>140</td>
</tr>
<tr>
<td>3.5</td>
<td>DrawPixels and ReadPixels types</td>
<td>151</td>
</tr>
<tr>
<td>3.6</td>
<td>DrawPixels and ReadPixels formats</td>
<td>152</td>
</tr>
<tr>
<td>3.7</td>
<td>Swap Bytes bit ordering</td>
<td>153</td>
</tr>
<tr>
<td>3.8</td>
<td>Packed pixel formats</td>
<td>155</td>
</tr>
<tr>
<td>3.9</td>
<td>UNSIGNED_BYTE formats. Bit numbers are indicated for each component.</td>
<td>156</td>
</tr>
<tr>
<td>3.10</td>
<td>UNSIGNED_SHORT formats</td>
<td>157</td>
</tr>
<tr>
<td>3.11</td>
<td>UNSIGNED_INT formats</td>
<td>158</td>
</tr>
<tr>
<td>3.12</td>
<td>Packed pixel field assignments</td>
<td>159</td>
</tr>
<tr>
<td>3.13</td>
<td>Color table lookup</td>
<td>165</td>
</tr>
<tr>
<td>3.14</td>
<td>Computation of filtered color components</td>
<td>166</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.15</td>
<td>Conversion from RGBA, depth, and stencil pixel components to internal texture, table, or filter components.</td>
<td>178</td>
</tr>
<tr>
<td>3.16</td>
<td>Sized internal color formats.</td>
<td>183</td>
</tr>
<tr>
<td>3.18</td>
<td>Sized internal depth and stencil formats.</td>
<td>184</td>
</tr>
<tr>
<td>3.17</td>
<td>Sized internal luminance and intensity formats.</td>
<td>184</td>
</tr>
<tr>
<td>3.19</td>
<td>Generic and specific compressed internal formats.</td>
<td>185</td>
</tr>
<tr>
<td>3.20</td>
<td>Texture parameters and their values.</td>
<td>201</td>
</tr>
<tr>
<td>3.21</td>
<td>Selection of cube map images.</td>
<td>202</td>
</tr>
<tr>
<td>3.22</td>
<td>Texel location wrap mode application.</td>
<td>207</td>
</tr>
<tr>
<td>3.23</td>
<td>Correspondence of filtered texture components to texture source components.</td>
<td>220</td>
</tr>
<tr>
<td>3.24</td>
<td>Texture functions REPLACE, MODULATE, and DECAL.</td>
<td>221</td>
</tr>
<tr>
<td>3.25</td>
<td>Texture functions BLEND and ADD.</td>
<td>221</td>
</tr>
<tr>
<td>3.26</td>
<td>COMBINE texture functions.</td>
<td>222</td>
</tr>
<tr>
<td>3.27</td>
<td>Arguments for COMBINE_RGB functions.</td>
<td>223</td>
</tr>
<tr>
<td>3.28</td>
<td>Arguments for COMBINE_ALPHA functions.</td>
<td>223</td>
</tr>
<tr>
<td>3.29</td>
<td>Depth texture comparison functions.</td>
<td>225</td>
</tr>
<tr>
<td>4.1</td>
<td>RGB and Alpha blend equations.</td>
<td>248</td>
</tr>
<tr>
<td>4.2</td>
<td>Blending functions.</td>
<td>250</td>
</tr>
<tr>
<td>4.3</td>
<td>Arguments to LogicOp and their corresponding operations.</td>
<td>253</td>
</tr>
<tr>
<td>4.4</td>
<td>Buffer selection for the default framebuffer</td>
<td>256</td>
</tr>
<tr>
<td>4.5</td>
<td>Buffer selection for a framebuffer object</td>
<td>256</td>
</tr>
<tr>
<td>4.6</td>
<td>DrawBuffers buffer selection for the default framebuffer</td>
<td>257</td>
</tr>
<tr>
<td>4.7</td>
<td>PixelStore parameters.</td>
<td>267</td>
</tr>
<tr>
<td>4.8</td>
<td>ReadPixels index masks.</td>
<td>271</td>
</tr>
<tr>
<td>4.9</td>
<td>ReadPixels GL data types and reversed component conversion formulas.</td>
<td>272</td>
</tr>
<tr>
<td>4.10</td>
<td>Effective ReadPixels format for DEPTH_STENCIL CopyPixels operation.</td>
<td>275</td>
</tr>
<tr>
<td>4.11</td>
<td>Correspondence of renderbuffer sized to base internal formats.</td>
<td>283</td>
</tr>
<tr>
<td>4.12</td>
<td>Framebuffer attachment points.</td>
<td>285</td>
</tr>
<tr>
<td>5.1</td>
<td>Values specified by the target to Map1.</td>
<td>298</td>
</tr>
<tr>
<td>5.2</td>
<td>Correspondence of feedback type to number of values per vertex.</td>
<td>307</td>
</tr>
<tr>
<td>5.3</td>
<td>Hint targets and descriptions.</td>
<td>314</td>
</tr>
<tr>
<td>6.1</td>
<td>Texture, table, and filter return values.</td>
<td>324</td>
</tr>
<tr>
<td>6.2</td>
<td>Attribute groups.</td>
<td>340</td>
</tr>
<tr>
<td>6.3</td>
<td>State Variable Types.</td>
<td>342</td>
</tr>
<tr>
<td>Table Number</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.4</td>
<td>GL Internal begin-end state variables (inaccessible)</td>
<td>343</td>
</tr>
<tr>
<td>6.5</td>
<td>Current Values and Associated Data</td>
<td>344</td>
</tr>
<tr>
<td>6.6</td>
<td>Vertex Array Object State</td>
<td>345</td>
</tr>
<tr>
<td>6.7</td>
<td>Vertex Array Object State (cont.)</td>
<td>346</td>
</tr>
<tr>
<td>6.8</td>
<td>Vertex Array Object State (cont.)</td>
<td>347</td>
</tr>
<tr>
<td>6.9</td>
<td>Vertex Array Object State (cont.)</td>
<td>348</td>
</tr>
<tr>
<td>6.10</td>
<td>Vertex Array Data (not in Vertex Array objects)</td>
<td>349</td>
</tr>
<tr>
<td>6.11</td>
<td>Buffer Object State</td>
<td>350</td>
</tr>
<tr>
<td>6.12</td>
<td>Transformation state</td>
<td>351</td>
</tr>
<tr>
<td>6.13</td>
<td>Coloring</td>
<td>352</td>
</tr>
<tr>
<td>6.14</td>
<td>Lighting (see also table 2.11 for defaults)</td>
<td>353</td>
</tr>
<tr>
<td>6.15</td>
<td>Lighting (cont.)</td>
<td>354</td>
</tr>
<tr>
<td>6.16</td>
<td>Rasterization</td>
<td>355</td>
</tr>
<tr>
<td>6.17</td>
<td>Rasterization (cont.)</td>
<td>356</td>
</tr>
<tr>
<td>6.18</td>
<td>Multisampling</td>
<td>357</td>
</tr>
<tr>
<td>6.19</td>
<td>Textures (state per texture unit and binding point)</td>
<td>358</td>
</tr>
<tr>
<td>6.20</td>
<td>Textures (state per texture object)</td>
<td>359</td>
</tr>
<tr>
<td>6.21</td>
<td>Textures (state per texture image)</td>
<td>360</td>
</tr>
<tr>
<td>6.22</td>
<td>Texture Environment and Generation</td>
<td>361</td>
</tr>
<tr>
<td>6.23</td>
<td>Texture Environment and Generation (cont.)</td>
<td>362</td>
</tr>
<tr>
<td>6.24</td>
<td>Pixel Operations</td>
<td>363</td>
</tr>
<tr>
<td>6.25</td>
<td>Pixel Operations (cont.)</td>
<td>364</td>
</tr>
<tr>
<td>6.26</td>
<td>Framebuffer Control</td>
<td>365</td>
</tr>
<tr>
<td>6.27</td>
<td>Framebuffer (state per target binding point)</td>
<td>366</td>
</tr>
<tr>
<td>6.28</td>
<td>Framebuffer (state per framebuffer object)</td>
<td>367</td>
</tr>
<tr>
<td>6.29</td>
<td>Framebuffer (state per attachment point)</td>
<td>368</td>
</tr>
<tr>
<td>6.30</td>
<td>Renderbuffer (state per target and binding point)</td>
<td>369</td>
</tr>
<tr>
<td>6.31</td>
<td>Renderbuffer (state per renderbuffer object)</td>
<td>370</td>
</tr>
<tr>
<td>6.32</td>
<td>Pixels</td>
<td>371</td>
</tr>
<tr>
<td>6.33</td>
<td>Pixels (cont.)</td>
<td>372</td>
</tr>
<tr>
<td>6.34</td>
<td>Pixels (cont.)</td>
<td>373</td>
</tr>
<tr>
<td>6.35</td>
<td>Pixels (cont.)</td>
<td>374</td>
</tr>
<tr>
<td>6.36</td>
<td>Pixels (cont.)</td>
<td>375</td>
</tr>
<tr>
<td>6.37</td>
<td>Pixels (cont.)</td>
<td>376</td>
</tr>
<tr>
<td>6.38</td>
<td>Evaluators (GetMap takes a map name)</td>
<td>377</td>
</tr>
<tr>
<td>6.39</td>
<td>Shader Object State</td>
<td>378</td>
</tr>
<tr>
<td>6.40</td>
<td>Program Object State</td>
<td>379</td>
</tr>
<tr>
<td>6.41</td>
<td>Program Object State (cont.)</td>
<td>380</td>
</tr>
<tr>
<td>6.42</td>
<td>Vertex Shader State</td>
<td>381</td>
</tr>
<tr>
<td>6.43</td>
<td>Query Object State</td>
<td>382</td>
</tr>
</tbody>
</table>
LIST OF TABLES

6.44 Transform Feedback State ........................................ 383
6.45 Hints ............................................................... 384
6.46 Implementation Dependent Values ................................. 385
6.47 Implementation Dependent Values (cont.) ......................... 386
6.48 Implementation Dependent Values (cont.) ......................... 387
6.49 Implementation Dependent Values (cont.) ......................... 388
6.50 Implementation Dependent Values (cont.) ......................... 389
6.51 Framebuffer Dependent Values .................................... 390
6.52 Miscellaneous ...................................................... 391
K.1 New token names ..................................................... 440
N.1 New token names ..................................................... 458
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.

Changes Relative to OpenGL 2.1

As an aid to readers, this version of the OpenGL 3.0 Specification marks changes relative to the OpenGL 2.1 Specification by typesetting them in purple, like this paragraph. Marginal comments describe reasons for changes; they are retained because they show the relationship to extensions on which OpenGL 3.0 features were based, but please note that the marginal comments also include a large number of Khronos-internal bug numbers and comments, and were not originally written for public consumption, but for use by the ARB members while reviewing specification drafts prior to release.

1.1 Formatting of Optional Features

Starting with version 1.2 of OpenGL, some features in the specification are considered optional; an OpenGL implementation may or may not choose to provide them (see section 3.7.2).

Portions of the specification which are optional are so described where the optional features are first defined (see section 3.7.2). State table entries which are optional are typeset against a gray background.
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, polygons, and bitmaps, but the way that some of this drawing occurs (such as when antialiasing or texturing 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. For the most part, OpenGL provides an immediate-mode interface, meaning that specifying an object causes it to be drawn.

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
1.5. OUR VIEW

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 2.20 and 3.12). These sections may include references to concepts and terms (such as shading language variable types) defined in the companion document.

OpenGL 3.0 implementations are guaranteed to support at least versions 1.10, 1.5. OUR VIEW

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 2.20 and 3.12). These sections may include references to concepts and terms (such as shading language variable types) defined in the companion document.

OpenGL 3.0 implementations are guaranteed to support at least versions 1.10, 
1.7. COMPANION DOCUMENTS

1.20, and 1.30 of the shading language, although versions 1.10 and 1.20 are deprecated in a forward-compatible context. The actual version supported may be queried as described in section 6.1.11.

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 O).

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 NSGLView. 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, polygon, or pixel rectangle. 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 (consisting of positional coordinates, colors, normals, and texture coordinates) are associated with a vertex and each vertex is processed independently, in order, and in the same way. The only exception to this rule is if the group of vertices must be clipped so that the indicated primitive fits within a specified region; in this case vertex data may be modified and new vertices created. The type of clipping depends on which primitive the group of vertices represents.

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

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

The GL provides direct control over the fundamental operations of 3D and 2D graphics. This includes specification of such parameters as vertex and fragment shaders, transformation matrices, lighting equation coefficients, antialiasing methods, and pixel update operators. 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 such as GLX.
2.1. OPENGL FUNDAMENTALS

WGL, and CGL for GL implementations running on the X Window System, Microsoft Windows, and MacOS X respectively.

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 $2^{16}$. The maximum representable magnitude for all other floating-point values must be at least $2^{32}$.

$x \cdot 0 = 0 \cdot x = 0$ for any non-infinite and non-NaN $x$, $1 \cdot x = x \cdot 1 = x$. $x + 0 = 0 + x = x$. $0^0 = 1$. (Occasionally further requirements will be specified.) Most single-precision floating-point formats meet these requirements.

The special values $\text{Inf}$ and $-\text{Inf}$ encode values with magnitudes too large to be represented; the special value $\text{NaN}$ encodes “Not A Number” values resulting

Version 3.0 (with changes marked - August 11, 2008)
from undefined arithmetic operations such as $\frac{1}{0}$. Implementations are permitted, but not required, to support $\text{Infs}$ and $\text{NaNs}$ 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:

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

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

- $S = \left\lfloor \frac{N \mod 65536}{32768} \right\rfloor$
- $E = \left\lfloor \frac{N \mod 32768}{1024} \right\rfloor$
- $M = N \mod 1024$.

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 $\text{Inf}$ or $\text{NaN}$) to such a command is unspecified, but must not lead to GL interruption or termination. Providing a denormalized number or negative zero to GL must yield predictable results. 
2.1. OPENGL FUNDAMENTALS

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 determined by the following:

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

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:

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

Version 3.0 (with changes marked - August 11, 2008)
2.2. GL STATE

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

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

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

\[ M = N \mod 32. \]

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

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

**Bug 3502 - remove negative zero**

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

Version 3.0 (with changes marked - August 11, 2008)
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 come from the `Vertex` command:

```c
void Vertex3f(float x, float y, float z);
```

and

```c
void Vertex2sv(short v[2]);
```

These examples show the ANSI C declarations for these commands. In general, a command declaration has the form¹

¹The 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.
2.3. **GL COMMAND SYNTAX**

<table>
<thead>
<tr>
<th>Letter</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>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 letters to GL argument types. Refer to table 2.2 for definitions of the GL types.

\[
\text{rtype Name}\{\epsilon1234}\{\epsilon b s i f d ub us ui}\{ev\}
\begin{align*}
&= (\{args,\} T arg1, \ldots, T argN [, args]) ;
\end{align*}
\]

rype is the return type of the function. The braces ({})) enclose a series of characters (or character pairs) of which one is selected. \(\epsilon\) indicates no character. The arguments enclosed in brackets ([args,] and [, args]) may or may not be present. The \(N\) arguments \(arg1\) through \(argN\) have type \(T\), which corresponds to one of the type letters or letter pairs as indicated in table 2.1 (if there are no letters, then the arguments’ type is given explicitly). If the final character is not \(v\), then \(N\) is given by the digit 1, 2, 3, or 4 (if there is no digit, then the number of arguments is fixed). If the final character is \(v\), then only \(arg1\) is present and it is an array of \(N\) values of the indicated type. Finally, we indicate an unsigned type by the shorthand of prepending a \(u\) to the beginning of the type name (so that, for instance, \(\text{unsigned char}\) is abbreviated \(uchar\)).

For example,

\[
\text{void Normal3}\{fd\}( T arg );
\]

indicates the two declarations

\[
\text{void Normal3f( float arg1, float arg2, float arg3 );}
\text{void Normal3d( double arg1, double arg2, double arg3 );}
\]

while

\[
\text{void Normal3}\{fd\}v( T arg );
\]
means the two declarations

```c
void Normal3fv(float arg[3]);
void Normal3dv(double arg[3]);
```

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. Most commands may be accumulated in a display list for processing by the GL at a later time. Otherwise, commands are effectively sent through a processing pipeline.

The first stage provides an efficient means for approximating curve and surface geometry by evaluating polynomial functions of input values. The next stage operates on geometric primitives described by vertices: points, line segments, and polygons. In this stage vertices are transformed and lit, and 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, there is a way to bypass the vertex processing portion of the pipeline to send a block of fragments directly to the individual fragment operations, eventually causing a block of pixels to be written to the framebuffer; 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.
### GL Types and Bit Widths

<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 2’s complement binary integer</td>
</tr>
<tr>
<td>ubyte</td>
<td>8</td>
<td>Unsigned binary integer</td>
</tr>
<tr>
<td>char</td>
<td>8</td>
<td>Characters making up strings</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>Signed 2’s complement binary integer</td>
</tr>
<tr>
<td>ushort</td>
<td>16</td>
<td>Unsigned binary integer</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>Signed 2’s complement binary integer</td>
</tr>
<tr>
<td>uint</td>
<td>32</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 2’s complement binary integer</td>
</tr>
<tr>
<td>sizeiptr</td>
<td>ptrbits</td>
<td>Non-negative binary integer size</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>
<tr>
<td>time</td>
<td>64</td>
<td>Unsigned binary representing an absolute absolute or relative time interval. Precision is nanoseconds but accuracy is implementation-dependent.</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 and sizeiptr must be sufficiently large as to store any address.
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.
2.6. BEGIN/END PARADIGM

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.

- 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 Begin/End Paradigm

In the GL, most geometric objects are drawn by enclosing a series of coordinate sets that specify vertices and optionally normals, texture coordinates, and colors between `Begin/End` pairs. There are ten geometric objects that are drawn this way: points, line segments, line segment loops, separated line segments, polygons, triangle strips, triangle fans, separated triangles, quadrilateral strips, and separated quadrilaterals.

Each vertex is specified with two, three, or four coordinates. In addition, a `current normal`, multiple `current texture coordinate sets`, multiple `current generic vertex attributes`, `current color`, `current secondary color`, and `current fog coordinate` may be used in processing each vertex. Normals are used by the GL in lighting calculations; the current normal is a three-dimensional vector that may be set by sending three coordinates that specify it. Texture coordinates determine how
2.6. BEGIN/END PARADIGM

<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>STACK_OVERFLOW</td>
<td>Command would cause a stack overflow</td>
<td>Yes</td>
</tr>
<tr>
<td>STACK_UNDERFLOW</td>
<td>Command would cause a stack underflow</td>
<td>Yes</td>
</tr>
<tr>
<td>OUT_OF_MEMORY</td>
<td>Not enough memory left to execute command</td>
<td>Unknown</td>
</tr>
<tr>
<td>TABLE_TOO_LARGE</td>
<td>The specified table is too large</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 2.3: Summary of GL errors

A texture image is mapped onto a primitive. Multiple sets of texture coordinates may be used to specify how multiple texture images are mapped onto a primitive. The number of texture units supported is implementation dependent but must be at least two. The number of texture units supported can be queried with the state `MAX_TEXTURE_UNITS`. Generic vertex attributes can be accessed from within vertex shaders (section 2.20) and used to compute values for consumption by later processing stages.

Primary and secondary colors are associated with each vertex (see section 3.10). These associated colors are either based on the current color and current secondary color or produced by lighting, depending on whether or not lighting is enabled. Texture and fog coordinates are similarly associated with each vertex. Multiple sets of texture coordinates may be associated with a vertex. Figure 2.2 summarizes the association of auxiliary data with a transformed vertex to produce a processed vertex.

The current values are part of GL state. Vertices and normals are transformed, colors may be affected or replaced by lighting, and texture coordinates are transformed and possibly affected by a texture coordinate generation function. The processing indicated for each current value is applied for each vertex that is sent to the GL.

The methods by which vertices, normals, texture coordinates, fog coordinate,
2.6. BEGIN/END PARADIGM

Figure 2.2. Association of current values with a vertex. The heavy lined boxes represent GL state. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation.

Version 3.0 (with changes marked - August 11, 2008)
2.6. \textit{BEGIN/END PARADIGM}

Figure 2.3. Primitive assembly and processing.

generic attributes, and colors are sent to the GL, as well as how normals are transformed and how vertices are mapped to the two-dimensional screen, are discussed later.

Before colors have been assigned to a vertex, the state required by a vertex is the vertex’s coordinates, the current normal, the current edge flag (see section 2.6.2), the current material properties (see section 2.19.2), the current fog coordinate, the multiple generic vertex attribute sets, and the multiple current texture coordinate sets. Because color assignment is done vertex-by-vertex, a processed vertex comprises the vertex’s coordinates, its edge flag, its fog coordinate, its assigned colors, and its multiple texture coordinate sets.

Figure 2.3 shows the sequence of operations that builds a \textit{primitive} (point, line segment, or polygon) from a sequence of vertices. After a primitive is formed, it is clipped to a viewing volume. This may alter the primitive by altering vertex coordinates, texture coordinates, and colors. 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 texture coordinates and colors associated with them.

\subsection{Begin and End}

Vertices making up one of the supported geometric object types are specified by enclosing commands defining those vertices between the two commands

\begin{verbatim}
void Begin(enum mode);
\end{verbatim}
void End(void);

There is no limit on the number of vertices that may be specified between a Begin and an End.

**Points.** A series of individual points may be specified by calling Begin with an argument value of POINTS. No special state need be kept between Begin and End in this case, since each point is independent of previous and following points.

**Line Strips.** A series of one or more connected line segments is specified by enclosing a series of two or more endpoints within a Begin/End pair when Begin is called with 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 between the Begin/End pair, 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, specified with the LINE_LOOP argument value to Begin, are the same as line strips except that a final segment is added from the final specified vertex to the first vertex. The additional state consists of the processed first vertex.

**Separate Lines.** Individual line segments, each specified by a pair of vertices, are generated by surrounding vertex pairs with Begin and End when the value of the argument to Begin is LINES. In this case, the first two vertices between a Begin and End pair 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 lines but it is used differently: a vertex holding the first vertex of the current segment, and a boolean flag indicating whether the current vertex is odd or even (a segment start or end).

**Polygons.** A polygon is described by specifying its boundary as a series of line segments. When Begin is called with POLYGON, the bounding line segments are specified in the same way as line loops. Depending on the current state of the GL, a polygon may be rendered in one of several ways such as outlining its border or filling its interior. A polygon described with fewer than three vertices does not generate a primitive.

Only convex polygons are guaranteed to be drawn correctly by the GL. If a specified polygon is nonconvex when projected onto the window, then the rendered polygon need only lie within the convex hull of the projected vertices defining its boundary.
The state required to support polygons consists of at least two processed vertices (more than two are never required, although an implementation may use more); this is because a convex polygon can be rasterized as its vertices arrive, before all of them have been specified. The order of the vertices is significant in lighting and polygon rasterization (see sections 2.19.1 and 3.6.1).

**Triangle strips.** A triangle strip is a series of triangles connected along shared edges. A triangle strip is specified by giving a series of defining vertices between a **Begin/End** pair when **Begin** is called with **TRIANGLE_STRIP**. In this case, the first three vertices define the first triangle (and their order is significant, just as for polygons). Each subsequent vertex defines a new triangle using that point along with two vertices from the previous triangle. A **Begin/End** pair enclosing fewer than three vertices, when **TRIANGLE_STRIP** has been supplied to **Begin**, produces no primitive. See figure 2.4.

The state required to support triangle strips 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. After a **Begin** (**TRIANGLE_STRIP**), the pointer is initialized to point to vertex A. Each vertex sent between a **Begin/End** pair 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. The vertices of a triangle fan are enclosed between \texttt{Begin} and \texttt{End} when the value of the argument to \texttt{Begin} is \texttt{TRIANGLE\_FAN}.

Separate Triangles. Separate triangles are specified by placing vertices between \texttt{Begin} and \texttt{End} when the value of the argument to \texttt{Begin} is \texttt{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 between the \texttt{Begin} and \texttt{End}. $k$ 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.

The rules given for polygons also apply to each triangle generated from a triangle strip, triangle fan or from separate triangles.

Quadrilateral (quad) strips. Quad strips generate a series of edge-sharing quadrilaterals from vertices appearing between \texttt{Begin} and \texttt{End}, when \texttt{Begin} is called with \texttt{QUAD\_STRIP}. If the $m$ vertices between the \texttt{Begin} and \texttt{End} are $v_1, \ldots, v_m$, where $v_j$ is the $j$th specified vertex, then quad $i$ has vertices (in order) $v_{2i}$, $v_{2i+1}$, $v_{2i+3}$, and $v_{2i+2}$ with $i = 0, \ldots, \lfloor m/2 \rfloor$. The state required is thus three processed vertices, to store the last two vertices of the previous quad along with the third vertex (the first new vertex) of the current quad, a flag to indicate when the first quad has been completed, and a one-bit counter to count members of a vertex pair. See figure 2.5.

A quad strip with fewer than four vertices generates no primitive. If the number of vertices specified for a quadrilateral strip between \texttt{Begin} and \texttt{End} is odd, the
2.6. BEGIN/END PARADIGM

final vertex is ignored.

Separate Quadrilaterals Separate quads are just like quad strips except that each group of four vertices, the $4j + 1$st, the $4j + 2$nd, the $4j + 3$rd, and the $4j + 4$th, generate a single quad, for $j = 0, 1, \ldots, n - 1$. The total number of vertices between Begin and End is $4n + k$, where $0 \leq k \leq 3$; if $k$ is not zero, the final $k$ vertices are ignored. Separate quads are generated by calling Begin with the argument value QUADS.

The rules given for polygons also apply to each quad generated in a quad strip or from separate quads.

The state required for Begin and End consists of an eleven-valued integer indicating either one of the ten possible Begin/End modes, or that no Begin/End mode is being processed.

Calling Begin will result in an INVALID_FRAMEBUFFER_OPERATION error if the object bound to DRAW_FRAMEBUFFER_BINDING is not framebuffer complete (see section 4.4.4).

2.6.2 Polygon Edges

Each edge of each primitive generated from a polygon, triangle strip, triangle fan, separate triangle set, quadrilateral strip, or separate quadrilateral set, is flagged as either boundary or non-boundary. These classifications are used during polygon rasterization; some modes affect the interpretation of polygon boundary edges (see section 3.6.4). By default, all edges are boundary edges, but the flagging of polygons, separate triangles, or separate quadrilaterals may be altered by calling

```c
void EdgeFlag(boolean flag);
void EdgeFlagv(boolean *flag);
```

to change the value of a flag bit. If flag is zero, then the flag bit is set to FALSE; if flag is non-zero, then the flag bit is set to TRUE.

When Begin is supplied with one of the argument values POLYGON, TRIANGLES, or QUADS, each vertex specified within a Begin and End pair begins an edge. If the edge flag bit is TRUE, then each specified vertex begins an edge that is flagged as boundary. If the bit is FALSE, then induced edges are flagged as non-boundary.

The state required for edge flagging consists of one current flag bit. Initially, the bit is TRUE. In addition, each processed vertex of an assembled polygonal primitive must be augmented with a bit indicating whether or not the edge beginning on that vertex is boundary or non-boundary.
2.6.3 GL Commands within Begin/End

The only GL commands that are allowed within any Begin/End pairs are the commands for specifying vertex coordinates, vertex colors, normal coordinates, texture coordinates, generic vertex attributes, and fog coordinates (Vertex, Color, SecondaryColor, Index, Normal, TexCoord and MultiTexCoord, VertexAttrib, FogCoord), the ArrayElement command (see section 2.8), the EvalCoord and EvalPoint commands (see section 5.1), commands for specifying lighting material parameters (Material commands; see section 2.19.2), display list invocation commands (CallList and CallLists; see section 5.4), and the EdgeFlag command. Executing any other GL command between the execution of Begin and the corresponding execution of End results in the error INVALID_OPERATION. Executing Begin after Begin has already been executed but before an End is executed generates the INVALID_OPERATION error, as does executing End without a previous corresponding Begin.

Execution of the commands EnableClientState, DisableClientState, PushClientAttrib, PopClientAttrib, ColorPointer, FogCoordPointer, EdgeFlagPointer, IndexPointer, NormalPointer, TexCoordPointer, SecondaryColorPointer, VertexPointer, VertexAttribPointer, ClientActiveTexture, InterleavedArrays, and PixelStore is not allowed within any Begin/End pair, but an error may or may not be generated if such execution occurs. If an error is not generated, GL operation is undefined. (These commands are described in sections 2.8, 3.7.1, and chapter 6.)

2.7 Vertex Specification

Vertices are specified by giving their coordinates in two, three, or four dimensions. This is done using one of several versions of the Vertex command:

\[
\text{void Vertex\{234\}\{sifd\}(T coords);}
\]

\[
\text{void Vertex\{234\}\{sifd\}v(T coords);}
\]

A call to any Vertex command specifies four coordinates: \(x, y, z,\) and \(w\). The \(x\) coordinate is the first coordinate, \(y\) is second, \(z\) is third, and \(w\) is fourth. A call to Vertex2 sets the \(x\) and \(y\) coordinates; the \(z\) coordinate is implicitly set to zero and the \(w\) coordinate to one. Vertex3 sets \(x, y,\) and \(z\) to the provided values and \(w\) to one. Vertex4 sets all four coordinates, allowing the specification of an arbitrary point in projective three-space. Invoking a Vertex command outside of a Begin/End pair results in undefined behavior.
Current values are used in associating auxiliary data with a vertex as described in section 2.6. A current value may be changed at any time by issuing an appropriate command. The commands

```c
void TexCoord{1234}{sifd}( T coords );
void TexCoord{1234}{sifd}v( T coords );
```

specify the current homogeneous texture coordinates, named $s$, $t$, $r$, and $q$. The `TexCoord` family of commands set the $s$ coordinate to the provided single argument while setting $t$ and $r$ to 0 and $q$ to 1. Similarly, `TexCoord2` sets $s$ and $t$ to the specified values, $r$ to 0 and $q$ to 1; `TexCoord3` sets $s$, $t$, and $r$, with $q$ set to 1, and `TexCoord4` sets all four texture coordinates.

Implementations must support at least two sets of texture coordinates. The commands

```c
void MultiTexCoord{1234}{sifd}(enum texture, T coords )
void MultiTexCoord{1234}{sifd}v(enum texture, T coords )
```

take the coordinate set to be modified as the `texture` parameter. `texture` is a symbolic constant of the form `TEXTUREi`, indicating that texture coordinate set $i$ is to be modified. The constants obey $TEXTUREi = TEXTURE0 + i$ ($i$ is in the range 0 to $k - 1$, where $k$ is the implementation-dependent number of texture coordinate sets defined by `MAX_TEXTURE_COORDS`).

The `TexCoord` commands are exactly equivalent to the corresponding `MultiTexCoord` commands with `texture` set to `TEXTURE0`. The `TexCoord` commands are exactly equivalent to the corresponding `MultiTexCoord` commands with `texture` set to `TEXTURE0.`

`Get s of CURRENT_TEXTURE_COORDS` return the texture coordinate set defined by the value of `ACTIVE_TEXTURE`.

Specifying an invalid texture coordinate set for the `texture` argument of `MultiTexCoord` results in undefined behavior.

The current normal is set using

```c
void Normal3{bsifd}( T coords );
void Normal3{bsifd}v( T coords );
```

Byte, short, or integer values passed to `Normal` are converted to floating-point values as indicated for the corresponding (signed) type in table 2.10.

The current fog coordinate is set using

```c
void FogCoord{fd}( T coord );
void FogCoord{fd}v( T coord );
```

Version 3.0 (with changes marked - August 11, 2008)
There are several ways to set the current color and secondary color. The GL stores a current single-valued color index, as well as a current four-valued RGBA color and secondary color. Either the index or the color and secondary color are significant depending as the GL is in color index mode or RGBA mode. The mode selection is made when the GL is initialized.

The commands to set RGBA colors are

```c
void Color{34}{bsifd ubusui}( T components );
void Color{34}{bsifd ubusui}v( T components );
void SecondaryColor3{bsifd ubusui}( T components );
void SecondaryColor3{bsifd ubusui}v( T components );
```

The Color command has two major variants: Color3 and Color4. The four value versions set all four values. The three value versions set R, G, and B to the provided values; A is set to 1.0. (The conversion of integer color components (R, G, B, and A) to floating-point values is discussed in section 2.19.)

The secondary color has only the three value versions. Secondary A is always set to 1.0.

Versions of the Color and SecondaryColor commands that take floating-point values accept values nominally between 0.0 and 1.0. 0.0 corresponds to the minimum while 1.0 corresponds to the maximum (machine dependent) value that a component may take on in the framebuffer (see section 2.19 on colors and coloring). Values outside [0, 1] are not clamped.

The command

```c
void Index{sifd ub}( T index );
void Index{bsifd ub}v( T index );
```

updates the current (single-valued) color index. It takes one argument, the value to which the current color index should be set. Values outside the (machine-dependent) representable range of color indices are not clamped.

Vertex shaders (see section 2.20) can be written to access an array of 4-component generic vertex attributes in addition to the conventional attributes specified previously. 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`.

To load values into a generic shader attribute declared as a floating-point scalar, vector, or matrix, use the commands

```c
void VertexAttrib{1234}{sfd}( uint index, T values );
void VertexAttrib{123}{sfd}v( uint index, T values );
void VertexAttrib4{bsifd ubusui}v( uint index, T values );
```

Version 3.0 (with changes marked - August 11, 2008)
2.7. VERTEX SPECIFICATION

void VertexAttrib4Nub(uint index, T values);
void VertexAttrib4N{bsi ubusui}v( uint index, T values);

The VertexAttrib4N* commands specify fixed-point values that are converted
to a normalized [0, 1] or [−1, 1] range as shown in table 2.10, while the other commands specify values that are converted directly to the internal floating-point rep-
presentation.

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 de-
clared 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. Ma-
trices 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 at-
tribute at slot index is not floating-point (e.g. is signed or unsigned integer). To load values into a generic shader attribute declared as a signed or unsigned scalar or vector, use the commands

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 at-
tribute at slot index is floating-point; if the base type is integer and unsigned in-
teger values are supplied (the VertexAttribI*ui, VertexAttribI*us, and Vertex-
AttribI*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.

Setting generic vertex attribute zero specifies a vertex; the four vertex coordi-
nates are taken from the values of attribute zero. A Vertex2, Vertex3, or Vertex4
command is completely equivalent to the corresponding `VertexAttrib*` command with an index of zero. Setting any other generic vertex attribute updates the current values of the attribute. There are no current values for vertex attribute zero.

There is no aliasing among generic attributes and conventional attributes. In other words, an application can set all `MAX_VERTEX_ATTRIBS` generic attributes and all conventional attributes without fear of one particular attribute overwriting the value of another attribute.

The state required to support vertex specification consists of four floating-point numbers per texture coordinate set to store the current texture coordinates $s$, $t$, $r$, and $q$, three floating-point numbers to store the three coordinates of the current normal, one floating-point number to store the current fog coordinate, four floating-point values to store the current RGBA color, four floating-point values to store the current RGBA secondary color, one floating-point value to store the current color index, and `MAX_VERTEX_ATTRIBS` − 1 four-component floating-point vectors to store generic vertex attributes.

There is no notion of a current vertex, so no state is devoted to vertex coordinates or generic attribute zero. The initial texture coordinates are $(s, t, r, q) = (0, 0, 0, 1)$ for each texture coordinate set. The initial current normal has coordinates $(0, 0, 1)$. The initial fog coordinate is zero. The initial RGBA color is $(R, G, B, A) = (1, 1, 1, 1)$ and the initial RGBA secondary color is $(0, 0, 0, 1)$. The initial color index is 1. The initial values for all generic vertex attributes are $(0, 0, 0, 1)$.

### 2.8 Vertex Arrays

The vertex specification commands described in section 2.7 accept data in almost any format, but their use requires many command executions to specify even simple geometry. Vertex data may also be placed into arrays that are stored in the client’s address space. 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 seven plus the values of `MAX_TEXTURE_COORDS` and `MAX_VERTEX_ATTRIBS` arrays: one each to store vertex coordinates, normals, colors, secondary colors, color indices, edge flags, fog coordinates, two or more texture coordinate sets, and one or more generic vertex attributes. The commands

```c
void VertexPointer( int size, enum type, sizei stride, void *pointer );
void NormalPointer( enum type, sizei stride, void *pointer );
```

Version 3.0 (with changes marked - August 11, 2008)
void ColorPointer( int size, enum type, sizei stride,  
   void *pointer );
void SecondaryColorPointer( int size, enum type,  
   sizei stride, void *pointer );
void IndexPointer( enum type, sizei stride, void *pointer );
void EdgeFlagPointer(sizei stride, void *pointer);
void FogCoordPointer( enum type, sizei stride,  
   void *pointer );
void TexCoordPointer( int size, enum type, sizei stride,  
   void *pointer );
void VertexAttribPointer( uint index, int size, enum type,  
   boolean normalized, sizei stride, const  
   void *pointer );
gpu_shader4
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. Because edge flags are al-  
ways type boolean, EdgeFlagPointer has no type argument. size, when present,  
indicates the number of values per vertex that are stored in the array. Because  
normals are always specified with three values, NormalPointer has no size argument.  
Likewise, because color indices and edge flags are always specified with  
a single value, IndexPointer and EdgeFlagPointer also have no size argument.  
Table 2.4 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.

The index parameter in the VertexAttribPointer and VertexAttribIPointer  
commands identify the generic vertex attribute array being described. The error INVALID_VALUE is generated if index is greater than or equal to 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 specified in table 2.10, 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.
### 2.8. VERTEX ARRAYS

<table>
<thead>
<tr>
<th>Command</th>
<th>Sizes</th>
<th>Integer Handling</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>VertexPointer</td>
<td>2,3,4</td>
<td>cast</td>
<td>short, int, float, half, double</td>
</tr>
<tr>
<td>NormalPointer</td>
<td>3</td>
<td>normalize</td>
<td>byte, short, int, float, half, double</td>
</tr>
<tr>
<td>ColorPointer</td>
<td>3,4</td>
<td>normalize</td>
<td>byte, ubyte, short, ushort, int, uint, float, half, double</td>
</tr>
<tr>
<td>SecondaryColorPointer</td>
<td>3</td>
<td>normalize</td>
<td>byte, ubyte, short, ushort, int, uint, float, half, double</td>
</tr>
<tr>
<td>IndexPointer</td>
<td>1</td>
<td>cast</td>
<td>ubyte, short, int, float, double</td>
</tr>
<tr>
<td>FogCoordPointer</td>
<td>1</td>
<td>n/a</td>
<td>float, half, double</td>
</tr>
<tr>
<td>TexCoordPointer</td>
<td>1,2,3,4</td>
<td>cast</td>
<td>short, int, float, half, double</td>
</tr>
<tr>
<td>EdgeFlagPointer</td>
<td>1</td>
<td>integer</td>
<td>boolean</td>
</tr>
<tr>
<td>VertexAttribPointer</td>
<td>1,2,3,4</td>
<td>flag</td>
<td>byte, ubyte, short, ushort, int, uint, float, half, double</td>
</tr>
<tr>
<td>VertexAttribIPointer</td>
<td>1,2,3,4</td>
<td>integer</td>
<td>byte, ubyte, short, ushort, int, uint</td>
</tr>
</tbody>
</table>

Table 2.4: Vertex array sizes (values per vertex) and data types. The “Integer Handling” column indicates how fixed-point data types are handled: “cast” means that they converted to floating-point directly, “normalize” means that they are converted to floating-point by normalizing to [0, 1] (for unsigned types) or [−1, 1] (for signed types), “integer” means that they remain as integer values, and “flag” means that either “cast” or “normalized” applies, depending on the setting of the normalized flag in `VertexAttribPointer`. 
The one, two, three, or four values in an array that correspond to a single vertex comprise an array element. The values within each array element are stored sequentially in memory. 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 location in memory of the first value of the first element of the array being specified.

An individual array is enabled or disabled by calling one of

\[
\text{void EnableClientState( enum array );} \\
\text{void DisableClientState( enum array );}
\]

with array set to VERTEX_ARRAY, NORMAL_ARRAY, COLOR_ARRAY, SECONDARY_COLOR_ARRAY, INDEX_ARRAY, EDGE_FLAG_ARRAY, FOG_COORD_ARRAY, or TEXTURE_COORD_ARRAY, for the vertex, normal, color, secondary color, color index, edge flag, fog coordinate, or texture coordinate array, respectively.

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

\[
\text{void EnableVertexAttribArray( uint index );} \\
\text{void DisableVertexAttribArray( uint index );}
\]

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.

The command

\[
\text{void ClientActiveTexture( enum texture );}
\]

is used to select the vertex array client state parameters to be modified by the TexCoordPointer command and the array affected by EnableClientState and DisableClientState with parameter TEXTURE_COORD_ARRAY. This command sets the client state variable CLIENT_ACTIVE_TEXTURE. Each texture coordinate set has a client state vector which is selected when this command is invoked. This state vector includes the vertex array state. This call also selects the texture coordinate set state used for queries of client state.

Specifying an invalid texture generates the error INVALID_ENUM. Valid values of texture are the same as for the MultiTexCoord commands described in section 2.7.

The command

Version 3.0 (with changes marked - August 11, 2008)
void ArrayElement(int i);

transfers the \(i\)th element of every enabled array to the GL. The effect of
\texttt{ArrayElement}(i) is the same as the effect of the command sequence

\begin{verbatim}
gpu_shader4
if (normal array enabled)
    \texttt{Normal3[type]v(normal array element \(i\))};
if (color array enabled)
    \texttt{Color[size][type]v(color array element \(i\))};
if (secondary color array enabled)
    \texttt{SecondaryColor3[type]v(secondary color array element \(i\))};
if (fog coordinate array enabled)
    \texttt{FogCoord[type]v(fog coordinate array element \(i\))};
for (\(j = 0; j < \text{textureUnits}; j++\}) {
    if (texture coordinate set \(j\) array enabled)
        \texttt{MultiTexCoord[size][type]v(TEXTURE0 + \(j\), texture coordinate set \(j\) array element \(i\))};
if (color index array enabled)
    \texttt{Index[type]v(color index array element \(i\))};
if (edge flag array enabled)
    \texttt{EdgeFlagv(edge flag array element \(i\))};
for (\(j = 1; j < \text{genericAttributes}; j++\}) {
    if (generic vertex attribute \(j\) array enabled) {
        if (generic vertex attribute \(j\) array is a pure integer array)
            \texttt{VertexAttribI[size][type]v(\(j\), generic vertex attribute \(j\) array element \(i\))};
        else if (generic vertex attribute \(j\) array normalization flag is set, and
            \texttt{type} is not \texttt{FLOAT} or \texttt{DOUBLE})
            \texttt{VertexAttribN[type]v(\(j\), generic vertex attribute \(j\) array element \(i\))};
        else
            \texttt{VertexAttrib[size][type]v(\(j\), generic vertex attribute \(j\) array element \(i\))};
    }
}
if (generic vertex attribute array 0 enabled) {
    if (generic vertex attribute 0 array is a pure integer array)
        \texttt{VertexAttrib[0][size][type]v(0, generic vertex attribute 0 array element \(i\))};
    else if (generic vertex attribute 0 array normalization flag is set, and
        \texttt{type} is not \texttt{FLOAT} or \texttt{DOUBLE})
        \texttt{VertexAttribN[type]v(0, generic vertex attribute 0 array element \(i\))};
    else
        \texttt{VertexAttrib[size][type]v(0, generic vertex attribute 0 array element \(i\))};
} else if (vertex array enabled) {

    \texttt{Version 3.0 (with changes marked - August 11, 2008}
\end{verbatim}
2.8. VERTEX ARRAYS

\[ \text{Vertex}[\text{size}][\text{type}]v(\text{vertex array element } i); \]

where \text{textureUnits} and \text{genericAttributes} give the number of texture coordinate sets and generic vertex attributes supported by the implementation, respectively. "[size]" and "[type]" correspond to the size and type of the corresponding array. For generic vertex attributes, it is assumed that a complete set of vertex attribute commands exists, even though not all such functions are provided by the GL.

Changes made to array data between the execution of \text{Begin} and the corresponding execution of \text{End} may affect calls to \text{ArrayElement} that are made within the same \text{Begin}/\text{End} period in non-sequential ways. That is, a call to \text{ArrayElement} that precedes a change to array data may access the changed data, and a call that follows a change to array data may access original data.

Specifying \( i < 0 \) results in undefined behavior. Generating the error \text{INVALID \_VALUE} is recommended in this case.

The command

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

constructs a sequence of geometric primitives using elements \( \text{first} \) through \( \text{first} + \text{count} - 1 \) of each enabled array. \( \text{mode} \) specifies what kind of primitives are constructed; it accepts the same token values as the \( \text{mode} \) parameter of the \text{Begin} command. The effect of

\[
\text{DrawArrays (mode, first, count );}
\]

is the same as the effect of the command sequence

\[
\text{if ( mode or count is invalid )}
\]
\[
\text{generate appropriate error}
\]
\[
\text{else} \{
\text{Begin (mode);}
\text{for (int i = 0; i < count; i++)}
\text{ArrayElement (first + i);}
\text{End();}
\}
\]

with one exception: the current normal coordinates, color, secondary color, color index, edge flag, fog coordinate, texture coordinates, and generic attributes are each indeterminate after execution of \text{DrawArrays}, if the corresponding array is enabled. Current values corresponding to disabled arrays are not modified by the execution of \text{DrawArrays}.

Version 3.0 (with changes marked - August 11, 2008)
2.8. VERTEX ARRAYS

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

The command

\begin{verbatim}
void MultiDrawArrays (enum mode, int *first, sizei *count, sizei primcount);
\end{verbatim}

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

\begin{verbatim}
for (i = 0; i < primcount; i++) {
  if (count[i] > 0)
    DrawArrays (mode, first[i], count[i]);
}
\end{verbatim}

The command

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

constructs a sequence of geometric primitives using the \texttt{count} elements whose indices are stored in \texttt{indices}. \texttt{type} must be one of \texttt{UNSIGNED_BYTE}, \texttt{UNSIGNED_SHORT}, or \texttt{UNSIGNED_INT}, indicating that the values in \texttt{indices} are indices of GL type \texttt{ubyte}, \texttt{ushort}, or \texttt{uint} respectively. \texttt{mode} specifies what kind of primitives are constructed; it accepts the same token values as the \texttt{mode} parameter of the \texttt{Begin} command. The effect of

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

is the same as the effect of the command sequence

\begin{verbatim}
if (mode, count, or type is invalid )
generate appropriate error
else {
  Begin (mode);
  for (int i = 0; i < count; i++)
    ArrayElement (indices[i]);
  End();
}
\end{verbatim}

Version 3.0 (with changes marked - August 11, 2008)
with one exception: the current normal coordinates, color, secondary color, color index, edge flag, fog coordinate, texture coordinates, and generic attributes are each indeterminate after the execution of \texttt{DrawElements}, if the corresponding array is enabled. Current values corresponding to disabled arrays are not modified by the execution of \texttt{DrawElements}.

The command

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

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

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

The command

\begin{verbatim}
void DrawRangeElements( enum mode, uint start,
                        uint end, sizei count, enum type, void *indices);
\end{verbatim}

is a restricted form of \texttt{DrawElements}. \textit{mode}, \textit{count}, \textit{type}, and \textit{indices} match the corresponding arguments to \texttt{DrawElements}, with the additional constraint that all values in the array \textit{indices} must lie between \textit{start} and \textit{end} inclusive.

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

The error \texttt{INVALID\_VALUE} is generated if \textit{end} \textit{start}. Invalid \textit{mode}, \textit{count}, or \textit{type} parameters generate the same errors as would the corresponding call to \texttt{DrawElements}. It is an error for indices to lie outside the range \texttt{[start,end]}, but implementations may not check for this. Such indices will cause implementation-dependent behavior.

The command

\begin{verbatim}
Version 3.0 (with changes marked - August 11, 2008
\end{verbatim}
void InterleavedArrays(enum format, sizei stride, void *pointer);

efficiently initializes the six arrays and their enables to one of 14 con-

The effect of

InterleavedArrays(format, stride, pointer);

is the same as the effect of the command sequence

if (format or stride is invalid)
  generate appropriate error
else {
  int str;
  set et, ec, en, st, sc, sv, tc, pc, pn, pv, and s as a function
  of table 2.5 and the value of format.
  str = stride;
  if (str is zero)
    str = s;
  DisableClientState(EDGE_FLAG_ARRAY);
  DisableClientState(INDEX_ARRAY);
  DisableClientState(SECONDARY_COLOR_ARRAY);
  DisableClientState(FOG_COORD_ARRAY);
  if (et) {
    EnableClientState(TEXTURE_COORD_ARRAY);
   TexCoordPointer(st, FLOAT, str, pointer);
  } else
    DisableClientState(TEXTURE_COORD_ARRAY);
  if (ec) {
    EnableClientState(COLOR_ARRAY);
    ColorPointer(sc, tc, str, pointer + pc);
  } else
    DisableClientState(COLOR_ARRAY);
  if (en) {
    EnableClientState(NORMAL_ARRAY);
    NormalPointer(FLOAT, str, pointer + pn);
  } else
    DisableClientState(NORMAL_ARRAY);
  else
    DisableClientState(T2F_V3F);
### Table 2.5: Variables that direct the execution of InterleavedArrays

<table>
<thead>
<tr>
<th>format</th>
<th>$e_t$</th>
<th>$e_c$</th>
<th>$e_n$</th>
<th>$s_t$</th>
<th>$s_c$</th>
<th>$s_v$</th>
<th>$t_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>V2F</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V3F</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C4UB_V2F</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>4</td>
<td>2</td>
<td></td>
<td>UNSIGNED_BYTE</td>
</tr>
<tr>
<td>C4UB_V3F</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>4</td>
<td>3</td>
<td></td>
<td>UNSIGNED_BYTE</td>
</tr>
<tr>
<td>C3F_V3F</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>3</td>
<td>3</td>
<td></td>
<td>FLOAT</td>
</tr>
<tr>
<td>N3F_V3F</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C4F_N3F_V3F</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>4</td>
<td>3</td>
<td></td>
<td>FLOAT</td>
</tr>
<tr>
<td>T2F_V3F</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T4F_V4F</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2F_C4UB_V3F</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>UNSIGNED_BYTE</td>
</tr>
<tr>
<td>T2F_C3F_V3F</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>FLOAT</td>
</tr>
<tr>
<td>T2F_N3F_V3F</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2F_C4F_N3F_V3F</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>FLOAT</td>
</tr>
<tr>
<td>T4F_C4F_N3F_V4F</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>FLOAT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>format</th>
<th>$p_c$</th>
<th>$p_n$</th>
<th>$p_v$</th>
<th>$s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>V2F</td>
<td>0</td>
<td>0</td>
<td>2f</td>
<td></td>
</tr>
<tr>
<td>V3F</td>
<td>0</td>
<td>c</td>
<td>c + 2f</td>
<td></td>
</tr>
<tr>
<td>C4UB_V2F</td>
<td>0</td>
<td>c</td>
<td>c + 3f</td>
<td></td>
</tr>
<tr>
<td>C4UB_V3F</td>
<td>0</td>
<td>3f</td>
<td></td>
<td>6f</td>
</tr>
<tr>
<td>C3F_V3F</td>
<td>0</td>
<td>3f</td>
<td></td>
<td>6f</td>
</tr>
<tr>
<td>N3F_V3F</td>
<td>4f</td>
<td>7f</td>
<td>10f</td>
<td></td>
</tr>
<tr>
<td>C4F_N3F_V3F</td>
<td>2f</td>
<td>5f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2F_V3F</td>
<td>2f</td>
<td>4f</td>
<td>8f</td>
<td></td>
</tr>
<tr>
<td>T4F_V4F</td>
<td>2f</td>
<td>c + 2f</td>
<td>c + 5f</td>
<td></td>
</tr>
<tr>
<td>T2F_C4UB_V3F</td>
<td>2f</td>
<td>5f</td>
<td>8f</td>
<td></td>
</tr>
<tr>
<td>T2F_C3F_V3F</td>
<td>2f</td>
<td>5f</td>
<td>8f</td>
<td></td>
</tr>
<tr>
<td>T2F_N3F_V3F</td>
<td>2f</td>
<td>6f</td>
<td>9f</td>
<td>12f</td>
</tr>
<tr>
<td>T2F_C4F_N3F_V3F</td>
<td>2f</td>
<td>6f</td>
<td>9f</td>
<td>12f</td>
</tr>
<tr>
<td>T4F_C4F_N3F_V4F</td>
<td>4f</td>
<td>8f</td>
<td>11f</td>
<td>15f</td>
</tr>
</tbody>
</table>

Table 2.5: Variables that direct the execution of InterleavedArrays. $f$ is `sizeof(FLOAT)`. $c$ is 4 times `sizeof(UNSIGNED_BYTE)`, rounded up to the nearest multiple of $f$. All pointer arithmetic is performed in units of `sizeof(UNSIGNED_BYTE)`.

Version 3.0 (with changes marked - August 11, 2008)
2.9. BUFFER OBJECTS

    DisableClientState(NORMAL_ARRAY);
    EnableClientState(VERTEX_ARRAY);
    VertexPointer(sv, FLOAT, str, pointer + pv);
}

If the number of supported texture units (the value of MAX_TEXTURE_COORDS) is \( m \) and 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 an integer for the client active texture unit selector, \( 7 + m + n \) boolean values, \( 7 + m + n \) memory pointers, \( 7 + m + n \) integer stride values, \( 7 + m + n \) symbolic constants representing array types, \( 3 + m + n \) integers representing values per element, \( n \) boolean values indicating normalization, and \( n \) boolean values indicating whether the attribute values are pure integers.

In the initial state, the client active texture unit selector is TEXTURE0, 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, and the normalized and pure integer flags are each false.

2.9 Buffer Objects

The vertex data arrays described in section 2.8 are stored in client memory. It is sometimes desirable to store frequently used client data, such as vertex array and pixel data, in high-performance server memory. GL buffer objects provide a mechanism that clients can use to allocate, initialize, and render from such memory.

The name space for buffer objects is the unsigned integers, with zero reserved for the GL. A buffer object is created by binding an unused name to a buffer target. The binding is effected by calling

    void BindBuffer( enum target, uint buffer );

target must be one of ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER, PIXEL_UNPACK_BUFFER, or PIXEL_PACK_BUFFER. The ARRAY_BUFFER target is discussed in section 2.9.1. The ELEMENT_ARRAY_BUFFER target is discussed in section 2.9.2. The PIXEL_UNPACK_BUFFER and PIXEL_PACK_BUFFER targets are discussed later in sections 3.7, 4.3.2, and 6.1. If the buffer object named buffer has not been previously bound or has been deleted since the last binding, the GL creates a new state vector, initialized with a zero-sized memory buffer and comprising the state values listed in table 2.6.

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.
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.

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.

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.

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, just as if they were unused.

While a buffer object is bound, any GL operations on that object 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 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.

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

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

with `target` set to one of `ARRAY_BUFFER`, `ELEMENT_ARRAY_BUFFER`, `PIXEL_UNPACK_BUFFER`, or `PIXEL_PACK_BUFFER`, `size` set to the size of the data store in basic machine units, and `data` pointing to the source data in client memory. If `data` is non-null, then the source data is copied to the buffer object’s data store. If `data` is null, then the contents of the buffer object’s data store are undefined.

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

- **STREAM_DRAW** The data store contents will be specified once by the application, and used at most a few times as the source for GL drawing and image specification commands.
- **STREAM_READ** The data store contents will be specified once by reading data from the GL, and queried at most a few times by the application.
- **STREAM_COPY** The data store contents will be specified once by reading data from the GL, and used at most a few times as the source for GL drawing and image specification commands.
- **STATIC_DRAW** The data store contents will be specified once by the application, and used many times as the source for GL drawing and image specification commands.
- **STATIC_READ** The data store contents will be specified once by reading data from the GL, and queried many times by the application.
- **STATIC_COPY** The data store contents will be specified once by reading data from the GL, and used many times as the source for GL drawing and image specification commands.
- **DYNAMIC_DRAW** The data store contents will be respecified repeatedly by the application, and used many times as the source for GL drawing and image specification commands.

Version 3.0 (with changes marked - August 11, 2008)
### 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_MAPPED</td>
<td>FALSE</td>
</tr>
<tr>
<td>BUFFER_MAP_POINTER</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Table 2.7: Buffer object initial state.

**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.

**Buffer** deletes any existing data store, and sets the values of the buffer object’s state variables as shown in table 2.7.

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);
```

with *target* set to **ARRAY_BUFFER**. *offset* and *size* indicate the range of data in the buffer object that is to be replaced, in terms of basic machine units. *data* specifies a region of client memory *size* basic machine units in length, containing the data that replace the specified buffer range. An **INVALID_VALUE** error is generated if *offset* or *size* is less than zero, or if *offset* + *size* is greater than the value of **BUFFER_SIZE**.

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);
```
2.9. BUFFER OBJECTS

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFFER_ACCESS</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>
</tbody>
</table>

Table 2.8: Buffer object state set by `MapBuffer`.

with `target` set to one of `ARRAY_BUFFER`, `ELEMENT_ARRAY_BUFFER`, `PIXEL_UNPACK_BUFFER`, or `PIXEL_PACK_BUFFER`. If the GL is able to map the buffer object’s data store into the client’s address space, `MapBuffer` returns the pointer value to the data store once all pending operations on that buffer have completed. If the buffer data store is already in the mapped state, `MapBuffer` returns NULL, and an `INVALID_OPERATION` error is generated. Otherwise `MapBuffer` returns NULL, and the error `OUT_OF_MEMORY` is generated. `access` is specified as one of `READ_ONLY`, `WRITE_ONLY`, or `READ_WRITE`, indicating the operations that the client may perform on the data store through the pointer while the data store is mapped.

`MapBuffer` sets buffer object state values as shown in table 2.8.

Non-NULL pointers returned by `MapBuffer` may be used by the client to modify and query buffer object data, consistent with the access rules of the mapping, while the mapping remains valid. No GL error is generated if the pointer is used to attempt to modify a `READ_ONLY` data store, or to attempt to read from a `WRITE_ONLY` data store, but operation may be slow and system errors (possibly including program termination) may result. Pointer values returned by `MapBuffer` 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.

Calling `BufferSubData` to modify the data store of a mapped buffer will generate an `INVALID_OPERATION` error.

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 `BUFFER_ACCESS`. Using a mapping in a fashion inconsistent with these values is liable to be multiple orders of magnitude slower than using normal memory.

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

   boolean UnmapBuffer( enum target );

with target set to one of ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER, PIXEL_UNPACK_BUFFER, or PIXEL_PACK_BUFFER. Unmapping a mapped buffer object invalidates the pointers to its data store and sets the object’s BUFFER_MAPPED state to FALSE and its BUFFER_MAP_POINTER state to NULL.

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.

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

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

with target set to one of ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER, PIXEL_UNPACK_BUFFER, or PIXEL_PACK_BUFFER. The offset and length indicate the range of data in the buffer object that is to be mapped, in terms of basic machine units. access is a bitfield containing flags which describe the requested mapping. These flags are described below.

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

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

Version 3.0 (with changes marked - August 11, 2008)
2.9. BUFFER OBJECTS

- 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.

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.

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 exceeds the size of the buffer, if offset + length exceeds the maximum allowed range for the mapping, or if MAP_INVALIDATE_BUFFER_BIT or MAP_INVALIDATE_RANGE_BIT are set and the specified region of the buffer overlaps the mapping.

Bug 3572
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, Bug 3582 MAP_INVALIDATE_BUFFER_BIT, or MAP_UNSYNCHRONIZED_BIT is set.
- MAP_FLUSH_EXPLICIT_BIT is set and MAP_WRITE_BIT is not set.

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.

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 ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER, PIXEL_UNPACK_BUFFER, or PIXEL_PACK_BUFFER. 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 buffer is not mapped, or is mapped without the MAP_FLUSH_EXPLICIT_BIT flag.

### 2.9.1 Vertex Arrays in Buffer Objects

Blocks of vertex array data may be stored in buffer objects with the same format and layout options supported for client-side vertex arrays. However, it is expected...
that GL implementations will (at minimum) be optimized for data with all components represented as floats, as well as for color data with components represented as either floats or unsigned bytes.

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 NormalPointer 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 NORMAL_ARRAY_BUFFER_BINDING.

Rendering commands ArrayElement, DrawArrays, DrawElements, DrawRangeElements, MultiDrawArrays, and MultiDrawElements operate as previously defined, except that data for enabled vertex and attrib arrays are sourced from buffers if the array’s buffer binding is non-zero. 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.

It is acceptable for vertex or attrib arrays to be sourced from any combination of client memory and various buffer objects during a single rendering operation. Any GL command that attempts to read data from a buffer object will fail and generate an INVALID_OPERATION error if the object is mapped at the time the command is issued.

2.9.2 Array Indices in Buffer Objects

Blocks of array indices may be stored in buffer objects with the same format options that are supported for client-side index arrays. Initially zero is bound to ELEMENT_ARRAY_BUFFER, indicating that DrawElements and DrawRangeElements are to source their indices from arrays passed as their indices parameters, and that MultiDrawElements is to source its indices from the array of pointers to arrays passed in as its indices parameter.

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.

While a non-zero buffer object name is bound to ELEMENT_ARRAY_BUFFER, DrawElements and DrawRangeElements source their indices from that buffer object, using their indices parameters as offsets into the buffer object in the same...
2.10. VERTEX ARRAY OBJECTS

fashion as described in section 2.9.1. **MultiDrawElements** also sources its indices from that buffer object, using its *indices* parameter as a pointer to an array of pointers that represent offsets into the buffer object.

Buffer objects created by binding an unused name to `ARRAY_BUFFER` and to `ELEMENT_ARRAY_BUFFER` are formally equivalent, but the GL may make different choices about storage implementation based on the initial binding. 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.

2.9.3 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 array, normal array, color array, index array, multiple texture coordinate arrays, edge flag array, secondary color array, fog coordinate array, and 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, 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 `arrays`. These names are marked as used, for the purposes of **GenVertexArrays** only, and are initialized with the state listed in tables 6.6 through 6.9.

Vertex array objects are deleted by calling

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

`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 is deleted it has no contents and its name is again unused.

Version 3.0 (with changes marked - August 11, 2008)
There is a set of GL commands to support efficient specification of rectangles as two corner vertices.

```c
void Rect(sifd)( T x1, T y1, T x2, T y2);
void Rect(sifd)v( T v1[2], T v2[2]);
```

Each command takes either four arguments organized as two consecutive pairs of \((x, y)\) coordinates, or two pointers to arrays each of which contains an \(x\) value followed by a \(y\) value. The effect of the `Rect` command

\[
\text{Rect} \ (x_1, y_1, x_2, y_2);
\]

is exactly the same as the following sequence of commands:

---

2. This error makes it impossible to create a vertex array object containing client array pointers.
2.12 Coordinate Transformations

This section and the following discussion through section 2.19 describe the state values and operations necessary for transforming vertex attributes according to a fixed-functionality method. An alternate programmable method for transforming vertex attributes is described in section 2.20.

Vertices, normals, and texture coordinates are transformed before their coordinates are used to produce an image in the framebuffer. We begin with a description of how vertex coordinates are transformed and how this transformation is controlled.

Figure 2.6 diagrams the sequence of transformations that are applied to vertices. The vertex coordinates that are presented to the GL are termed object coordinates. The model-view matrix is applied to these coordinates to yield eye coordinates. Then another matrix, called the projection matrix, is applied to eye coordinates to yield clip coordinates. A perspective division is carried out on clip coordinates to yield normalized device coordinates. A final viewport transformation is applied to convert these coordinates into window coordinates.

Object coordinates, eye coordinates, and clip coordinates are four-dimensional, consisting of \( x, y, z, \) and \( w \) coordinates (in that order). The model-view and projection matrices are thus \( 4 \times 4 \).

If a vertex in object coordinates is given by

\[
\begin{pmatrix}
x_o \\
y_o \\
z_o \\
w_o
\end{pmatrix}
\]

and the model-view matrix is \( M \), then the vertex’s eye coordinates are found as

\[
\begin{pmatrix}
x_e \\
y_e \\
z_e \\
w_e
\end{pmatrix} = M \begin{pmatrix}
x_o \\
y_o \\
z_o \\
w_o
\end{pmatrix}.
\]
2.12. COORDINATE TRANSFORMATIONS

Similarly, if $P$ is the projection matrix, then the vertex’s clip coordinates are

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = P \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix}.$$

The vertex’s normalized device coordinates are then

$$\begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} = P \begin{pmatrix} x_c \\ w_c \\ y_c \\ w_c \\ z_c \\ w_c \end{pmatrix}.$$

2.12.1 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

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

Version 3.0 (with changes marked - August 11, 2008)
The factor and offset applied to \( z_d \) encoded by \( n \) and \( f \) are set using

\[
\text{void DepthRange}(\text{clampd } n, \text{clampd } f);
\]

\( z_w \) 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

\[
\text{void Viewport}(\text{int } x, \text{int } y, \text{sizei } w, \text{sizei } h);
\]

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 \( o_x = x + w/2 \) and \( o_y = y + h/2 \); \( px = w, py = h \).

Viewport width and height are clamped to implementation-dependent maximums when specified. The maximum width and height may be found by issuing an appropriate \text{Get} command (see chapter 6). The maximum viewport dimensions must be greater than or equal to the visible dimensions of the display being rendered to. \text{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. \( o_x \) and \( o_y \) are set to \( w/2 \) and \( h/2 \), respectively. \( n \) and \( f \) are set to 0.0 and 1.0, respectively.

### 2.12.2 Matrices

The projection matrix and model-view matrix are set and modified with a variety of commands. The affected matrix is determined by the current matrix mode. The current matrix mode is set with

\[
\text{void MatrixMode}(\text{enum } mode);
\]

which takes one of the pre-defined constants \text{TEXTURE}, \text{MODELVIEW}, \text{COLOR}, or \text{PROJECTION} as the argument value. \text{TEXTURE} is described later in section 2.12.2, and \text{COLOR} is described in section 3.7.3. If the current matrix mode is \text{MODELVIEW}, then matrix operations apply to the model-view matrix; if \text{PROJECTION}, then they apply to the projection matrix.

The two basic commands for affecting the current matrix are
void LoadMatrix{fd}(T m[16]);
void MultMatrix{fd}(T m[16]);

LoadMatrix takes a pointer to a $4 \times 4$ matrix stored in column-major order as 16 consecutive floating-point values, i.e. as

$$
\begin{pmatrix}
  a_1 & a_5 & a_9 & a_{13} \\
  a_2 & a_6 & a_{10} & a_{14} \\
  a_3 & a_7 & a_{11} & a_{15} \\
  a_4 & a_8 & a_{12} & a_{16}
\end{pmatrix}.
$$

(This differs from the standard row-major C ordering for matrix elements. If the standard ordering is used, all of the subsequent transformation equations are transposed, and the columns representing vectors become rows.)

The specified matrix replaces the current matrix with the one pointed to. MultMatrix takes the same type argument as LoadMatrix, but multiplies the current matrix by the one pointed to and replaces the current matrix with the product. If $C$ is the current matrix and $M$ is the matrix pointed to by MultMatrix's argument, then the resulting current matrix, $C'$, is

$$
C' = C \cdot M.
$$

The commands

void LoadTransposeMatrix{fd}(T m[16]);
void MultTransposeMatrix{fd}(T m[16]);

take pointers to $4 \times 4$ matrices stored in row-major order as 16 consecutive floating-point values, i.e. as

$$
\begin{pmatrix}
  a_1 & a_2 & a_3 & a_4 \\
  a_5 & a_6 & a_7 & a_8 \\
  a_9 & a_{10} & a_{11} & a_{12} \\
  a_{13} & a_{14} & a_{15} & a_{16}
\end{pmatrix}.
$$

The effect of

LoadTransposeMatrix{fd}(m);

is the same as the effect of

LoadMatrix{fd}(m^T);

The effect of

LoadMatrix{fd}(m);
2.12. COORDINATE TRANSFORMATIONS

\texttt{MultTransposeMatrix[fd]}(m) ;

is the same as the effect of

\texttt{MultMatrix[fd]}(m^T) ;

The command

\texttt{void LoadIdentity( void);} 

effectively calls \texttt{LoadMatrix} with the identity matrix:

\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}

There are a variety of other commands that manipulate matrices. \texttt{Rotate}, \texttt{Translate}, \texttt{Scale}, \texttt{Frustum}, and \texttt{Ortho} manipulate the current matrix. Each computes a matrix and then invokes \texttt{MultMatrix} with this matrix. In the case of

\texttt{void Rotate\{fd\}( T \theta, T x, T y, T z);} 

\(\theta\) gives an angle of rotation in degrees; the coordinates of a vector \(v\) are given by \(v = \begin{pmatrix} x & y & z \end{pmatrix}^T\). The computed matrix is a counter-clockwise rotation about the line through the origin with the specified axis when that axis is pointing up (i.e. the right-hand rule determines the sense of the rotation angle). The matrix is thus

\begin{pmatrix}
0 & \quad & \quad & \\
R & \quad & \quad \\
0 & \quad & \quad \\
0 & \quad & \quad \\
\end{pmatrix}

Let \(u = v/||v|| = \begin{pmatrix} x' & y' & z' \end{pmatrix}^T\). If

\begin{pmatrix}
\begin{pmatrix} 0 & -z' & y' \\
z' & 0 & -x' \\
y' & 0 & x' \\
\end{pmatrix}
\end{pmatrix}

then

\[ R = uu^T + \cos \theta (I - uu^T) + \sin \theta S. \]

The arguments to

Version 3.0 (with changes marked - August 11, 2008)
2.12. COORDINATE TRANSFORMATIONS

void Translate(fd)(T x, T y, T z);

give the coordinates of a translation vector as \((x \ y \ z)^T\). The resulting matrix is a translation by the specified vector:

\[
\begin{pmatrix}
1 & 0 & 0 & x \\
0 & 1 & 0 & y \\
0 & 0 & 1 & z \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

void Scale(fd)(T x, T y, T z);

produces a general scaling along the \(x\)-, \(y\)-, and \(z\)-axes. The corresponding matrix is

\[
\begin{pmatrix}
x & 0 & 0 & 0 \\
0 & y & 0 & 0 \\
0 & 0 & z & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

For

void Frustum(double l, double r, double b, double t, double n, double f);

the coordinates \((l \ b - n)^T\) and \((r \ t - n)^T\) specify the points on the near clipping plane that are mapped to the lower left and upper right corners of the window, respectively (assuming that the eye is located at \((0 \ 0 \ 0)^T\)). \(f\) gives the distance from the eye to the far clipping plane. If either \(n\) or \(f\) is less than or equal to zero, \(l\) is equal to \(r\), \(b\) is equal to \(t\), or \(n\) is equal to \(f\), the error INVALID_VALUE results. The corresponding matrix is

\[
\begin{pmatrix}
\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\
0 & \frac{2n}{t-b} & \frac{r+b}{t-b} & 0 \\
0 & 0 & \frac{-f+n}{f-n} & -\frac{2fn}{f-n} \\
0 & 0 & -1 & 0
\end{pmatrix}
\]

void Ortho(double l, double r, double b, double t, double n, double f);

describes a matrix that produces parallel projection. \((l \ b - n)^T\) and \((r \ t - n)^T\) specify the points on the near clipping plane that are mapped to the lower left and upper right corners of the window, respectively. \(f\) gives the distance from the eye.
to the far clipping plane. If \( l \) is equal to \( r \), \( b \) is equal to \( t \), or \( n \) is equal to \( f \), the error INVALID VALUE results. The corresponding matrix is

\[
\begin{pmatrix}
\frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\
0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\
0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\
0 & 0 & 0 & 1
\end{pmatrix}.
\]

For each texture coordinate set, a \( 4 \times 4 \) matrix is applied to the corresponding texture coordinates. This matrix is applied as

\[
\begin{pmatrix}
m_{1} & m_{5} & m_{9} & m_{13} \\
m_{2} & m_{6} & m_{10} & m_{14} \\
m_{3} & m_{7} & m_{11} & m_{15} \\
m_{4} & m_{8} & m_{12} & m_{16}
\end{pmatrix}
\begin{pmatrix}
s \\
t \\
r \\
g
\end{pmatrix},
\]

where the left matrix is the current texture matrix. The matrix is applied to the coordinates resulting from texture coordinate generation (which may simply be the current texture coordinates), and the resulting transformed coordinates become the texture coordinates associated with a vertex. Setting the matrix mode to TEXTURE causes the already described matrix operations to apply to the texture matrix.

The command

```c
void ActiveTexture( enum texture );
```

specifies the active texture unit selector, ACTIVE_TEXTURE. Each texture unit contains up to two distinct sub-units: a texture coordinate processing unit (consisting of a texture matrix stack and texture coordinate generation state) and a texture image unit (consisting of all the texture state defined in section 3.9). In implementations with a different number of supported texture coordinate sets and texture image units, some texture units may consist of only one of the two sub-units.

The active texture unit selector specifies the texture coordinate set accessed by commands involving texture coordinate processing. Such commands include those accessing the current matrix stack (if MATRIX_MODE is TEXTURE), TexEnv commands controlling point sprite coordinate replacement (see section 3.4), TexGen (section 2.12.4), Enable/Disable (if any texture coordinate generation enum is selected), as well as queries of the current texture coordinates and current raster texture coordinates. If the texture coordinate set number corresponding to the current value of ACTIVE_TEXTURE is greater than or equal to the implementation-dependent constant MAX_TEXTURE_COORDS, the error INVALID_OPERATION is generated by any such command.

Version 3.0 (with changes marked - August 11, 2008)
The active texture unit selector also selects the texture image unit accessed by commands involving texture image processing (section 3.9). Such commands include all variants of TexEnv (except for those controlling point sprite coordinate replacement), TexParameter, andTexImage commands, Enable/Disable for any texture target (e.g., TEXTURE_2D), and queries of all such state. If the texture image unit number corresponding to the current value of ACTIVE_TEXTURE is greater than or equal to the implementation-dependent constant MAX_COMBINED_TEXTURE_IMAGE_UNITS, the error INVALID_OPERATION is generated by any such command.

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 larger of MAX_TEXTURE_COORDS and MAX_COMBINED_TEXTURE_IMAGE_UNITS).

For backwards compatibility, the implementation-dependent constant MAX_TEXTURE_UNITS specifies the number of conventional texture units supported by the implementation. Its value must be no larger than the minimum of MAX_TEXTURE_COORDS and MAX_COMBINED_TEXTURE_IMAGE_UNITS.

There is a stack of matrices for each of matrix modes MODELVIEW, PROJECTION, and COLOR, and for each texture unit. For MODELVIEW mode, the stack depth is at least 32 (that is, there is a stack of at least 32 model-view matrices). For the other modes, the depth is at least 2. Texture matrix stacks for all texture units have the same depth. The current matrix in any mode is the matrix on the top of the stack for that mode.

```c
void PushMatrix( void );
```

pushes the stack down by one, duplicating the current matrix in both the top of the stack and the entry below it.

```c
void PopMatrix( void );
```

pops the top entry off of the stack, replacing the current matrix with the matrix that was the second entry in the stack. The pushing or popping takes place on the stack corresponding to the current matrix mode. Popping a matrix off a stack with only one entry generates the error STACK_UNDERFLOW; pushing a matrix onto a full stack generates STACK_OVERFLOW.

When the current matrix mode is TEXTURE, the texture matrix stack of the active texture unit is pushed or popped.

The state required to implement transformations consists of an integer for the active texture unit selector, a four-valued integer indicating the current matrix
mode, one stack of at least two $4 \times 4$ matrices for each of COLOR, PROJECTION, and each texture coordinate set, TEXTURE; and a stack of at least 32 $4 \times 4$ matrices for MODELVIEW. Each matrix stack has an associated stack pointer. Initially, there is only one matrix on each stack, and all matrices are set to the identity. The initial active texture unit selector is TEXTURE0, and the initial matrix mode is MODELVIEW.

### 2.12.3 Normal Transformation

Finally, we consider how the model-view matrix and transformation state affect normals. Before use in lighting, normals are transformed to eye coordinates by a matrix derived from the model-view matrix. Rescaling and normalization operations are performed on the transformed normals to make them unit length prior to use in lighting. Rescaling and normalization are controlled by

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

and

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

with `target` equal to RESCALE_NORMAL or NORMALIZE. This requires two bits of state. The initial state is for normals not to be rescaled or normalized.

If the model-view matrix is $M$, then the normal is transformed to eye coordinates by:

$$
\begin{pmatrix}
    n' \\
    y' \\
    z'
\end{pmatrix}
= \begin{pmatrix}
    nx \\
    ny \\
    nz
\end{pmatrix} \cdot M^{-1}
$$

where, if $\begin{pmatrix}
    x \\
    y \\
    z \\
    w
\end{pmatrix}$ are the associated vertex coordinates, then

$$
q = \begin{cases}
    0, & w = 0, \\
    -\begin{pmatrix}
        nx \\
        ny \\
        nz
    \end{pmatrix}
    \begin{pmatrix}
        x \\
        y \\
        z
    \end{pmatrix}, & w \neq 0
\end{cases}
$$

(2.1)

Implementations may choose instead to transform $\begin{pmatrix}
    nx \\
    ny \\
    nz
\end{pmatrix}$ to eye coordinates using
2.12. COORDINATE TRANSFORMATIONS

\[(n_x', n_y', n_z') = (n_x, n_y, n_z) \cdot M_u^{-1}\]

where \(M_u\) is the upper leftmost 3x3 matrix taken from \(M\).

Rescale multiplies the transformed normals by a scale factor

\[(n_x'', n_y'', n_z'') = f (n_x', n_y', n_z')\]

If rescaling is disabled, then \(f = 1\). If rescaling is enabled, then \(f\) is computed as \((m_{ij}\) denotes the matrix element in row \(i\) and column \(j\) of \(M^{-1}\), numbering the topmost row of the matrix as row 1 and the leftmost column as column 1)

\[f = \frac{1}{\sqrt{m_{31}^2 + m_{32}^2 + m_{33}^2}}\]

Note that if the normals sent to GL were unit length and the model-view matrix uniformly scales space, then rescale makes the transformed normals unit length.

Alternatively, an implementation may choose \(f\) as

\[f = \frac{1}{\sqrt{n_x'^2 + n_y'^2 + n_z'^2}}\]

recomputing \(f\) for each normal. This makes all non-zero length normals unit length regardless of their input length and the nature of the model-view matrix.

After rescaling, the final transformed normal used in lighting, \(n_f\), is computed as

\[n_f = m \cdot (n_x'', n_y'', n_z'')\]

If normalization is disabled, then \(m = 1\). Otherwise

\[m = \frac{1}{\sqrt{n_x''^2 + n_y''^2 + n_z''^2}}\]

Because we specify neither the floating-point format nor the means for matrix inversion, we cannot specify behavior in the case of a poorly-conditioned (nearly singular) model-view matrix \(M\). In case of an exactly singular matrix, the transformed normal is undefined. If the GL implementation determines that the model-view matrix is uninvertible, then the entries in the inverted matrix are arbitrary. In any case, neither normal transformation nor use of the transformed normal may lead to GL interruption or termination.

Version 3.0 (with changes marked - August 11, 2008)
2.12.4 Generating Texture Coordinates

Texture coordinates associated with a vertex may either be taken from the current texture coordinates or generated according to a function dependent on vertex coordinates. The command

```c
void TexGen(id){ enum coord, enum pname, T param );
void TexGen(id)v( enum coord, enum pname, T params );
```

controls texture coordinate generation. coord must be one of the constants \( S, T, R, \) or \( Q \), indicating that the pertinent coordinate is the \( s, t, r, \) or \( q \) coordinate, respectively. In the first form of the command, param is a symbolic constant specifying a single-valued texture generation parameter; in the second form, params is a pointer to an array of values that specify texture generation parameters. pname must be one of the three symbolic constants TEXGENMODE, OBJECTPLANE, or EYEPLANE. If pname is TEXGENMODE, then either params points to or param is an integer that is one of the symbolic constants OBJECTLINEAR, EYELINEAR, SPHERE_MAP, REFLECTION_MAP, or NORMAL_MAP.

If TEXGENMODE indicates OBJECTLINEAR, then the generation function for the coordinate indicated by coord is

\[
g = p_1 x_o + p_2 y_o + p_3 z_o + p_4 w_o.
\]

\( x_o, y_o, z_o, \) and \( w_o \) are the object coordinates of the vertex. \( p_1, \ldots, p_4 \) are specified by calling TexGen with pname set to OBJECTPLANE in which case params points to an array containing \( p_1, \ldots, p_4 \). There is a distinct group of plane equation coefficients for each texture coordinate; coord indicates the coordinate to which the specified coefficients pertain.

If TEXGENMODE indicates EYELINEAR, then the function is

\[
g = p'_1 x_e + p'_2 y_e + p'_3 z_e + p'_4 w_e
\]

where

\[
\begin{pmatrix}
  p'_1 & p'_2 & p'_3 & p'_4 \\
\end{pmatrix}
= \begin{pmatrix}
  p_1 & p_2 & p_3 & p_4 \\
\end{pmatrix} M^{-1}
\]

\( x_e, y_e, z_e, \) and \( w_e \) are the eye coordinates of the vertex. \( p_1, \ldots, p_4 \) are set by calling TexGen with pname set to EYEPLANE in correspondence with setting the coefficients in the OBJECTPLANE case. \( M \) is the model-view matrix in effect when \( p_1, \ldots, p_4 \) are specified. Computed texture coordinates may be inaccurate or undefined if \( M \) is poorly conditioned or singular.

When used with a suitably constructed texture image, calling TexGen with TEXGENMODE indicating SPHERE_MAP can simulate the reflected image of...
2.12. COORDINATE TRANSFORMATIONS

a spherical environment on a polygon. SPHERE_MAP texture coordinates are generated as follows. Denote the unit vector pointing from the origin to the vertex (in eye coordinates) by \( \mathbf{u} \). Denote the current normal, after transformation to eye coordinates, by \( \mathbf{n}_f \). Let \( \mathbf{r} = (r_x \ y \ z)^T \), the reflection vector, be given by

\[
\mathbf{r} = \mathbf{u} - 2\mathbf{n}_f^T (\mathbf{n}_f \mathbf{u}),
\]

and let \( m = 2\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2} \). Then the value assigned to an \( s \) coordinate (the first TexGen argument value is \( s \)) is \( s = r_x/m + \frac{1}{2} \); the value assigned to a \( t \) coordinate is \( t = r_y/m + \frac{1}{2} \). Calling TexGen with a coord of either \( R \) or \( Q \) when \( \text{pname} \) indicates SPHERE_MAP generates the error INVALID_ENUM.

If \( \text{TEXTURE_GEN_MODE} \) indicates REFLECTION_MAP, compute the reflection vector \( \mathbf{r} \) as described for the SPHERE_MAP mode. Then the value assigned to an \( s \) coordinate is \( s = r_x \); the value assigned to a \( t \) coordinate is \( t = r_y \); and the value assigned to an \( r \) coordinate is \( r = r_z \). Calling TexGen with a coord of \( Q \) when \( \text{pname} \) indicates REFLECTION_MAP generates the error INVALID_ENUM.

If \( \text{TEXTURE_GEN_MODE} \) indicates NORMAL_MAP, compute the normal vector \( \mathbf{n}_f \) as described in section 2.12.3. Then the value assigned to an \( s \) coordinate is \( s = n_{fx} \); the value assigned to a \( t \) coordinate is \( t = n_{fy} \); and the value assigned to an \( r \) coordinate is \( r = n_{fz} \) (the values \( n_{fx}, n_{fy}, \) and \( n_{fz} \) are the components of \( \mathbf{n}_f \)). Calling TexGen with a coord of \( Q \) when \( \text{pname} \) indicates NORMAL_MAP generates the error INVALID_ENUM.

A texture coordinate generation function is enabled or disabled using Enable and Disable with an argument of \( \text{TEXTURE_GEN_S}, \text{TEXTURE_GEN_T}, \text{TEXTURE_GEN_R}, \) or \( \text{TEXTURE_GEN_Q} \) (each indicates the corresponding texture coordinate). When enabled, the specified texture coordinate is computed according to the current EYE_LINEAR, OBJECT_LINEAR or SPHERE_MAP specification, depending on the current setting of TEXTURE_GEN_MODE for that coordinate. When disabled, subsequent vertices will take the indicated texture coordinate from the current texture coordinates.

The state required for texture coordinate generation for each texture unit comprises a five-valued integer for each coordinate indicating coordinate generation mode, and a bit for each coordinate to indicate whether texture coordinate generation is enabled or disabled. In addition, four coefficients are required for the four coordinates for each of EYE_LINEAR and OBJECT_LINEAR. The initial state has the texture generation function disabled for all texture coordinates. The initial values of \( p_i \) for \( s \) are all 0 except \( p_1 \) which is one; for \( t \) all the \( p_i \) are zero except \( p_2 \), which is 1. The values of \( p_i \) for \( r \) and \( q \) are all 0. These values of \( p_i \) apply for both the EYE_LINEAR and OBJECT_LINEAR versions. Initially all texture generation modes are EYE_LINEAR.

Version 3.0 (with changes marked - August 11, 2008)
2.13 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 2.15) 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 4.1.7) 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 6.1.12 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 by calling

```c
void BeginQuery( enum target, uint id );
```

with an unused name id. target indicates the type of query to be performed; valid values of target are defined in subsequent sections. When a query object is created, the name id is marked as used and associated with a new query object.

**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 active query object name for any query type, or if id is the active query object for conditional rendering (see section 2.14), 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
2.14. CONDITIONAL RENDERING

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.

2.14 Conditional Rendering

Conditional rendering can be used to discard rendering commands based on the result of an occlusion query. Conditional rendering is started and stopped using the commands

```c
void BeginConditionalRender(uint id, enum mode);
void EndConditionalRender(void);
```

$id$ specifies the name of an occlusion query object whose results are used to determine if the rendering commands are discarded. If the result (SAMPLES_PASSED) of the query is zero, all rendering commands between BeginConditionalRender and the corresponding EndConditionalRender are discarded. In this case, Begin, End, all vertex array commands performing an implicit Begin and End, DrawPixels (see section 3.7.4), Bitmap (see section 3.8), Clear (see section 4.2.3), Accum (see section 4.2.4), CopyPixels (see section 4.3.3), and EvalMesh1 and

Version 3.0 (with changes marked - August 11, 2008
EvalMesh2 (see section 5.1) have no effect. The effect of commands setting current vertex state, such as Color or 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.

### 2.15 Transform Feedback

In transform feedback mode, attributes of the vertices of transformed primitives processed by a vertex shader are written out to one or more buffer objects. The vertices are fed back after vertex color clamping, but before clipping. The transformed vertices may be optionally discarded after being stored into one or more buffer objects, or they can be passed on down to the clipping stage for further processing. The set of attributes captured is determined when a program is linked.

Transform feedback is started and finished by calling

```c
void BeginTransformFeedback( enum primitiveMode );
```

and

```
```
Transform Feedback

2.15. TRANSFORM FEEDBACK

<table>
<thead>
<tr>
<th>Transform Feedback primitiveMode</th>
<th>Allowed render primitive (Begin) 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, QUADS, QUAD_STRIP, POLYGON</td>
</tr>
</tbody>
</table>

Table 2.9: Legal combinations of the transform feedback primitive mode, as passed to `BeginTransformFeedback`, and the current primitive mode.

```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 2.9.

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 an active vertex shader. The error INVALID_OPERATION is generated by `BeginTransformFeedback` if no vertex shader is active.

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 `Begin` or any operation that implicitly calls `Begin` (such as `DrawElements`) if mode is not one of the allowed modes in table 2.9.

Buffer objects are made to be targets of transform feedback 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 );
```

with `target` set to TRANSFORM_FEEDBACK_BUFFER. There is an array of buffer object binding points that are used while transform feedback is active, plus a

Version 3.0 (with changes marked - August 11, 2008
single general binding point that can be used by other buffer object manipulation functions (e.g., `BindBuffer`, `MapBuffer`). Both commands bind the buffer object named by `buffer` to the general binding point, and additionally bind the buffer object 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 value of `MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`.

For `BindBufferRange`, `offset` specifies a starting offset into the buffer object `buffer`, and `size` specifies the amount of data that can be written to the buffer object while transform feedback mode is active. Both `offset` and `size` are in basic machine units. The error `INVALID_VALUE` is generated if the value of `size` is less than or equal to zero, if `offset + size` is greater than the value of `BUFFER_SIZE`, or if either `offset` or `size` are not word-aligned. `BindBufferBase` is equivalent to calling `BindBufferRange` with `offset` zero and `size` equal to the size of `buffer`, rounded down so that it is word-aligned.

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 shader is undefined.

When quads and polygons are provided to transform feedback with a primitive mode of `TRIANGLES`, they will be tessellated and recorded as triangles (the order of tessellation within a primitive 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 2.20.3). 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 \( \text{MAX}_\text{TRANSFORM\_FEEDBACK\_SEPARATE\_ATTRIBS} \).

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 \( \text{offset} + \text{size} - 1 \), as set by \text{BindBufferRange}, then no vertices of that primitive are recorded in any buffer object, and the counter corresponding to the asynchronous query target \text{TRANSFORM\_FEEDBACK\_PRIMITIVES\_WRITTEN} (see section 2.16) 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 \text{BeginTransformFeedback} is called. The error \text{INVALID\_OPERATION} is generated by \text{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 \text{INVALID\_OPERATION} is also generated by \text{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 \text{INVALID\_OPERATION} is generated by \text{UseProgram}, by \text{LinkProgram} if \text{program} is the currently active program object, and by \text{BindBufferRange} or \text{BindBufferBase} if \text{target} is \text{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 \text{DrawPixels} and \text{ReadPixels} to a pixel buffer object binding point and client access to a buffer mapped with \text{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, \text{MapBuffer} may need to block pending the completion of a previous transform feedback operation.
2.16 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.

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.

2.17 Clipping

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

\[-w_c \leq x_c \leq w_c\]
\[-w_c \leq y_c \leq w_c\]
\[-w_c \leq z_c \leq w_c\]

This view volume may be further restricted by as many as *n* client-defined clip planes to generate the clip volume. (*n* is an implementation dependent maximum that must be at least 6.) Each client-defined plane specifies a half-space. The clip volume is the intersection of all such half-spaces with the view volume (if there no client-defined clip planes are enabled, the clip volume is the view volume).

A client-defined clip plane is specified with

```c
void ClipPlane(enum p, double eqn[4]);
```

The value of the first argument, *p*, is a symbolic constant, CLIP_PLANE*i*, where *i* is an integer between 0 and *n* − 1, indicating one of *n* client-defined clip planes. *eqn*
is an array of four double-precision floating-point values. These are the coefficients of a plane equation in object coordinates: \(p_1, p_2, p_3,\) and \(p_4\) (in that order). The inverse of the current model-view matrix is applied to these coefficients, at the time they are specified, yielding

\[
(p'_1 \ p'_2 \ p'_3 \ p'_4) = (p_1 \ p_2 \ p_3 \ p_4) \ M^{-1}
\]

(where \(M\) is the current model-view matrix; the resulting plane equation is undefined if \(M\) is singular and may be inaccurate if \(M\) is poorly-conditioned) to obtain the plane equation coefficients in eye coordinates. All points with eye coordinates \((x_e \ y_e \ z_e \ w_e)^T\) that satisfy

\[
(p'_1 \ p'_2 \ p'_3 \ p'_4) \begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix} \geq 0
\]

lie in the half-space defined by the plane; points that do not satisfy this condition do not lie in the half-space.

When a vertex shader is active, the vector \((x_e \ y_e \ z_e \ w_e)^T\) is no longer computed. Instead, the value of the \text{gl\_ClipVertex} built-in variable is used in its place. If \text{gl\_ClipVertex} is not written by the vertex shader, its value is undefined, which implies that the results of clipping to any client-defined clip planes are also undefined. The user must ensure that the clip vertex and client-defined clip planes are defined in the same coordinate space.

A vertex shader may, instead of writing to \text{gl\_ClipVertex}, write a single clip distance for each supported clip plane to elements of the \text{gl\_ClipDistance[]} array. The half-space corresponding to clip plane \(n\) 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 clip planes are enabled with the generic \text{Enable} command and disabled with the \text{Disable} command. The value of the argument to either command is \text{CLIP\_PLANE}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 \text{CLIP\_PLANE}i = \text{CLIP\_PLANE}0 + i.
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 in color, secondary color, texture coordinate, and fog coordinate clipping (section 2.19.8).

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. Edge flags are associated with these vertices so that edges introduced by clipping are flagged as boundary (edge flag \( \text{TRUE} \)), and so that original edges of the polygon that become cut off at these vertices retain their original flags.

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. This point must lie in the intersection of the boundary edge and the convex hull of the vertices of the original polygon. We impose this requirement because the polygon may not be exactly planar.

Primitives rendered with clip planes must satisfy a complementarity criterion. Suppose a single clip plane with coefficients \( (p'_1, p'_2, p'_3, p'_4) \) (or a number of similarly specified clip planes) is enabled and a series of primitives are drawn. Next, suppose that the original clip plane is respecified with coefficients \( (-p'_1, -p'_2, -p'_3, -p'_4) \) (and correspondingly for any other clip planes) and the primitives are drawn again (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 6 sets of plane equations (each consisting of four double-precision floating-point coefficients) and at least 6 corresponding bits indicating which of these client-defined plane equations are enabled. In the
2.18. CURRENT RASTER POSITION

initial state, all client-defined plane equation coefficients are zero and all planes are disabled.

### 2.18 Current Raster Position

The **current raster position** is used by commands that directly affect pixels in the framebuffer. These commands, which bypass vertex transformation and primitive assembly, are described in the next chapter. The current raster position, however, shares some of the characteristics of a vertex.

The current raster position is set using one of the commands

```c
void RasterPos234{sifd}( T coords );
void RasterPos234{sifd}v( T coords );
```

**RasterPos4** takes four values indicating \(x, y, z,\) and \(w\). **RasterPos3** (or **RasterPos2**) is analogous, but sets only \(x, y,\) and \(z\) with \(w\) implicitly set to 1 (or only \(x\) and \(y\) with \(z\) implicitly set to 0 and \(w\) implicitly set to 1).

**Gets** of **CURRENT_RASTER_TEXTURE_COORDS** are affected by the setting of the state **ACTIVE_TEXTURE**.

The coordinates are treated as if they were specified in a **Vertex** command. If a vertex shader is active, this vertex shader is executed using the \(x, y, z,\) and \(w\) coordinates as the object coordinates of the vertex. Otherwise, the \(x, y, z,\) and \(w\) coordinates are transformed by the current model-view and projection matrices. These coordinates, along with current values, are used to generate primary and secondary colors and texture coordinates just as is done for a vertex. The colors and texture coordinates so produced replace the colors and texture coordinates stored in the current raster position’s associated data. If a vertex shader is active then the current raster distance is set to the value of the shader built in varying `gl_FogFragCoord`. Otherwise, if the value of the fog source (see section 3.11) is `FOGCOORD`, then the current raster distance is set to the value of the current fog coordinate. Otherwise, the current raster distance is set to the distance from the origin of the eye coordinate system to the vertex as transformed by only the current model-view matrix. This distance may be approximated as discussed in section 3.11.

Since vertex shaders may be executed when the raster position is set, any attributes not written by the shader will result in undefined state in the current raster position. Vertex shaders should output all varying variables that would be used when rasterizing pixel primitives using the current raster position.

The transformed coordinates are passed to clipping as if they represented a point. If the “point” is not culled, then the projection to window coordinates is
computed (section 2.12) and saved as the current raster position, and the valid bit is set. If the “point” is culled, the current raster position and its associated data become indeterminate and the valid bit is cleared. Figure 2.7 summarizes the behavior of the current raster position.

Alternately, the current raster position may be set by one of the WindowPos commands:

```cpp
void WindowPos{23}{ifds}( T coords );
void WindowPos{23}{ifds}v( const T coords );
```

WindowPos3 takes three values indicating x, y and z, while WindowPos2 takes two values indicating x and y with z implicitly set to 0. The current raster position, \((x_w, y_w, z_w, w_c)\), is defined by:

\[
\begin{align*}
x_w &= x \\
y_w &= y \\
z_w &= \begin{cases}
  n, & z \leq 0 \\
  f, & z \geq 1 \\
  n + z(f - n), & \text{otherwise}
\end{cases}
\end{align*}
\]

where \(n\) and \(f\) are the values passed to DepthRange (see section 2.12.1).

Lighting, texture coordinate generation and transformation, and clipping are not performed by the WindowPos functions. Instead, in RGBA mode, the current raster color and secondary color are obtained from the current color and secondary color, respectively. If vertex color clamping is enabled, the current raster color and secondary color are clamped to \([0, 1]\). In color index mode, the current raster color index is set to the current color index. The current raster texture coordinates are set to the current texture coordinates, and the valid bit is set.

If the value of the fog source is FOG_COORD_SRC, then the current raster distance is set to the value of the current fog coordinate. Otherwise, the raster distance is set to 0.

The current raster position requires six single-precision floating-point values for its \(x_w, y_w, z_w\), window coordinates, its \(w_c\) clip coordinate, its raster distance (used as the fog coordinate in raster processing), a single valid bit, four floating-point values to store the current RGBA color, four floating-point values to store the current RGBA secondary color, one floating-point value to store the current color index, and 4 floating-point values for texture coordinates for each texture unit. In the initial state, the coordinates and texture coordinates are all \((0, 0, 0, 1)\), the eye

Version 3.0 (with changes marked - August 11, 2008
Figure 2.7. The current raster position and how it is set. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation.
coordinate distance is 0, the fog coordinate is 0, the valid bit is set, the associated RGBA color is \((1, 1, 1, 1)\), the associated RGBA secondary color is \((0, 0, 0, 1)\), and the associated color index color is 1. In RGBA mode, the associated color index always has its initial value; in color index mode, the RGBA color and secondary color always maintain their initial values.

### 2.19 Colors and Coloring

Figures 2.8 and 2.9 diagram the processing of RGBA colors and color indices before rasterization. Incoming colors arrive in one of several formats. Table 2.10 summarizes the conversions that take place on R, G, B, and A components depending on which version of the `Color` command was invoked to specify the components. As a result of limited precision, some converted values will not be represented exactly. In color index mode, a single-valued color index is not mapped.

Next, lighting, if enabled, produces either a color index or primary and secondary colors. If lighting is disabled, the current color index or current color (primary color) and current secondary color are used in further processing. After lighting, RGBA colors may be clamped to the range \([0, 1]\) as described in section 2.19.6. A color index is converted to fixed-point and then its integer por-
2.19. COLORS AND COLORING

Figure 2.9. Processing of color indices. \( n \) is the number of bits in a color index.

<table>
<thead>
<tr>
<th>GL Type of ( c )</th>
<th>Conversion to floating-point</th>
</tr>
</thead>
<tbody>
<tr>
<td>ubyte</td>
<td>( \frac{2^8 - 1}{2^8 - 1} )</td>
</tr>
<tr>
<td>byte</td>
<td>( \frac{2^8 - 1}{2^8 - 1} )</td>
</tr>
<tr>
<td>ushort</td>
<td>( \frac{2^{16} - 1}{2^{16} - 1} )</td>
</tr>
<tr>
<td>short</td>
<td>( \frac{2^{16} - 1}{2^{16} - 1} )</td>
</tr>
<tr>
<td>ushort</td>
<td>( \frac{2^{16} - 1}{2^{16} - 1} )</td>
</tr>
<tr>
<td>uint</td>
<td>( \frac{2^{16} - 1}{2^{16} - 1} )</td>
</tr>
<tr>
<td>int</td>
<td>( \frac{2^{16} - 1}{2^{16} - 1} )</td>
</tr>
<tr>
<td>half</td>
<td>( \frac{c}{2^{16} - 1} )</td>
</tr>
<tr>
<td>float</td>
<td>( c )</td>
</tr>
<tr>
<td>double</td>
<td>( c )</td>
</tr>
</tbody>
</table>

Table 2.10: Component conversions. Color, normal, and depth component values \((c)\) of different types are converted to an internal floating-point representation using the equations in this table. All arithmetic is done in the internal floating point format. These conversions apply to components specified as parameters to GL commands and to components in pixel data. The equations remain the same even if the implemented ranges of the GL data types are greater than the minimum required ranges. (Refer to table 2.2)
tion is masked (see section 2.19.6). After clamping or masking, a primitive may be flatshaded, indicating that all vertices of the primitive are to have the same colors. Finally, if a primitive is clipped, then colors (and texture coordinates) must be computed at the vertices introduced or modified by clipping.

2.19.1 Lighting

GL lighting computes colors for each vertex sent to the GL. This is accomplished by applying an equation defined by a client-specified lighting model to a collection of parameters that can include the vertex coordinates, the coordinates of one or more light sources, the current normal, and parameters defining the characteristics of the light sources and a current material. The following discussion assumes that the GL is in RGBA mode. (Color index lighting is described in section 2.19.5.)

Lighting is turned on or off using the generic Enable or Disable commands with the symbolic value LIGHTING. If lighting is off, the current color and current secondary color are assigned to the vertex primary and secondary color, respectively. If lighting is on, colors computed from the current lighting parameters are assigned to the vertex primary and secondary colors.

Lighting Operation

A lighting parameter is of one of five types: color, position, direction, real, or boolean. A color parameter consists of four floating-point values, one for each of R, G, B, and A, in that order. There are no restrictions on the allowable values for these parameters. A position parameter consists of four floating-point coordinates (x, y, z, and w) that specify a position in object coordinates (w may be zero, indicating a point at infinity in the direction given by x, y, and z). A direction parameter consists of three floating-point coordinates (x, y, and z) that specify a direction in object coordinates. A real parameter is one floating-point value. The various values and their types are summarized in table 2.11. The result of a lighting computation is undefined if a value for a parameter is specified that is outside the range given for that parameter in the table.

There are n light sources, indexed by $i = 0, \ldots, n - 1$. (n is an implementation dependent maximum that must be at least 8.) Note that the default values for $d_{cli}$ and $s_{cli}$ differ for $i = 0$ and $i > 0$.

Before specifying the way that lighting computes colors, we introduce operators and notation that simplify the expressions involved. If $c_1$ and $c_2$ are colors without alpha where $c_1 = (r_1, g_1, b_1)$ and $c_2 = (r_2, g_2, b_2)$, then define $c_1 \ast c_2 = (r_1r_2, g_1g_2, b_1b_2)$. Addition of colors is accomplished by addition of
### Material Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{cm}$</td>
<td>color</td>
<td>$(0.2, 0.2, 0.2, 1.0)$</td>
<td>ambient color of material</td>
</tr>
<tr>
<td>$d_{cm}$</td>
<td>color</td>
<td>$(0.8, 0.8, 0.8, 1.0)$</td>
<td>diffuse color of material</td>
</tr>
<tr>
<td>$s_{cm}$</td>
<td>color</td>
<td>$(0.0, 0.0, 0.0, 1.0)$</td>
<td>specular color of material</td>
</tr>
<tr>
<td>$e_{cm}$</td>
<td>color</td>
<td>$(0.0, 0.0, 0.0, 1.0)$</td>
<td>emissive color of material</td>
</tr>
<tr>
<td>$s_{rm}$</td>
<td>real</td>
<td>$0.0$</td>
<td>specular exponent (range: $[0.0, 128.0)$)</td>
</tr>
<tr>
<td>$a_m$</td>
<td>real</td>
<td>$0.0$</td>
<td>ambient color index</td>
</tr>
<tr>
<td>$d_m$</td>
<td>real</td>
<td>$1.0$</td>
<td>diffuse color index</td>
</tr>
<tr>
<td>$s_m$</td>
<td>real</td>
<td>$1.0$</td>
<td>specular color index</td>
</tr>
</tbody>
</table>

### Light Source Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{li}$</td>
<td>color</td>
<td>$(0.0, 0.0, 0.0, 1.0)$</td>
<td>ambient intensity of light $i$</td>
</tr>
<tr>
<td>$d_{li}(i = 0)$</td>
<td>color</td>
<td>$(1.0, 1.0, 1.0, 1.0)$</td>
<td>diffuse intensity of light $i$ (range: $[0.0, 128.0]$)</td>
</tr>
<tr>
<td>$d_{li}(i &gt; 0)$</td>
<td>color</td>
<td>$(0.0, 0.0, 0.0, 1.0)$</td>
<td>diffuse intensity of light $i$</td>
</tr>
<tr>
<td>$s_{li}(i = 0)$</td>
<td>color</td>
<td>$(1.0, 1.0, 1.0, 1.0)$</td>
<td>specular intensity of light $i$ (range: $[0.0, 128.0]$)</td>
</tr>
<tr>
<td>$s_{li}(i &gt; 0)$</td>
<td>color</td>
<td>$(0.0, 0.0, 0.0, 1.0)$</td>
<td>specular intensity of light $i$</td>
</tr>
<tr>
<td>$P_{pi}$</td>
<td>position</td>
<td>$(0.0, 0.0, 1.0, 0.0)$</td>
<td>position of light $i$</td>
</tr>
<tr>
<td>$s_{li}$</td>
<td>direction</td>
<td>$(0.0, 0.0, 0.0, 1.0)$</td>
<td>direction of spotlight for light $i$</td>
</tr>
<tr>
<td>$s_{el}$</td>
<td>real</td>
<td>$0.0$</td>
<td>spotlight exponent for light $i$ (range: $[0.0, 128.0]$)</td>
</tr>
<tr>
<td>$c_{el}$</td>
<td>real</td>
<td>$180.0$</td>
<td>spotlight cutoff angle for light $i$ (range: $[0.0, 90.0], 180.0$)</td>
</tr>
<tr>
<td>$k_{0i}$</td>
<td>real</td>
<td>$1.0$</td>
<td>constant attenuation factor for light $i$ (range: $[0.0, \infty)$)</td>
</tr>
<tr>
<td>$k_{1i}$</td>
<td>real</td>
<td>$0.0$</td>
<td>linear attenuation factor for light $i$ (range: $[0.0, \infty)$)</td>
</tr>
<tr>
<td>$k_{2i}$</td>
<td>real</td>
<td>$0.0$</td>
<td>quadratic attenuation factor for light $i$ (range: $[0.0, \infty)$)</td>
</tr>
</tbody>
</table>

### Lighting Model Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{es}$</td>
<td>color</td>
<td>$(0.2, 0.2, 0.2, 1.0)$</td>
<td>ambient color of scene</td>
</tr>
<tr>
<td>$v_{bs}$</td>
<td>boolean</td>
<td>FALSE</td>
<td>viewer assumed to be at $(0, 0, 0)$ in eye coordinates (TRUE) or $(0, 0, \infty)$ (FALSE)</td>
</tr>
<tr>
<td>$c_{es}$</td>
<td>enum</td>
<td>SINGLE_COLOR</td>
<td>controls computation of colors</td>
</tr>
<tr>
<td>$t_{bs}$</td>
<td>boolean</td>
<td>FALSE</td>
<td>use two-sided lighting mode</td>
</tr>
</tbody>
</table>

Table 2.11: Summary of lighting parameters. The range of individual color components is $(-\infty, +\infty)$. 
the components. Multiplication of colors by a scalar means multiplying each component by that scalar. If \( \mathbf{d}_1 \) and \( \mathbf{d}_2 \) are directions, then define
\[
\mathbf{d}_1 \odot \mathbf{d}_2 = \max\{\mathbf{d}_1 \cdot \mathbf{d}_2, 0\}.
\]
(Directions are taken to have three coordinates.) If \( \mathbf{P}_1 \) and \( \mathbf{P}_2 \) are (homogeneous, with four coordinates) points then let \( \overrightarrow{\mathbf{P}_1 \mathbf{P}_2} \) be the unit vector that points from \( \mathbf{P}_1 \) to \( \mathbf{P}_2 \). Note that if \( \mathbf{P}_2 \) has a zero \( w \) coordinate and \( \mathbf{P}_1 \) has non-zero \( w \) coordinate, then \( \overrightarrow{\mathbf{P}_1 \mathbf{P}_2} \) is the unit vector that is the negative of that corresponding to the direction specified by \( \mathbf{P}_1 \). If both \( \mathbf{P}_1 \) and \( \mathbf{P}_2 \) have zero \( w \) coordinates, then \( \overrightarrow{\mathbf{P}_1 \mathbf{P}_2} \) is the unit vector obtained by normalizing the direction corresponding to \( \mathbf{P}_2 - \mathbf{P}_1 \).

If \( \mathbf{d} \) is an arbitrary direction, then let \( \hat{\mathbf{d}} \) be the unit vector in \( \mathbf{d} \)'s direction. Let \( ||\overrightarrow{\mathbf{P}_1 \mathbf{P}_2}|| \) be the distance between \( \mathbf{P}_1 \) and \( \mathbf{P}_2 \). Finally, let \( \mathbf{V} \) be the point corresponding to the vertex being lit, and \( \mathbf{n} \) be the corresponding normal. Let \( \mathbf{P}_e \) be the eyepoint \(((0, 0, 0, 1)\) in eye coordinates).

Lighting produces two colors at a vertex: a primary color \( \mathbf{c}_{pri} \) and a secondary color \( \mathbf{c}_{sec} \). The values of \( \mathbf{c}_{pri} \) and \( \mathbf{c}_{sec} \) depend on the light model color control, \( \mathbf{c}_{es} \).

If \( \mathbf{c}_{es} = \text{SINGLE}_\text{COLOR} \), then the equations to compute \( \mathbf{c}_{pri} \) and \( \mathbf{c}_{sec} \) are
\[
\mathbf{c}_{pri} = \mathbf{e}_{cm} + \mathbf{a}_{cm} \ast \mathbf{a}_{es}
+ \sum_{i=0}^{n-1} (att_i)(\text{spot}_i) [\mathbf{a}_{cm} \ast \mathbf{a}_{cli}]
+ (\mathbf{n} \odot \overrightarrow{\mathbf{V} \mathbf{P}_{cli}}) \mathbf{d}_{cm} \ast \mathbf{d}_{cli}
+ (f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^s \mathbf{s}_{cm} \ast \mathbf{s}_{cli} \]
\[
\mathbf{c}_{sec} = (0, 0, 0, 1)
\]

If \( \mathbf{c}_{es} = \text{SEPARATE}_\text{SPECULAR}_\text{COLOR} \), then
\[
\mathbf{c}_{pri} = \mathbf{e}_{cm} + \mathbf{a}_{cm} \ast \mathbf{a}_{es}
+ \sum_{i=0}^{n-1} (att_i)(\text{spot}_i) [\mathbf{a}_{cm} \ast \mathbf{a}_{cli}]
+ (\mathbf{n} \odot \overrightarrow{\mathbf{V} \mathbf{P}_{cli}}) \mathbf{d}_{cm} \ast \mathbf{d}_{cli} \]
\[
\mathbf{c}_{sec} = \sum_{i=0}^{n-1} (att_i)(\text{spot}_i)(f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^s \mathbf{s}_{cm} \ast \mathbf{s}_{cli} \]
where

\[
 f_i = \begin{cases} 
 1, & \mathbf{n} \odot \mathbf{VP}_{pli} \neq 0, \\
 0, & \text{otherwise},
\end{cases}
\]  
\[ (2.2) \]

\[
 h_i = \begin{cases} 
 \mathbf{VP}_{pli} + \mathbf{VP}_c, & v_{bs} = \text{TRUE}, \\
 \mathbf{VP}_{pli} + (0 \; 0 \; 1)^T, & v_{bs} = \text{FALSE},
\end{cases}
\]  
\[ (2.3) \]

\[
 att_i = \begin{cases} 
 \frac{1}{k_0 + k_1 \| \mathbf{VP}_{pli} \| + k_2 \| \mathbf{VP}_{pli} \|^2}, & \text{if } P_{pli} \text{'s } w \neq 0, \\
 1.0, & \text{otherwise}.
\end{cases}
\]  
\[ (2.4) \]

\[
 spot_i = \begin{cases} 
 \frac{(\mathbf{P}_{pli} \mathbf{V} \odot \mathbf{s}_{dli})^{s_{pri}}}{c_{rti} \neq 180.0}, & \mathbf{P}_{pli} \mathbf{V} \odot \mathbf{s}_{dli} \geq \cos(c_{rti}), \\
 0.0, & \mathbf{P}_{pli} \mathbf{V} \odot \mathbf{s}_{dli} < \cos(c_{rti}), \text{ or } c_{rti} \neq 180.0, \\
 1.0, & c_{rti} = 180.0.
\end{cases}
\]  
\[ (2.5) \]

All computations are carried out in eye coordinates.

The value of A produced by lighting is the alpha value associated with \( d_{cm} \). A is always associated with the primary color \( c_{pri} \); the alpha component of \( c_{sec} \) is always 1.

Results of lighting are undefined if the \( w \_e \) coordinate (\( w \) in eye coordinates) of \( V \) is zero.

Lighting may operate in two-sided mode (\( t_{bs} = \text{TRUE} \)), in which a front color is computed with one set of material parameters (the front material) and a back color is computed with a second set of material parameters (the back material). This second computation replaces \( n \) with \( -n \). If \( t_{bs} = \text{FALSE} \), then the back color and front color are both assigned the color computed using the front material with \( n \).

Additionally, vertex shaders can operate in two-sided color mode. When a vertex shader is active, front and back colors can be computed by the vertex shader and written to the \( \text{gl\_FrontColor} \), \( \text{gl\_BackColor} \), \( \text{gl\_FrontSecondaryColor} \) and \( \text{gl\_BackSecondaryColor} \) outputs. If \( \text{VERTEX\_PROGRAM\_TWO\_SIDE} \) is enabled, the GL chooses between front and back colors, as described below. Otherwise, the front color output is always selected. Two-sided color mode is
2.19. COLORS AND COLORING

enabled and disabled by calling **Enable** or **Disable** with the symbolic value **VERTEX_PROGRAM.TWO_SIDE**.

The selection between back and front colors depends on the primitive of which the vertex being lit is a part. If the primitive is a point or a line segment, the front color is always selected. If it is a polygon, then the selection is based on the sign of the (clipped or unclipped) polygon’s signed area computed in window coordinates. One way to compute this area is

\[
a = \frac{1}{2} \sum_{i=0}^{n-1} x^i_w y^i_{w-1} - x^i_{w-1} y^i_w
\]

where \(x^i_w\) and \(y^i_w\) are the \(x\) and \(y\) window coordinates of the \(i\)th vertex of the \(n\)-vertex polygon (vertices are numbered starting at zero for purposes of this computation) and \(i \oplus 1\) is \((i + 1) \mod n\). The interpretation of the sign of this value is controlled with

**void** **FrontFace**( enum dir );

Setting **dir** to **CCW** (corresponding to counter-clockwise orientation of the projected polygon in window coordinates) indicates that if \(a \leq 0\), then the color of each vertex of the polygon becomes the back color computed for that vertex while if \(a > 0\), then the front color is selected. If **dir** is **CW**, then \(a\) is replaced by \(-a\) in the above inequalities. This requires one bit of state; initially, it indicates **CCW**.

### 2.19.2 Lighting Parameter Specification

Lighting parameters are divided into three categories: material parameters, light source parameters, and lighting model parameters (see table 2.11). Sets of lighting parameters are specified with

**void** **Material**{if}( enum face, enum pname, T param );
**void** **Material**{if}v( enum face, enum pname, T params );
**void** **Light**{if}( enum light, enum pname, T param );
**void** **Light**{if}v( enum light, enum pname, T params );
**void** **LightModel**{if}( enum pname, T param );
**void** **LightModel**{if}v( enum pname, T params );

**pname** is a symbolic constant indicating which parameter is to be set (see table 2.12). In the vector versions of the commands, **params** is a pointer to a group of values to which to set the indicated parameter. The number of values pointed to depends on the parameter being set. In the non-vector versions, **param** is a value to

Version 3.0 (with changes marked - August 11, 2008)
which to set a single-valued parameter. (If \textit{param} corresponds to a multi-valued parameter, the error \texttt{INVALID\_ENUM} results.) For the \texttt{Material} command, \textit{face} must be one of \texttt{FRONT}, \texttt{BACK}, or \texttt{FRONT\_AND\_BACK}, indicating that the property \textit{name} of the front or back material, or both, respectively, should be set. In the case of \texttt{Light}, \textit{light} is a symbolic constant of the form \texttt{LIGHT\_i}, indicating that light \textit{i} is to have the specified parameter set. The constants obey \texttt{LIGHT\_i} = \texttt{LIGHT\_0} + \textit{i}.

Table 2.12 gives, for each of the three parameter groups, the correspondence between the pre-defined constant names and their names in the lighting equations, along with the number of values that must be specified with each. Color parameters specified with \texttt{Material} and \texttt{Light} are converted to floating-point values (if specified as integers) as indicated in table 2.10 for signed integers. The error \texttt{INVALID\_VALUE} occurs if a specified lighting parameter lies outside the allowable range given in table 2.11. (The symbol \(\infty\) indicates the maximum representable magnitude for the indicated type.)

Material properties can be changed inside a \texttt{Begin/End} pair by calling \texttt{Material}. However, when a vertex shader is active such property changes are not guaranteed to update material parameters, defined in table 2.12, until the following \texttt{End} command.

The current model-view matrix is applied to the position parameter indicated with \texttt{Light} for a particular light source when that position is specified. These transformed values are the values used in the lighting equation.

The spotlight direction is transformed when it is specified using only the upper leftmost 3x3 portion of the model-view matrix. That is, if \(M_u\) is the upper left 3x3 matrix taken from the current model-view matrix \(M\), then the spotlight direction

\[
\begin{pmatrix}
  d_x \\
  d_y \\
  d_z 
\end{pmatrix}
\]

is transformed to

\[
\begin{pmatrix}
  d'_x \\
  d'_y \\
  d'_z 
\end{pmatrix} = M_u \begin{pmatrix}
  d_x \\
  d_y \\
  d_z 
\end{pmatrix} .
\]

An individual light is enabled or disabled by calling \texttt{Enable} or \texttt{Disable} with the symbolic value \texttt{LIGHT\_i} (\textit{i} is in the range 0 to \(n - 1\), where \(n\) is the implementation-dependent number of lights). If light \textit{i} is disabled, the \textit{i}th term in the lighting equation is effectively removed from the summation.
Table 2.12: Correspondence of lighting parameter symbols to names. AMBIENT_AND_DIFFUSE is used to set $a_{cm}$ and $d_{cm}$ to the same value.
2.19. COLORS AND COLORING

Figure 2.10. ColorMaterial operation. Material properties are continuously updated from the current color while ColorMaterial is enabled and has the appropriate mode. Only the front material properties are included in this figure. The back material properties are treated identically, except that face must be BACK or FRONT_AND_BACK.
2.19.3 ColorMaterial

It is possible to attach one or more material properties to the current color, so that they continuously track its component values. This behavior is enabled and disabled by calling `Enable` or `Disable` with the symbolic value `COLOR_MATERIAL`.

The command that controls which of these modes is selected is

```c
void ColorMaterial(enum face, enum mode);
```

*face* is one of `FRONT`, `BACK`, or `FRONT_AND_BACK`, indicating whether the front material, back material, or both are affected by the current color. *mode* is one of `EMISSION`, `AMBIENT`, `DIFFUSE`, `SPECULAR`, or `AMBIENT_AND_DIFFUSE` and specifies which material property or properties track the current color. If *mode* is `EMISSION`, `AMBIENT`, `DIFFUSE`, or `SPECULAR`, then the value of $e_{cm}$, $a_{cm}$, $d_{cm}$, or $s_{cm}$, respectively, will track the current color. If *mode* is `AMBIENT_AND_DIFFUSE`, both $a_{cm}$ and $d_{cm}$ track the current color. The replacements made to material properties are permanent; the replaced values remain until changed by either sending a new color or by setting a new material value when `ColorMaterial` is not currently enabled to override that particular value. When `COLOR_MATERIAL` is enabled, the indicated parameter or parameters always track the current color. For instance, calling

```c
ColorMaterial(FRONT, AMBIENT)
```

while `COLOR_MATERIAL` is enabled sets the front material $a_{cm}$ to the value of the current color.

Material properties can be changed inside a `Begin/End` pair indirectly by enabling `ColorMaterial` mode and making `Color` calls. However, when a vertex shader is active such property changes are not guaranteed to update material parameters, defined in table 2.12, until the following `End` command.

2.19.4 Lighting State

The state required for lighting consists of all of the lighting parameters (front and back material parameters, lighting model parameters, and at least 8 sets of light parameters), a bit indicating whether a back color distinct from the front color should be computed, at least 8 bits to indicate which lights are enabled, a five-valued variable indicating the current `ColorMaterial` mode, a bit indicating whether or not `COLOR_MATERIAL` is enabled, and a single bit to indicate whether lighting is enabled or disabled. In the initial state, all lighting parameters have their default values. Back color evaluation does not take place. `ColorMaterial` is `FRONT_AND_BACK` and `AMBIENT_AND_DIFFUSE`, and both lighting and `COLOR_MATERIAL` are disabled.

Version 3.0 (with changes marked - August 11, 2008)
2.19.5 Color Index Lighting

A simplified lighting computation applies in color index mode that uses many of the parameters controlling RGBA lighting, but none of the RGBA material parameters. First, the RGBA diffuse and specular intensities of light \(i\) (\(d_{cli}\) and \(s_{cli}\), respectively) determine color index diffuse and specular light intensities, \(d_{li}\) and \(s_{li}\) from

\[
d_{li} = (0.3)R(d_{cli}) + (0.59)G(d_{cli}) + (0.11)B(d_{cli})
\]

and

\[
s_{li} = (0.3)R(s_{cli}) + (0.59)G(s_{cli}) + (0.11)B(s_{cli}).
\]

\(R(x)\) indicates the R component of the color \(x\) and similarly for \(G(x)\) and \(B(x)\).

Next, let

\[
s = \sum_{i=0}^{n} (att_i)(spot_i)(s_{li})(f_i)(n \odot \hat{h}_i)^{s_{rm}}
\]

where \(att_i\) and \(spot_i\) are given by equations 2.4 and 2.5, respectively, and \(f_i\) and \(\hat{h}_i\) are given by equations 2.2 and 2.3, respectively. Let \(s' = \min\{s, 1\}\). Finally, let

\[
d = \sum_{i=0}^{n} (att_i)(spot_i)(d_{li})(n \odot \overrightarrow{VP}_{pli}).
\]

Then color index lighting produces a value \(c\), given by

\[
c = a_m + d(1 - s')(d_m - a_m) + s'(s_m - a_m).
\]

The final color index is

\[
c' = \min\{c, s_m\}.
\]

The values \(a_m\), \(d_m\) and \(s_m\) are material properties described in tables 2.11 and 2.12. Any ambient light intensities are incorporated into \(a_m\). As with RGBA lighting, disabled lights cause the corresponding terms from the summations to be omitted.

The interpretation of \(t_{ba}\) and the calculation of front and back colors is carried out as has already been described for RGBA lighting.

The values \(a_m\), \(d_m\), and \(s_m\) are set with Material using a pname of COLOR_INDEXES. Their initial values are 0, 1, and 1, respectively. The additional state consists of three floating-point values. These values have no effect on RGBA lighting.

Version 3.0 (with changes marked - August 11, 2008)
2.19. COLORS AND COLORING

2.19.6 Clamping or Masking

When the GL is in RGBA mode and vertex color clamping is enabled, all components of both primary and secondary colors are clamped to the range \([0, 1]\) after lighting. If color clamping is disabled, the primary and secondary colors are unmodified. Vertex color clamping is controlled by calling

```c
void ClampColor( enum target, enum clamp );
```

with `target` set to `CLAMP_VERTEX_COLOR`. If `clamp` is `TRUE`, vertex color clamping is enabled; if `clamp` is `FALSE`, vertex color clamping is disabled. If `clamp` is `FIXED_ONLY`, vertex color clamping is enabled if all enabled color buffers have fixed-point components.

For a color index, the index is first converted to fixed-point with an unspecified number of bits to the right of the binary point; the nearest fixed-point value is selected. Then, the bits to the right of the binary point are left alone while the integer portion is masked (bitwise ANDed) with \(2^n - 1\), where \(n\) is the number of bits in a color in the color index buffer (buffers are discussed in chapter 4).

The state required for color clamping is a three-valued integer, initially set to `TRUE`.

2.19.7 Flatshading

A primitive may be flatshaded, meaning that all vertices of the primitive are assigned the same color index or the same primary and secondary colors. These colors are the colors of the vertex that spawned the primitive. For a point, these are the colors associated with the point. For a line segment, they are the colors of the second (final) vertex of the segment. For a polygon, they come from a selected vertex depending on how the polygon was generated. Table 2.13 summarizes the possibilities.

Flatshading is controlled by

```c
void ShadeModel( enum mode );
```

`mode` value must be either of the symbolic constants `SMOOTH` or `FLAT`. If `mode` is `SMOOTH` (the initial state), vertex colors are treated individually. If `mode` is `FLAT`, flatshading is turned on. `ShadeModel` thus requires one bit of state.

If a vertex shader is active, the flat shading control applies to the built-in varying variables `gl_FrontColor`, `gl_BackColor`, `gl_FrontSecondaryColor` and `gl_BackSecondaryColor`. Non-color varying variables can be specified as being flat-shaded via the `flat` qualifier, as described in section 4.3.6 of the OpenGL Shading Language Specification.
2.19 COLORS AND COLORING

Table 2.13: Polygon flatshading color selection. The colors used for flatshading the i\(^{th}\) polygon generated by the indicated \textbf{Begin/End} type are derived from the current color (if lighting is disabled) in effect when the indicated vertex is specified. If lighting is enabled, the colors are produced by lighting the indicated vertex. Vertices are numbered 1 through \(n\), where \(n\) is the number of vertices between the \textbf{Begin/End} pair.

<table>
<thead>
<tr>
<th>Primitive type of polygon (i)</th>
<th>Vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>single polygon ((i \equiv 1))</td>
<td>1</td>
</tr>
<tr>
<td>triangle strip</td>
<td>(i + 2)</td>
</tr>
<tr>
<td>triangle fan</td>
<td>(i + 2)</td>
</tr>
<tr>
<td>independent triangle</td>
<td>(3i)</td>
</tr>
<tr>
<td>quad strip</td>
<td>(2i + 2)</td>
</tr>
<tr>
<td>independent quad</td>
<td>(4i)</td>
</tr>
</tbody>
</table>

2.19.8 Color and Associated Data Clipping

After lighting, clamping or masking and possible flatshading, colors are clipped. Those colors associated with a vertex that lies within the clip volume are unaffected by clipping. If a primitive is clipped, however, the colors assigned to vertices produced by clipping are clipped colors.

Let the colors 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 2.17) for a clipped point \(P\) is used to obtain the color associated with \(P\) as

\[
c = tc_1 + (1 - t)c_2.
\]

(For a color index color, multiplying a color by a scalar means multiplying the index by the scalar. For an RGBA color, it means multiplying each of R, G, B, and A by the scalar. Both primary and secondary colors are treated in the same fashion.) 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 plane of the clip volume’s boundary at a time. Color 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.

Texture and fog coordinates, vertex shader varying variables (section 2.20.3), and point sizes computed on a per vertex basis must also be clipped when a primitive is clipped. The method is exactly analogous to that used for color clipping.

For vertex shader varying variables specified to be interpolated without per-
2.20. VERTEX SHADERS

spective correction (using the 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.

2.19.9 Final Color Processing

In RGBA mode with vertex color clamping disabled, the floating-point RGBA components are not modified.

In RGBA mode with vertex color clamping enabled, each color component (already clamped to \([0, 1]\)) may be converted (by rounding to nearest) to a fixed-point value with \( m \) bits. We assume that the fixed-point representation used 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). \( m \) must be at least as large as the number of bits in the corresponding component of the framebuffer. \( m \) must be at least 2 for \( A \) if the framebuffer does not contain an \( A \) component, or if there is only 1 bit of \( A \) in the framebuffer. GL implementations are not required to convert clamped color components to fixed-point.

Because a number of the form \( k/(2^m - 1) \) may not be represented exactly as a limited-precision floating-point quantity, we place a further requirement on the fixed-point conversion of RGBA components. Suppose that lighting is disabled, the color associated with a vertex has not been clipped, and one of \texttt{Colorub}, \texttt{Colorus}, or \texttt{Colorui} was used to specify that color. When these conditions are satisfied, an RGBA component must convert to a value that matches the component as specified in the \texttt{Color} command: if \( m \) is less than the number of bits \( b \) with which the component was specified, then the converted value must equal the most significant \( m \) bits of the specified value; otherwise, the most significant \( b \) bits of the converted value must equal the specified value.

A color index is converted (by rounding to nearest) to a fixed-point value with at least as many bits as there are in the color index portion of the framebuffer.

2.20 Vertex Shaders

The sequence of operations described in sections 2.12 through 2.19 is a fixed-function method for processing vertex data. Applications can more generally describe the operations that occur on vertex values and their associated data by using a vertex shader.

A vertex shader is an array of strings containing source code for the operations that are meant to occur on each vertex that is processed. The language used for vertex shaders is described in the OpenGL Shading Language Specification.
To use a vertex shader, shader source code is first loaded into a shader object and then compiled. One or more vertex shader objects are then attached to a program object. A program object is then 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, fragment shaders can be created, compiled, and linked into program objects. Fragment shaders affect the processing of fragments during rasterization, and are described in section 3.12. A single program object can contain both vertex and fragment shaders.

When the program object currently in use includes a vertex shader, its vertex shader is considered active and is used to process vertices. If the program object has no vertex shader, or no program object is currently in use, the fixed-function method for processing vertices is used instead.

### 2.20.1 Shader Objects

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

The name space for shader objects is the unsigned integers, with zero reserved for the GL. This name space is shared with program objects. The following sections define commands that operate on shader and program objects by name. Commands that accept shader or program object names will generate the error INVALID_VALUE if the provided name is not the name of either a shader or program object and INVALID_OPERATION if the provided name identifies an object that is not the expected type.

To create a shader object, use the command

```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, COMPILER_STATUS, that is modified as a result of compilation. This status can be queried with GetShaderiv (see section 6.1.15). This status will be set to TRUE if shader was compiled without errors and is ready for use, and FALSE otherwise. Compilation can fail for a variety of reasons as listed in the OpenGL Shading Language Specification. If CompileShader failed, any information about a previous compile is lost. Thus a failed compile does not restore the old state of shader.

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

Each shader object has an information log, which is a text string that is overwritten as a result of compilation. This information log can be queried with GetShaderInfoLog to obtain more information about the compilation attempt (see section 6.1.15).

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 6.1.15). DeleteShader will silently ignore the value zero.

### 2.20.2 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

Version 3.0 (with changes marked - August 11, 2008)
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, 0 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 `pro-
gram`.

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 6.1.15). 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 section 2.20.3). 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`.

Version 3.0 (with changes marked - August 11, 2008)
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 6.1.15).

If a valid executable is created, it can be made part of the current rendering state with the command

```c
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, it is as if the GL had no programmable stages and the fixed-function paths will be used instead. 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

```c
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.

### 2.20.3 Shader Variables

A vertex shader can reference a number of variables as it executes. **Vertex attributes** are the per-vertex values specified in section 2.7. **Uniforms** are per-program vari-
ables that are constant during program execution. Samplers are a special form of uniform used for texturing (section 3.9). Varying variables hold the results of vertex shader execution that are used later in the pipeline. The following sections describe each of these variable types.

Vertex Attributes

Vertex shaders can access built-in vertex attribute variables corresponding to the per-vertex state set by commands such as **Vertex, Normal, Color**. Vertex shaders can also 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 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).

An attribute variable (either conventional or generic) 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 sum of the active generic and active conventional attributes exceeds **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);
```

Version 3.0 (with changes marked - August 11, 2008)
This command provides information about the attribute selected by index. An index of 0 selects the first active attribute, and an index of ACTIVE_ATTRIBUTES − 1 selects the last active attribute. The value of ACTIVE_ATTRIBUTES can be queried with GetProgramiv (see section 6.1.15). If index is greater than or equal to ACTIVE_ATTRIBUTES, the error INVALID_VALUE is generated. Note that index simply identifies a member in a list of active attributes, and has no relation to the generic attribute that the corresponding variable is bound to.

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

The name of the selected attribute is returned as a null-terminated string in name. The actual number of characters written into name, excluding the null terminator, is returned in length. If length is NULL, no length is returned. The maximum number of characters that may be written into name, including the null terminator, is specified by bufSize. The returned attribute name can be the name of a generic attribute or a conventional attribute (which begin with the prefix "gl_", see the OpenGL Shading Language specification for a complete list). The length of the longest attribute name in program is given by ACTIVE_ATTRIBUTE_MAX_LENGTH, which can be queried with GetProgramiv (see section 6.1.15).

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

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

returns the generic attribute index that the attribute variable named name was bound
2.20. VERTEX SHADERS

to when the program object named \textit{program} was last linked. \textit{name} must be a null-terminated string. If \textit{name} is active and is an attribute matrix, \texttt{GetAttribLocation} returns the index of the first column of that matrix. If \textit{program} has not been successfully linked, the error \texttt{INVALID\_OPERATION} is generated. If \textit{name} is not an active attribute, if \textit{name} is a conventional 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

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

specifies that the attribute variable named \textit{name} in program \textit{program} should be bound to generic vertex attribute \textit{index} when the program is next linked. If \textit{name} was bound previously, its assigned binding is replaced with \textit{index}. \textit{name} must be a null terminated string. The error \texttt{INVALID\_VALUE} is generated if \textit{index} is equal or greater than \texttt{MAX\_VERTEX\_ATTRIBS}. \texttt{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.

Built-in attribute variables are automatically bound to conventional attributes, and cannot have an assigned binding. The error \texttt{INVALID\_OPERATION} is generated if \textit{name} starts with the reserved "gl_" prefix.

When a program is linked, any active attributes without a binding specified through \texttt{BindAttribLocation} will be automatically bound to vertex attributes by the GL. Such bindings can be queried using the command \texttt{GetAttribLocation}. \texttt{LinkProgram} will fail if the assigned binding of an active attribute variable would cause the GL to reference a non-existant generic attribute (one greater than or equal to \texttt{MAX\_VERTEX\_ATTRIBS}). \texttt{LinkProgram} will fail if the attribute bindings assigned by \texttt{BindAttribLocation} do not leave not enough space to assign a location for an active matrix attribute, which requires multiple contiguous generic attributes. \texttt{LinkProgram} will also fail if the vertex shaders used in the program object contain assignments (not removed during pre-processing) to an attribute variable bound to generic attribute zero and to the conventional vertex position (\texttt{gl\_Vertex}).

\texttt{BindAttribLocation} may be issued before any vertex shader objects are attached to a program object. Hence it is allowed to bind any name (except a name starting with "gl_") 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 \textit{i} are part of current state, just like the conventional attributes. 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. It is not possible to alias generic attributes with conventional ones.

**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.

The amount of storage available for uniform variables accessed by a vertex shader is specified by the implementation dependent constant `MAX_VERTEX_UNIFORM_COMPONENTS`. This value represents the number of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a vertex shader. A uniform matrix will 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 vertex shader uniform variables.

When a program is successfully linked, all active uniforms belonging to the program object 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. The values of active uniforms 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.

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

Version 3.0 (with changes marked - August 11, 2008)
2.20. VERTEX SHADERS

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

This command will return the location of uniform variable `name`. The value `-1` will be returned if `name` does not correspond to an active uniform variable name in `program` or if `name` starts with the reserved prefix "gl_". 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]".

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

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

This command provides information about the uniform selected by `index`. An `index` of 0 selects the first active uniform, and an `index` of `ACTIVE_UNIFORMS` – 1 selects the last active uniform. The value of `ACTIVE_UNIFORMS` can be queried with `GetProgramiv` (see section 6.1.15). If `index` is greater than or equal to `ACTIVE_UNIFORMS`, the error `INVALID_VALUE` is generated. Note that `index` simply identifies a member in a list of active uniforms, and has no relation to the location assigned to the corresponding uniform variable.

The parameter `program` is a 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.

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

For the selected uniform, the uniform name is returned into `name`. The string `name` will be null terminated. The actual number of characters written into `name`,

Version 3.0 (with changes marked - August 11, 2008)
excluding the null terminator, is returned in \textit{length}. If \textit{length} is \texttt{NULL}, no length is returned. The maximum number of characters that may be written into \textit{name}, 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 \textit{program} is given by \texttt{ACTIVE_UNIFORM_MAX_LENGTH}, which can be queried with \texttt{GetProgramiv} (see section 6.1.15).

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}. Each of these strings constitutes one active uniform, and each string is assigned an index.

For the selected uniform, the type of the uniform is returned into \textit{type}. The size of the uniform is returned into \textit{size}. The value in \textit{size} is in units of the type returned in \textit{type}. The type returned can be any of \texttt{FLOAT}, \texttt{FLOAT_VEC2}, \texttt{FLOAT_VEC3}, \texttt{FLOAT_VEC4}, \texttt{INT}, \texttt{INT_VEC2}, \texttt{INT_VEC3}, \texttt{INT_VEC4}, \texttt{BOOL}, \texttt{BOOL_VEC2}, \texttt{BOOL_VEC3}, \texttt{BOOL_VEC4}, \texttt{FLOAT_MAT2}, \texttt{FLOAT_MAT3}, \texttt{FLOAT_MAT4}, \texttt{FLOAT_MAT2x3}, \texttt{FLOAT_MAT2x4}, \texttt{FLOAT_MAT3x2}, \texttt{FLOAT_MAT3x4}, \texttt{FLOAT_MAT4x2}, \texttt{FLOAT_MAT4x3}, \texttt{SAMPLER_1D}, \texttt{SAMPLER_2D}, \texttt{SAMPLER_3D}, \texttt{SAMPLER_CUBE}, \texttt{SAMPLER_1D_SHADOW}, \texttt{SAMPLER_2D_SHADOW}, \texttt{SAMPLER_1D_ARRAY}, \texttt{SAMPLER_2D_ARRAY}, \texttt{SAMPLER_1D_ARRAY_SHADOW}, \texttt{SAMPLER_2D_ARRAY_SHADOW}, \texttt{SAMPLER_CUBE_SHADOW}, \texttt{INT_SAMPLER_1D}, \texttt{INT_SAMPLER_2D}, \texttt{INT_SAMPLER_3D}, \texttt{INT_SAMPLER_CUBE}, \texttt{UNSIGNED_INT}, \texttt{UNSIGNED_INT_VEC2}, \texttt{UNSIGNED_INT_VEC3}, \texttt{UNSIGNED_INT_VEC4}, \texttt{UNSIGNED_INT_SAMPLER_1D}, \texttt{UNSIGNED_INT_SAMPLER_2D}, \texttt{UNSIGNED_INT_SAMPLER_3D}, \texttt{UNSIGNED_INT_SAMPLER_CUBE}, \texttt{UNSIGNED_INT_SAMPLER_1D_ARRAY}, or \texttt{UNSIGNED_INT_SAMPLER_2D_ARRAY}.

If one or more elements of an array are active, \texttt{GetActiveUniform} will return the name of the array in \textit{name}, subject to the restrictions listed above. The type of the array is returned in \textit{type}. The \textit{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.

\texttt{GetActiveUniform} will return as much information about active uniforms as possible. If no information is available, \textit{length} will be set to zero and \textit{name} will be an empty string. This situation could arise if \texttt{GetActiveUniform} is issued after a failed link.

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

\texttt{gpu_shader4}

Version 3.0 (with changes marked - August 11, 2008)
2.20. VERTEX SHADERS

void Uniform{1,234}{if}(int location, T value);
void Uniform{1,234}{if}(int location, sizei count, T value);
void Uniform{1,2,3,4}ui(int location, T value);
void Uniform{1,2,3,4}ui(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 uniform variable location identified by location.

The Uniform*f{v} commands will load count sets of one to four floating-point values into a uniform location defined as a float, a floating-point vector, an array of floats, or an array of floating-point vectors.

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

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

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

The UniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}fv commands will load count 2 × 3, 3 × 2, 2 × 4, 4 × 2, 3 × 4, or 4 × 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, UniformMatrix2x4fv is used to load a matrix consisting of two columns and four rows. If transpose is FALSE, the matrix is specified in column major order, otherwise in row major order.

When loading values for a uniform declared as a boolean, a boolean vector, an array of booleans, or an array of boolean vectors, the Uniform*i{v},

Version 3.0 (with changes marked - August 11, 2008)
Uniform*ui{v}, and Uniform*f{v} set of commands can be used to load boolean values. Type conversion is done by the GL. The uniform is set to FALSE if the input value is 0 or 0.0f, and set to TRUE otherwise. The Uniform* command used must match the size of the uniform, as declared in the shader. For example, to load a uniform declared as a bvec2, any of the Uniform2{ui}* commands may be used. An INVALID_OPERATION error will be generated if an attempt is made to use a non-matching Uniform* command. In this example using Uniform1iv would generate an error.

For all other uniform types the 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 vec4, Uniform4f{v} must be used. To load a 3x3 matrix, UniformMatrix3fv must be used. An INVALID_OPERATION error will be generated if an attempt is made to use a non-matching Uniform* command. In this example, using Uniform4i{v} would generate an error.

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

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

If any of the following conditions occur, an 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.

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

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

Active samplers are samplers actually being used in a program object. The \text{LinkProgram} command determines if a sampler is active or not. The \text{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. If this cannot be determined at link time, for example if the program object only contains a vertex shader, then it will be determined at the next rendering command issued, and an \text{INVALID\_OPERATION} error will then be generated.

### Varying Variables

A vertex shader may define one or more \textit{varying} variables (see the OpenGL Shading Language specification). These values are expected to be interpolated across the primitive being rendered. The OpenGL Shading Language specification defines a set of built-in varying variables for vertex shaders that correspond to the values required for the fixed-function processing that occurs after vertex processing.

The number of interpolators available for processing varying variables is given by the \text{MAX\_VARYING\_COMPONENTS}. This value represents the number of individual floating-point values that can be interpolated; varying variables declared as vec-
2.20. VERTEX SHADERS

tors, matrices, and arrays will all consume multiple interpolators. When a program is linked, all components of any varying variable written by a vertex shader, read by a fragment shader, or used for transform feedback will count against this limit. The transformed vertex position (gl_Position) is not a varying variable and does not count against this limit. A program whose shaders access more than the value of MAX_VARYING_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.

Each program object can specify a set of one or more varying variables to be recorded in transform feedback mode 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. The varying variables specified in *varyings* can be either built-in varying variables (beginning with "gl_") or user-defined ones. 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 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 shader;
- any variable name specified in the *varyings* array is not declared as an output in the vertex shader.
- any two entries in the *varyings* array specify the same varying variable.
the total number of components to capture in any varying variable in \textit{varyings} is greater than the constant \texttt{MAX\_TRANSFORM\_FEEDBACK\_SEPARATE\_COMPONENTS} and the buffer mode is \texttt{SEPARATE\_ATTRIBS}; or

- the total number of components to capture is greater than the constant \texttt{MAX\_TRANSFORM\_FEEDBACK\_INTERLEAVED\_COMPONENTS} and the buffer mode is \texttt{INTERLEAVED\_ATTRIBS}.

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

\begin{verbatim}
void GetTransformFeedbackVarying( uint program,
    uint index, sizei bufSize, sizei *length, sizei *size,
    enum *type, char *name );
\end{verbatim}

provides information about the varying variable selected by \textit{index}. An \textit{index} of 0 selects the first varying variable specified in the \textit{varyings} array of \texttt{TransformFeedbackVaryings}, and an \textit{index} of \texttt{TRANSFORM\_FEEDBACK\_VARYINGS}-1 selects the last such varying variable. The value of \texttt{TRANSFORM\_FEEDBACK\_VARYINGS} can be queried with \texttt{GetProgramiv} (see section 6.1.15). If \textit{index} is greater than or equal to \texttt{TRANSFORM\_FEEDBACK\_VARYINGS}, the error \texttt{INVALID\_VALUE} is generated. The parameter \textit{program} is the name of a program object for which the command \texttt{LinkProgram} has been issued in the past. If a new set of varying variables is specified by \texttt{TransformFeedbackVaryings} after a program object has been linked, the information returned by \texttt{GetTransformFeedbackVarying} will not reflect those variables until the program is re-linked.

The name of the selected varying is returned as a null-terminated string in \textit{name}. The actual number of characters written into \textit{name}, excluding the null terminator, is returned in \textit{length}. If \textit{length} is \texttt{NULL}, no length is returned. The maximum number of characters that may be written into \textit{name}, including the null terminator, is specified by \textit{bufSize}. The returned varying name can be the name of a user defined varying variable or the name of a built-in varying (which begin with the prefix \texttt{gl}, see the OpenGL Shading Language specification for a complete list). The length of the longest varying name in program is given by \texttt{TRANSFORM\_FEEDBACK\_VARYING\_MAX\_LENGTH}, which can be queried with \texttt{GetProgramiv} (see section 6.1.15).

For the selected varying variable, its type is returned into \textit{type}. The size of the varying is returned into \textit{size}. The value in \textit{size} is in units of the type returned in \textit{type}. The type returned can be any of \texttt{FLOAT}, \texttt{FLOAT\_VEC2}, \texttt{FLOAT\_VEC3}, \texttt{FLOAT\_VEC4}, \texttt{INT}, \texttt{INT\_VEC2}, \texttt{INT\_VEC3}, \texttt{INT\_VEC4}, \texttt{UNSIGNED\_INT}, \texttt{UNSIGNED\_INT\_VEC2}, \texttt{UNSIGNED\_INT\_VEC3}, \texttt{UNSIGNED\_INT\_VEC4}.
2.20. VERTEX SHADERS

UNSIGNED_INT_VEC3, UNSIGNED_INT_VEC4, FLOAT_MAT2, FLOAT_MAT3, or FLOAT_MAT4. 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.

2.20.4 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 rather than the fixed-function vertex processing described in sections 2.12 through 2.19. In particular,

- The model-view and projection matrices are not applied to vertex coordinates (section 2.12).
- The texture matrices are not applied to texture coordinates (section 2.12.2).
- Normals are not transformed to eye coordinates, and are not rescaled or normalized (section 2.12.3).
- Normalization of AUTO_NORMAL evaluated normals is not performed. (section 5.1).
- Texture coordinates are not generated automatically (section 2.12.4).
- Per vertex lighting is not performed (section 2.19.1).
- Color material computations are not performed (section 2.19.3).
- Color index lighting is not performed (section 2.19.5).
- All of the above applies when setting the current raster position (section 2.18).

The following operations are applied to vertex values that are the result of executing the vertex shader:

- Color clamping or masking (section 2.19.6).
- Perspective division on clip coordinates (section 2.12).
- Viewport mapping, including depth range scaling (section 2.12.1).
2.20. VERTEX SHADERS

- Clipping, including client-defined clip planes (section 2.17).
- Front face determination (section 2.19.1).
- Flat-shading (section 2.19.7).
- Color, texture coordinate, fog, point-size and generic attribute clipping (section 2.19.8).
- Final color processing (section 2.19.9).

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

Shader Only Texturing

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

Additional OpenGL Shading Language texture lookup functions (see section 8.7 of the OpenGL Shading Language Specification) return either signed or unsigned integer values if the internal format of the texture is signed or unsigned, respectively.

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 \texttt{NEAREST} or \texttt{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

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

and the size parameters \(w_s, h_s, d_s, \) and \(b_s\) refer to the width, height, depth, and border size of the image, as in equations 3.15

• the texture being accessed is not complete (or cube complete for cubemaps).

**Texture Size Query**

The OpenGL Shading Language texture size functions provide the ability to query the size of a texture image. The LOD value \(lod\) passed in as an argument to the texture size functions is added to the \(level_{base}\) 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 \([level_{base}, level_{max}]\), the results are undefined. When querying the size of an array texture, both the dimensions and the layer index are returned.

**Texture Access**

Vertex shaders have the ability to do a lookup into a texture map, if supported by the GL implementation. The maximum number of texture image units available to a vertex shader is \(\text{MAX\_VERTEX\_TEXTURE\_IMAGE\_UNITS}\); a maximum number of zero indicates that the GL implementation does not support texture accesses in vertex shaders. The maximum number of texture image units available to the fragment stage of the GL is \(\text{MAX\_TEXTURE\_IMAGE\_UNITS}\). Both the vertex shader and fragment processing combined cannot use more than \(\text{MAX\_COMBINED\_TEXTURE\_IMAGE\_UNITS}\) texture image units. If both the vertex shader and the fragment processing stage access the same texture image unit, then that counts as using two texture image units against the \(\text{MAX\_COMBINED\_TEXTURE\_IMAGE\_UNITS}\) limit.

When a texture lookup is performed in a vertex shader, the filtered texture value \(\tau\) is computed in the manner described in sections 3.9.7 and 3.9.8, and converted it to a texture source color \(C_s\) according to table 3.23 (section 3.9.13). A four-component vector \((R_s, G_s, B_s, A_s)\) is returned to the vertex shader.

Version 3.0 (with changes marked - August 11, 2008)
In a vertex 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 3.9.7. 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 3.16). 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 3.20) 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 3.9.14. The comparison operation is requested in the shader by using any of the shadow sampler types 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 vertex shader uses a sampler where the associated texture object is not complete, as defined in section 3.9.10, 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 variable `gl_VertexID`. `gl_VertexID` holds the integer index \( i \) explicitly passed to `ArrayElement` to specify the vertex, or implicitly

Version 3.0 (with changes marked - August 11, 2008)
2.20. VERTEX SHADERS

passed by the **DrawArrays**, **MultiDrawArrays**, **DrawElements**, **MultiDrawElements**, and **DrawRangeElements** commands. The value of `gl_VertexID` is defined if and only if:

- the vertex comes from a vertex array command that specifies a complete primitive (**DrawArrays**, **MultiDrawArrays**, **DrawElements**, **MultiDrawElements**, or **DrawRangeElements**)  
- all enabled vertex arrays have non-zero buffer object bindings, and  
- the vertex does not come from a display list, even if the display list was compiled using one of the vertex array commands described above with data sourced from buffer objects.

Also see section 7.1 of the OpenGL Shading Language Specification.

**Shader Outputs**

A vertex 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. Refer to section 2.19.7 and the OpenGL Shading Language specification sections 4.3.6, 7.1 and 7.6 for more detail.

The built-in output variables `gl_FrontColor`, `gl_BackColor`, `gl_FrontSecondaryColor`, and `gl_BackSecondaryColor` hold the front and back colors for the primary and secondary colors for the current vertex.

The built-in output variable `gl_TexCoord[]` is an array and holds the set of texture coordinates for the current vertex.

The built-in output variable `gl_FogFragCoord` is used as the `c` value described in section 3.11.

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_ClipVertex` holds the vertex coordinate used in the clipping stage, as described in section 2.17.

The built-in special variable `gl_PointSize`, if written, holds the size of the point to be rasterized, measured in pixels.

**Position Invariance**

If a vertex shader uses the built-in function `ftransform` to generate a vertex position, then this generally guarantees that the transformed position will be the same whether using this vertex shader or the fixed-function pipeline. This allows for correct multi-pass rendering algorithms, where some passes use fixed-function vertex
transformation and other passes use a vertex shader. If a vertex shader does not use \texttt{ftransform} to generate a position, transformed positions are not guaranteed to match, even if the sequence of instructions used to compute the position match the sequence of transformations described in section 2.12.

\textbf{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 \texttt{Begin, RasterPos}, or any command that performs an implicit \texttt{Begin} will generate the error \texttt{INVALID\_OPERATION}.

This error is generated by \texttt{Begin, RasterPos}, or any command that performs an implicit \texttt{Begin} if:

- any two active samplers in the current program object are of different types, but refer to the same texture image unit,
- any active sampler in the current program object refers to a texture image unit where fixed-function fragment processing accesses a texture target that does not match the sampler type, or
- the sum of the number of active samplers in the program and the number of texture image units enabled for fixed-function fragment processing exceeds the combined limit on the total number of texture image units allowed.

Fixed-function fragment processing operations will be performed if the program object in use has no fragment shader.

The \texttt{INVALID\_OPERATION} error reported by these rendering commands may not provide enough information to find out why the currently active program object would not execute. No information at all is available about a program object that would still execute, but is inefficient or suboptimal given the current GL state. As a development aid, use the command

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

to validate the program object \texttt{program} against the current GL state. Each program object has a boolean status, \texttt{VALIDATE\_STATUS}, that is modified as a result of validation. This status can be queried with \texttt{GetProgramiv} (see section 6.1.15). If validation succeeded this status will be set to \texttt{TRUE}, otherwise it will be set to \texttt{FALSE}.
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.

### 2.20.5 Required State

The GL maintains state to indicate which shader and program object names are in use. Initially, no shader or program objects exist, and no names are in use.

The state required per shader object consists of:

- An unsigned integer specifying the shader object name.
- An integer holding the value of `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.
2.20. **VERTEX SHADERS**

- 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 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 of words that hold 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 vertex 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.
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 initially determined by the rasterization operations (sections 3.4 through 3.8) and modified by either the execution of the texturing, color sum, and fog operations defined in sections 3.9, 3.10, and 3.11, or by a fragment shader as defined in section 3.12. 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, polygon, pixel rectangle or bitmap can be routed through a fragment shader.

A grid square along with its parameters of assigned colors, $z$ (depth), fog coordinate, and texture coordinates 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.
Figure 3.1. Rasterization.
Several factors affect rasterization. Primitives may be discarded before rasterization. Lines and polygons may be stippled. 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 2.15). When disabled, primitives are passed through to the rasterization stage to be processed normally. RASTERIZER_DISCARD also affects the DrawPixels, CopyPixels, Bitmap, Clear and Accum commands.

### 3.2 Invariance

Consider a primitive $p'$ obtained by translating a primitive $p$ through an offset $(x, y)$ in window coordinates, where $x$ and $y$ are integers. As long as neither $p'$ nor $p$ is clipped, it must be the case that each fragment $f'$ produced from $p'$ is identical to a corresponding fragment $f$ from $p$ except that the center of $f'$ is offset by $(x, y)$ from the center of $f$.

### 3.3 Antialiasing

Antialiasing of a point, line, or polygon is effected in one of two ways depending on whether the GL is in RGBA or color index mode.

In RGBA mode, 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.

In color index mode, the least significant $b$ bits (to the left of the binary point) of the color index are used for antialiasing; $b = \min\{4, m\}$, where $m$ is the number of bits in the color index portion of the framebuffer. The antialiasing process sets these $b$ bits based on the fragment’s coverage value: the bits are set to zero for no coverage and to all ones for complete coverage.

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

---

Version 3.0 (with changes marked - August 11, 2008)
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 num-
ber 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 an-
tialiasing 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 prim-
itive 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 com-
puted 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 5.6), 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, poly-
gons, bitmaps, and images. The technique is to sample all primitives multiple times

Version 3.0 (with changes marked - August 11, 2008)
3.3. ANTIALIASING

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 (left, right, front, back, and aux) 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 points or 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.

Second, each fragment includes SAMPLES depth values, color values, and sets of texture coordinates, instead of the single depth value, color value, and set of texture coordinates that is maintained in single-sample rendering mode. An implementation may choose to assign the same color value and the same set of texture coordinates to more than one sample. The location for evaluating the color value and the set of texture coordinates can be anywhere within the pixel including the fragment center or any of the sample locations. The color value and the set of texture coordinates need not be evaluated at the same location. Each pixel fragment thus consists of integer x and y grid coordinates, SAMPLES color and depth values, SAMPLES sets of texture coordinates, and a coverage value with a maximum of SAMPLES bits.

Multisample rasterization is enabled or disabled by calling Enable or Disable
3.4 POINTS

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.

It is not possible to query the actual sample locations of a pixel.

3.4 Points

If a vertex shader is not active, then the rasterization of points is controlled with

```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.

The requested point size is multiplied with a distance attenuation factor, clamped to a specified point size range, and further clamped to the implementation-dependent point size range to produce the derived point size:

\[
derived\_size = clamp\left( size \times \sqrt{\frac{1}{a + b \times d + c \times d^2}} \right)
\]

where \(d\) is the eye-coordinate distance from the eye, \((0, 0, 0, 1)\) in eye coordinates, to the vertex, and \(a\), \(b\), and \(c\) are distance attenuation function coefficients.

If multisampling is not enabled, the derived size is passed on to rasterization as the point width.
3.4. POINTS

If a vertex shader is active and vertex program point size mode is enabled, then the derived point size is taken from the (potentially clipped) shader built-in `glPointSize` 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 a vertex shader is active and vertex program point size mode is disabled, then the derived point size is taken from the point size state as specified by the `PointSize` command. In this case no distance attenuation is performed. Vertex program point size mode is enabled and disabled by calling `Enable` or `Disable` with the symbolic value `VERTEX_PROGRAM_Point_SIZE`.

If multisampling is enabled, an implementation may optionally fade the point alpha (see section 3.14) instead of allowing the point width to go below a given threshold. In this case, the width of the rasterized point is

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

(3.1)

and the fade factor is computed as follows:

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

(3.2)

The distance attenuation function coefficients \(a\), \(b\), and \(c\), the bounds of the first point size range clamp, and the point fade threshold, are specified with

```c
void PointParameter\(\text{if}\)(enum pname, T param);
void PointParameter\(\text{if}\)\(v\)(enum pname, const T params);
```

If `pname` is `POINT_SIZE_MIN` or `POINT_SIZE_MAX`, then `param` specifies, or `params` points to the lower or upper bound respectively to which the derived point size is clamped. If the lower bound is greater than the upper bound, the point size after clamping is undefined. If `pname` is `POINT_DISTANCE_ATTENUATION`, then `params` points to the coefficients `a`, `b`, and `c`. If `pname` is `POINT_FADE_THRESHOLD_SIZE`, then `param` specifies, or `params` points to the point fade `threshold`. Values of `POINT_SIZE_MIN`, `POINT_SIZE_MAX`, or `POINT_FADE_THRESHOLD_SIZE` less than zero result in the error `INVALID_VALUE`.

Point antialiasing is enabled or disabled by calling `Enable` or `Disable` with the symbolic constant `POINT_SMOOTH`. The default state is for point antialiasing to be disabled.

Point sprites are enabled or disabled by calling `Enable` or `Disable` with the symbolic constant `POINT_SPRITE`. The default state is for point sprites to be dis-
3.4. POINTS

abled. When point sprites are enabled, the state of the point antialiasing enable is ignored.

The point sprite texture coordinate replacement mode is set with one of the Tex-
Env* commands described in section 3.9.13, where target is POINT_SPRITE and
pname is COORD_REPLACE. The possible values for param are FALSE and TRUE.
The default value for each texture coordinate set is for point sprite texture coordi-
nate replacement to be disabled.

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.

3.4.1 Basic Point Rasterization

In the default state, a point is rasterized by truncating its $x_w$ and $y_w$ coordinates (recall that the subscripts indicate that these are $x$ and $y$ window coordinates) to integers. This $(x, y)$ address, along with data derived from the data associated with the vertex corresponding to the point, is sent as a single fragment to the per-
fragment stage of the GL.

The effect of a point width other than 1.0 depends on the state of point antialias-
ing and point sprites. If antialiasing and point sprites are disabled, the actual width is determined by rounding the supplied width to the nearest integer, then clamp-
ing it to the implementation-dependent maximum non-antialiased point width. This implementation-dependent value must be no less than the implementation-
dependent maximum antialiased point width, rounded to the nearest integer value, and in any event no less than 1. If rounding the specified width results in the value 0, then it is as if the value were 1. If the resulting width is odd, then the point

\[(x, y) = (\lfloor x_w \rfloor + \frac{1}{2}, \lfloor y_w \rfloor + \frac{1}{2})\]

is computed from the vertex’s $x_w$ and $y_w$, and a square grid of the odd width cen-
tered at $(x, y)$ defines the centers of the rasterized fragments (recall that fragment centers lie at half-integer window coordinate values). If the width is even, then the center point is

\[(x, y) = ([x_w + \frac{1}{2}], [y_w + \frac{1}{2}]);\]

the rasterized fragment centers are the half-integer window coordinate values within the square of the even width centered on $(x, y)$. See figure 3.2.
3.4. POINTS

Figure 3.2. Rasterization of non-antialiased wide points. The crosses show fragment centers produced by rasterization for any point that lies within the shaded region. The dotted grid lines lie on half-integer coordinates.
Figure 3.3. Rasterization of antialiased wide points. The black dot indicates the point to be rasterized. The shaded region has the specified width. The X marks indicate those fragment centers produced by rasterization. A fragment’s computed coverage value is based on the portion of the shaded region that covers the corresponding fragment square. Solid lines lie on integer coordinates.
All fragments produced in rasterizing a non-antialiased point are assigned the same associated data, which are those of the vertex corresponding to the point.

If antialiasing is enabled and point sprites are disabled, then point rasterization produces a fragment for each fragment square that intersects the region lying within the circle having diameter equal to the current point width and centered at the point’s \((x_w, y_w)\) (figure 3.3). The coverage value for each fragment is the window coordinate area of the intersection of the circular region with the corresponding fragment square (but see section 3.3). This value is saved and used in the final step of rasterization (section 3.13). The data associated with each fragment are otherwise the data associated with the point being rasterized.

Not all widths need be supported when point antialiasing is on, but the width 1.0 must be provided. If an unsupported width is requested, the nearest supported width is used instead. The range of supported widths and the width of evenly-spaced gradations within that range are implementation dependent. The range and gradations may be obtained using the query mechanism described in chapter 6. If, for instance, the width range is from 0.1 to 2.0 and the gradation width is 0.1, then the widths 0.1, 0.2, . . . , 1.9, 2.0 are supported.

If point sprites are enabled, then 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, for each texture coordinate set where \texttt{COORD REPLACE} is \texttt{TRUE}, these texture coordinates are replaced with point sprite texture coordinates. The \(s\) 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 \(t\) coordinates:

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

\[
t = \begin{cases} 
\frac{1}{2} + \frac{(y_f + \frac{1}{2} - y_w)}{size}, & \text{POINT_SPRITE_COORD ORIGIN = LOWER_LEFT} \\
\frac{1}{2} - \frac{(y_f + \frac{1}{2} - y_w)}{size}, & \text{POINT_SPRITE_COORD ORIGIN = UPPER_LEFT} 
\end{cases} \tag{3.4}
\]

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.

The widths supported for point sprites must be a superset of those supported for antialiased points. 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, three floating-point values specifying the minimum and maximum point size and the point fade threshold size, three floating-point values specifying the distance attenuation coefficients, a bit indicating whether or not antialiasing is enabled, a bit for the point sprite texture coordinate replacement mode for each texture coordinate set, and a bit for the point sprite texture coordinate origin.

### 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, regardless of whether point antialiasing (\texttt{POINT\_SMOOTH}) is enabled or disabled. 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 circle having diameter equal to the current point width if \texttt{POINT\_SPRITE} is disabled, or a square with side equal to the current point width if \texttt{POINT\_SPRITE} is enabled. 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, with the exception of texture coordinates when \texttt{POINT\_SPRITE} is enabled; these texture coordinates are computed as described in section 3.4.

Point size range and number of gradations are equivalent to those supported for antialiased points when \texttt{POINT\_SPRITE} is disabled. The set of point sizes supported is equivalent to those for point sprites without multisample when \texttt{POINT\_SPRITE} is enabled.

### 3.5 Line Segments

A line segment results from a line strip \texttt{Begin/End} object, 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

_version 3.0 (with changes marked - August 11, 2008_
3.5. LINE SEGMENTS

```c
void LineWidth(float width);
```

with an appropriate positive floating-point width, controls the width of rasterized line segments. The default width is 1.0. Values less than or equal to 0.0 generate the error `INVALID_VALUE`. Antialiasing is controlled with `Enable` and `Disable` using the symbolic constant `LINE_SMOOTH`. Finally, line segments may be stippled. Stippling is controlled by a GL command that sets a stipple pattern (see below).

### 3.5.1 Basic Line Segment Rasterization

Line segment rasterization begins by characterizing the segment as either x-major or y-major. x-major line segments have slope in the closed interval $[-1, 1]$; all other line segments are y-major (slope is determined by the segment’s endpoints). We shall specify rasterization only for x-major segments except in cases where the modifications for y-major segments are not self-evident.

Ideally, the GL uses a “diamond-exit” rule to determine those fragments that are produced by rasterizing a line segment. For each fragment $f$ with center at window coordinates $x_f$ and $y_f$, define a diamond-shaped region that is the intersection of four half planes:

$$R_f = \{ (x, y) \mid |x - x_f| + |y - y_f| < 1/2 \}$$

Essentially, a line segment starting at $p_a$ and ending at $p_b$ produces those fragments $f$ for which the segment intersects $R_f$, except if $p_b$ is contained in $R_f$. See figure 3.4.

To avoid difficulties when an endpoint lies on a boundary of $R_f$ we (in principle) perturb the supplied endpoints by a tiny amount. Let $p_a$ and $p_b$ have window coordinates $(x_a, y_a)$ and $(x_b, y_b)$, respectively. Obtain the perturbed endpoints $p'_a$ given by $(x_a, y_a) - (\epsilon, \epsilon^2)$ and $p'_b$ given by $(x_b, y_b) - (\epsilon, \epsilon^2)$. Rasterizing the line segment starting at $p_a$ and ending at $p_b$ produces those fragments $f$ for which the segment starting at $p'_a$ and ending on $p'_b$ intersects $R_f$, except if $p'_b$ is contained in $R_f$. $\epsilon$ is chosen to be so small that rasterizing the line segment produces the same fragments when $\delta$ is substituted for $\epsilon$ for any $0 < \delta \leq \epsilon$.

When $p_a$ and $p_b$ lie on fragment centers, this characterization of fragments reduces to Bresenham’s algorithm with one modification: lines produced in this description are “half-open,” meaning that the final fragment (corresponding to $p_b$) is not drawn. This means that when rasterizing a series of connected line segments, shared endpoints will be produced only once rather than twice (as would occur with Bresenham’s algorithm).

Because the initial and final conditions of the diamond-exit rule may be difficult to implement, other line segment rasterization algorithms are allowed, subject to the following rules:

Version 3.0 (with changes marked - August 11, 2008)
1. The coordinates of a fragment produced by the algorithm may not deviate by more than one unit in either $x$ or $y$ window coordinates from a corresponding fragment produced by the diamond-exit rule.

2. The total number of fragments produced by the algorithm may differ from that produced by the diamond-exit rule by no more than one.

3. For an $x$-major line, no two fragments may be produced that lie in the same window-coordinate column (for a $y$-major line, no two fragments may appear in the same row).

4. If two line segments share a common endpoint, and both segments are either $x$-major (both left-to-right or both right-to-left) or $y$-major (both bottom-to-top or both top-to-bottom), then rasterizing both segments may not produce 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
3.5. LINE SEGMENTS

by \( p_r = (x_d, y_d) \) and let \( p_a = (x_a, y_a) \) and \( p_b = (x_b, y_b) \). Set

\[
t = \frac{(p_r - p_a) \cdot (p_b - p_a)}{\|p_b - p_a\|^2}. \tag{3.5}
\]

(Note that \( t = 0 \) at \( p_a \) and \( t = 1 \) at \( p_b \).) The value of an associated datum \( f \) for the fragment, whether it be primary or secondary R, G, B, or A (in RGBA mode) or a color index (in color index mode), the fog coordinate, an \( s, t, r, \) or \( q \) texture coordinate, or the clip \( w \) coordinate, is found as

\[
f = \frac{(1 - t) f_a / w_a + t f_b / w_b}{(1 - t) / w_a + t / w_b} \tag{3.6}
\]

where \( f_a \) and \( f_b \) are the data associated with the starting and ending endpoints of the segment, respectively; \( w_a \) and \( w_b \) are the clip \( w \) coordinates of the starting and ending endpoints of the segments, respectively. However, depth values for lines must be interpolated by

\[
z = (1 - t) z_a + t z_b \tag{3.7}
\]

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

3.5.2 Other Line Segment Features

We have just described the rasterization of non-antialiased line segments of width one using the default line stipple of \( F F F F_{16} \). We now describe the rasterization of line segments for general values of the line segment rasterization parameters.

Line Stipple

The command

```c
void LineStipple(int factor, ushort pattern);
```

defines a line stipple. \( \text{pattern} \) is an unsigned short integer. The line stipple is taken from the lowest order 16 bits of \( \text{pattern} \). It determines those fragments that are to be drawn when the line is rasterized. \( \text{factor} \) is a count that is used to modify the effective line stipple by causing each bit in line stipple to be used \( \text{factor} \) times. \( \text{factor} \) is clamped to the range \([1, 256]\). Line stippling may be enabled or disabled using Enable or Disable with the constant LINE_STIPPLE. When disabled, it is as if the line stipple has its default value.

Line stippling masks certain fragments that are produced by rasterization so that they are not sent to the per-fragment stage of the GL. The masking is achieved
using three parameters: the 16-bit line stipple $p$, the line repeat count $r$, and an integer stipple counter $s$. Let

$$b = \lfloor s/r \rfloor \mod 16,$$

Then a fragment is produced if the $b$th bit of $p$ is 1, and not produced otherwise. The bits of $p$ are numbered with 0 being the least significant and 15 being the most significant. The initial value of $s$ is zero; $s$ is incremented after production of each fragment of a line segment (fragments are produced in order, beginning at the starting point and working towards the ending point). $s$ is reset to 0 whenever a **Begin** occurs, and before every line segment in a group of independent segments (as specified when **Begin** is invoked with **LINES**).

If the line segment has been clipped, then the value of $s$ at the beginning of the line segment is indeterminate.

**Wide Lines**

The actual width of non-antialiased lines is determined by rounding the supplied width to the nearest integer, then clamping it to the implementation-dependent maximum non-antialiased line width. This implementation-dependent value must be no less than the implementation-dependent maximum antialiased line width, rounded to the nearest integer value, and in any event no less than 1. If rounding the specified width results in the value 0, then it is as if the value were 1.

Non-antialiased line segments of width other than one are rasterized by offsetting them in the minor direction (for an $x$-major line, the minor direction is $y$, and for a $y$-major line, the minor direction is $x$) and replicating fragments in the minor direction (see figure 3.5). Let $w$ be the width rounded to the nearest integer (if $w = 0$, then it is as if $w = 1$). If the line segment has endpoints given by $(x_0, y_0)$ and $(x_1, y_1)$ in window coordinates, the segment with endpoints $(x_0, y_0 - (w - 1)/2)$ and $(x_1, y_1 - (w - 1)/2)$ is rasterized, but instead of a single fragment, a column of fragments of height $w$ (a row of fragments of length $w$ for a $y$-major segment) is produced at each $x$ ($y$ for $y$-major) location. The lowest fragment of this column is the fragment that would be produced by rasterizing the segment of width 1 with the modified coordinates. The whole column is not produced if the stipple bit for the column’s $x$ location is zero; otherwise, the whole column is produced.

**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...
3.5. **LINE SEGMENTS**

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 3.6; 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.

For purposes of antialiasing, a stippled line is considered to be a sequence of contiguous rectangles centered on the line segment. Each rectangle has width equal to the current line width and length equal to 1 pixel (except the last, which may be shorter). These rectangles are numbered from 0 to \( n \), starting with the rectangle incident on the starting endpoint of the segment. Each of these rectangles is either eliminated or produced according to the procedure given under **Line Stipple**, above, where “fragment” is replaced with “rectangle.” Each rectangle so produced

---

**Figure 3.5.** Rasterization of non-antialiased wide lines. x-major line segments are shown. The heavy line segment is the one specified to be rasterized; the light segment is the offset segment used for rasterization. x marks indicate the fragment centers produced by rasterization.

---

Version 3.0 (with changes marked - August 11, 2008)
is rasterized as if it were an antialiased polygon, described below (but culling, non-default settings of PolygonMode, and polygon stippling are not applied).

### 3.5.3 Line Rasterization State

The state required for line rasterization consists of the floating-point line width, a 16-bit line stipple, the line stipple repeat count, a bit indicating whether stippling is enabled or disabled, and a bit indicating whether line antialiasing is on or off. In addition, during rasterization, an integer stipple counter must be maintained to implement line stippling. The initial value of the line width is 1.0. The initial value of the line stipple is $FFFF_{16}$ (a stipple of all ones). The initial value of the line stipple repeat count is one. The initial state of line stippling is disabled. The initial state of line segment antialiasing is disabled.

### 3.5.4 Line Multisample Rasterization

If Multisample is enabled, and the value of SAMPLE_BUFFERS is one, then lines are rasterized using the following algorithm, regardless of whether line antialiasing (LINE_SMOOTH) is enabled or disabled. Line rasterization produces a fragment for each framebuffer pixel with one or more sample points that intersect the rectangular region that is described in the Antialiasing portion of section 3.5.2 (Other Line Segment Features). If line stippling is enabled, the rectangular region is subdivided.
3.6 POLYGONS

into adjacent unit-length rectangles, with some rectangles eliminated according to the procedure given in section 3.5.2, where “fragment” is replaced by “rectangle”.

Coverage bits that correspond to sample points that intersect a retained rectangle are 1, other coverage bits are 0. Each color, depth, and set of texture coordinates 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 same color value and the same set of texture coordinates 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 color value and the set of texture coordinates need not be evaluated at the same location.

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

3.6 Polygons

A polygon results from a polygon Begin/End object, a triangle resulting from a triangle strip, triangle fan, or series of separate triangles, or a quadrilateral arising from a quadrilateral strip, series of separate quadrilaterals, or a Rect command. 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. The analog to line segment stippling for polygons is polygon stippling, described below.

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 by examining the sign of the area computed by equation 2.6 of section 2.19.1 (including the possible reversal of this sign as indicated by the last call to FrontFace). If this sign is positive, the polygon is frontfacing; 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 boundary edge. In such a case we require that if two polygons lie on either side of a common edge (with identical endpoints) on which a fragment center lies, then exactly one of the polygons results in the production of the fragment during rasterization.

As for the data associated with each fragment produced by rasterizing a polygon, we begin by specifying how these values are produced for fragments in a triangle. Define barycentric coordinates for a triangle. Barycentric coordinates are a set of three numbers, a, b, and c, each in the range [0, 1], with \(a + b + c = 1\). These coordinates uniquely specify any point \(p\) within the triangle or on the triangle’s boundary as

\[
p = ap_a + bp_b + cp_c,
\]

where \(p_a\), \(p_b\), and \(p_c\) are the vertices of the triangle. \(a\), \(b\), and \(c\) can be found as

\[
a = \frac{A(pp_b p_c)}{A(p_a p_b p_c)}, \quad b = \frac{A(pp_a p_c)}{A(p_a p_b p_c)}, \quad c = \frac{A(pp_a p_b)}{A(p_a p_b p_c)},
\]

where \(A(lmn)\) denotes the area in window coordinates of the triangle with vertices \(l\), \(m\), and \(n\).

Denote an associated datum at \(p_a\), \(p_b\), or \(p_c\) as \(f_a\), \(f_b\), or \(f_c\), respectively. Then the value \(f\) of a datum at a fragment produced by rasterizing a triangle is given by

\[
f = \frac{af_a/w_a + bf_b/w_b + cf_c/w_c}{a/w_a + b/w_b + c/w_c} \tag{3.8}
\]

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

\[
z = az_a + bz_b + cz_c,
\]

where \(z_a\), \(z_b\), and \(z_c\) are the depth values of \(p_a\), \(p_b\), and \(p_c\), respectively.
For a polygon with more than three edges, 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 3.8 should be iterated independently and a division performed for each fragment).

### 3.6.2 Stippling

Polygon stippling works much the same way as line stippling, masking out certain fragments produced by rasterization so that they are not sent to the next stage of the GL. This is the case regardless of the state of polygon antialiasing. Stippling is controlled with

```c
void PolygonStipple(ubyte *pattern);
```

*pattern* is a pointer to memory into which a \( 32 \times 32 \) pattern is packed. The pattern is unpacked from memory according to the procedure given in section 3.7.4 for DrawPixels; it is as if the *height* and *width* passed to that command were both equal to 32, the *type* were BITMAP, and the *format* were COLOR_INDEX. The unpacked values (before any conversion or arithmetic would have been performed) form a stipple pattern of zeros and ones.

If \( x_w \) and \( y_w \) are the window coordinates of a rasterized polygon fragment, then that fragment is sent to the next stage of the GL if and only if the bit of the pattern \((x_w \mod 32, y_w \mod 32)\) is 1.

Polygon stippling may be enabled or disabled with Enable or Disable using the constant POLYGON_STIPPLE. When disabled, it is as if the stipple pattern were all ones.
3.6.3 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 3.13. 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.

Polygon stippling operates in the same way whether polygon antialiasing is enabled or not. The polygon point sampling rule defined in section 3.6.1, however, is not enforced for antialiased polygons.

3.6.4 Options Controlling Polygon Rasterization

The interpretation of polygons for rasterization is controlled using

```c
void PolygonMode( enum face, enum mode );
```

*face* is one of *FRONT*, *BACK*, or *FRONT_AND_BACK*, indicating that the rasterizing method described by *mode* replaces the rasterizing method for front facing polygons, back facing polygons, or both front and back facing polygons, respectively.

*mode* is one of the symbolic constants *POINT*, *LINE*, or *FILL*. Calling *PolygonMode* with *POINT* causes certain vertices of a polygon to be treated, for rasterization purposes, just as if they were enclosed within a *Begin*(POINT) and *End* pair. The vertices selected for this treatment are those that have been tagged as having a polygon boundary edge beginning on them (see section 2.6.2). *LINE* causes edges that are tagged as boundary to be rasterized as line segments. (The line stipple counter is reset at the beginning of the first rasterized edge of the polygon, but not for subsequent edges.) *FILL* is the default mode of polygon rasterization, corresponding to the description in sections 3.6.1, 3.6.2, and 3.6.3. 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. POLYGONS

3.6.5 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

\[
m = \sqrt{\left( \frac{\partial z_w}{\partial x_w} \right)^2 + \left( \frac{\partial z_w}{\partial y_w} \right)^2}
\]  

(3.9)

where \((x_w, y_w, z_w)\) is a point on the triangle. \( m \) may be approximated as

\[
m = \max \left\{ \left| \frac{\partial z_w}{\partial x_w} \right|, \left| \frac{\partial z_w}{\partial y_w} \right| \right\}.
\]  

(3.10)

If the polygon has more than three vertices, one or more values of \( m \) may be used during rasterization. Each may take any value in the range \([\text{min}, \text{max}]\), where \( \text{min} \) and \( \text{max} \) are the smallest and largest values obtained by evaluating equation 3.9 or equation 3.10 for the triangles formed by all three-vertex combinations.

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

\[ r = 2^{e-n} \].

Version 3.0 (with changes marked - August 11, 2008)
The offset value \( o \) for a polygon is

\[
o = m \times factor + r \times units. \tag{3.11}
\]

\( m \) 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.

### 3.6.6 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, including the special treatment for sample points that lie on a polygon boundary edge. If a polygon is culled, based on its orientation and the \texttt{CullFace} mode, then no fragments are produced during rasterization. Fragments are culled by the polygon stipple just as they are for aliased and antialiased polygons.

Coverage bits that correspond to sample points that satisfy the point sampling criteria are 1, other coverage bits are 0. Each color, depth, and set of texture coordinates is produced by substituting the corresponding sample location into the barycentric equations described in section 3.6.1, using the approximation to equation 3.8 that omits \( w \) components. An implementation may choose to assign the same color value and the same set of texture coordinates to more than one sample by barycentric evaluation using any location with the pixel including the fragment...
center or one of the sample locations. The color value and the set of texture coordinates need not be evaluated at the same location.

The rasterization described above applies only to the FILL state of PolygonMode. For POINT and LINE, the rasterizations described in sections 3.4.3 (Point Multisample Rasterization) and 3.5.4 (Line Multisample Rasterization) apply.

3.6.7 Polygon Rasterization State

The state required for polygon rasterization consists of a polygon stipple pattern, whether stippling is enabled or disabled, the current state of polygon antialiasing (enabled or disabled), the current values of the PolygonMode setting for each of front and back facing polygons, 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 stipple pattern is all ones; initially stippling is disabled. The initial setting of polygon antialiasing is disabled. The initial state for PolygonMode is FILL for both front and back facing polygons. 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 converted to fragments using the DrawPixels command (described in section 3.7.4). Some of the parameters and operations governing the operation of DrawPixels are shared by ReadPixels (used to obtain pixel values from the framebuffer) and CopyPixels (used to copy pixels from one framebuffer location to another); the discussion of ReadPixels and CopyPixels, 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 DrawPixels also pertain to ReadPixels or CopyPixels.

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 three commands: PixelStore, PixelTransfer, and PixelMap.

3.7.1 Pixel Storage Modes and Pixel Buffer Objects

Pixel storage modes affect the operation of DrawPixels and ReadPixels (as well as other commands; see sections 3.6.2, 3.8, and 3.9) when one of these commands is issued. This may differ from the time that the command is executed if the command is placed in a display list (see section 5.4). Pixel storage modes are set with
void PixelStore(if)(enum name, T param);

`name` is a symbolic constant indicating a parameter to be set, and `param` is the value to set it to. Table 3.1 summarizes the pixel storage parameters, their types, their initial values, and their allowable ranges. Setting a parameter to a value outside the given range results in the error **INVALID_VALUE**.

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 `DrawPixels` 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 The Imaging Subset

Some pixel transfer and per-fragment operations are only made available in GL implementations which incorporate the optional imaging subset. The imaging subset includes both new commands, and new enumerants allowed as parameters to existing commands. If the subset is supported, all of these calls and enumerants must be implemented as described later in the GL specification. If the subset is not supported, calling any unsupported command generates the error INVALID_OPERATION, and using any of the new enumerants generates the error INVALID_ENUM.

The individual operations available only in the imaging subset are described in section 3.7.3. Imaging subset operations include:

1. Color tables, including all commands and enumerants described in subsections Color Table Specification, Alternate Color Table Specification Commands, Color Table State and Proxy State, Color Table Lookup, Post Convolution Color Table Lookup, and Post Color Matrix Color Table Lookup, as well as the query commands described in section 6.1.7.

2. Convolution, including all commands and enumerants described in subsections Convolution Filter Specification, Alternate Convolution Filter Specification Commands, and Convolution, as well as the query commands described in section 6.1.8.

3. Color matrix, including all commands and enumerants described in subsections Color Matrix Specification and Color Matrix Transformation, as well as the simple query commands described in section 6.1.6.

4. Histogram and minmax, including all commands and enumerants described in subsections Histogram Table Specification, Histogram State and Proxy State, Histogram, Minmax Table Specification, and Minmax, as well as the query commands described in section 6.1.9 and section 6.1.10.

The imaging subset is supported only if the EXTENSIONS string includes the substring "GL_ARB_imaging" Querying EXTENSIONS is described in section 6.1.11.

If the imaging subset is not supported, the related pixel transfer operations are not performed; pixels are passed unchanged to the next operation.

3.7.3 Pixel Transfer Modes

Pixel transfer modes affect the operation of DrawPixels (section 3.7.4), ReadPixels (section 4.3.2), and CopyPixels (section 4.3.3) at the time when one of these
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>MAP_COLOR</td>
<td>boolean</td>
<td>FALSE</td>
<td>TRUE/FALSE</td>
</tr>
<tr>
<td>MAP_STENCIL</td>
<td>boolean</td>
<td>FALSE</td>
<td>TRUE/FALSE</td>
</tr>
<tr>
<td>INDEX_SHIFT</td>
<td>integer</td>
<td>0</td>
<td>$(-\infty, \infty)$</td>
</tr>
<tr>
<td>INDEX_OFFSET</td>
<td>integer</td>
<td>0</td>
<td>$(-\infty, \infty)$</td>
</tr>
<tr>
<td>$x$_SCALE</td>
<td>float</td>
<td>1.0</td>
<td>$(-\infty, \infty)$</td>
</tr>
<tr>
<td>DEPTH_SCALE</td>
<td>float</td>
<td>1.0</td>
<td>$(-\infty, \infty)$</td>
</tr>
<tr>
<td>$x$_BIAS</td>
<td>float</td>
<td>0.0</td>
<td>$(-\infty, \infty)$</td>
</tr>
<tr>
<td>DEPTH_BIAS</td>
<td>float</td>
<td>0.0</td>
<td>$(-\infty, \infty)$</td>
</tr>
<tr>
<td>POST_CONVOLUTION_.$x$.SCALE</td>
<td>float</td>
<td>1.0</td>
<td>$(-\infty, \infty)$</td>
</tr>
<tr>
<td>POST_CONVOLUTION_.$x$.BIAS</td>
<td>float</td>
<td>0.0</td>
<td>$(-\infty, \infty)$</td>
</tr>
<tr>
<td>POST_COLOR_MATRIX_.$x$.SCALE</td>
<td>float</td>
<td>1.0</td>
<td>$(-\infty, \infty)$</td>
</tr>
<tr>
<td>POST_COLOR_MATRIX_.$x$.BIAS</td>
<td>float</td>
<td>0.0</td>
<td>$(-\infty, \infty)$</td>
</tr>
</tbody>
</table>

Table 3.2: PixelTransfer parameters. $x$ is RED, GREEN, BLUE, or ALPHA.

Some pixel transfer modes are set with

```c
void PixelTransfer{if}( enum param, T value );
```

`param` is a symbolic constant indicating a parameter to be set, and `value` is the value to set it to. Table 3.2 summarizes the pixel transfer parameters that are set with `PixelTransfer`, their types, their initial values, and their allowable ranges. Setting a parameter to a value outside the given range results in the error `INVALID_VALUE`. The same versions of the command exist as for `PixelStore`, and the same rules apply to accepting and converting passed values to set parameters.

The pixel map lookup tables are set with

```c
void PixelMap{ui us f}{v( enum map, size i size, T values );
```

`map` is a symbolic map name, indicating the map to set, `size` indicates the size of the map, and `values` refers to an array of `size` map values.

The entries of a table may be specified using one of three types: single-precision floating-point, unsigned short integer, or unsigned integer, depending on which of the three versions of `PixelMap` is called. A table entry is converted to the appropriate type when it is specified. An entry giving a color component value is converted according to table 2.10 and then clamped to the range $[0, 1]$. An entry giving a color index value is converted from an unsigned short integer or unsigned...
integer to floating-point. An entry giving a stencil index is converted from single-precision floating-point to an integer by rounding to nearest. The various tables and their initial sizes and entries are summarized in table 3.3. A table that takes an index as an address must have \( \text{size} = 2^n \) or the error `INVALID_VALUE` results. The maximum allowable \( \text{size} \) of each table is specified by the implementation dependent value `MAX_PIXEL_MAP_TABLE`, but must be at least 32 (a single maximum applies to all tables). The error `INVALID_VALUE` is generated if a \( \text{size} \) larger than the implemented maximum, or less than one, is given to `PixelMap`.

If a pixel unpack buffer is bound (as indicated by a non-zero value of `PIXEL_UNPACK_BUFFER_BINDING`), `values` is an offset into the pixel unpack buffer; otherwise, `values` is a pointer to client memory. All pixel storage and pixel transfer modes are ignored when specifying a pixel map. \( n \) machine units are read where \( n \) is the \( \text{size} \) of the pixel map times the size of a `float`, `uint`, or `ushort` datum in basic machine units, depending on the respective `PixelMap` version. If a pixel unpack buffer object is bound and \( data + n \) is greater than the size of the pixel buffer, an `INVALID_OPERATION` error results. If a pixel unpack buffer object is bound and `values` is not evenly divisible by the number of basic machine units needed to store in memory a `float`, `uint`, or `ushort` datum depending on their respective `PixelMap` version, an `INVALID_OPERATION` error results.

**Color Table Specification**

Color lookup tables are specified with

---

Table 3.3: `PixelMap` parameters.

<table>
<thead>
<tr>
<th>Map Name</th>
<th>Address</th>
<th>Value</th>
<th>Init. Size</th>
<th>Init. Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PIXEL_MAP_I_TO_I</code></td>
<td>color idx</td>
<td>color idx</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td><code>PIXEL_MAP_S_TO_S</code></td>
<td>stencil idx</td>
<td>stencil idx</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><code>PIXEL_MAP_I_TO_R</code></td>
<td>color idx</td>
<td>R</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td><code>PIXEL_MAP_I_TO_G</code></td>
<td>color idx</td>
<td>G</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td><code>PIXEL_MAP_I_TO_B</code></td>
<td>color idx</td>
<td>B</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td><code>PIXEL_MAP_I_TO_A</code></td>
<td>color idx</td>
<td>A</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td><code>PIXEL_MAP_R_TO_R</code></td>
<td>R</td>
<td>R</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td><code>PIXEL_MAP_G_TO_G</code></td>
<td>G</td>
<td>G</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td><code>PIXEL_MAP_B_TO_B</code></td>
<td>B</td>
<td>B</td>
<td>1</td>
<td>0.0</td>
</tr>
<tr>
<td><code>PIXEL_MAP_A_TO_A</code></td>
<td>A</td>
<td>A</td>
<td>1</td>
<td>0.0</td>
</tr>
</tbody>
</table>
Table 3.4: Color table names. Regular tables have associated image data. Proxy tables have no image data, and are used only to determine if an image can be loaded into the corresponding regular table.

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLOR_TABLE</td>
<td>regular</td>
</tr>
<tr>
<td>POST_CONVOLUTION_COLOR_TABLE</td>
<td></td>
</tr>
<tr>
<td>POST_COLOR_MATRIX_COLOR_TABLE</td>
<td></td>
</tr>
<tr>
<td>PROXY_COLOR_TABLE</td>
<td>proxy</td>
</tr>
<tr>
<td>PROXY_POST_CONVOLUTION_COLOR_TABLE</td>
<td></td>
</tr>
<tr>
<td>PROXY_POST_COLOR_MATRIX_COLOR_TABLE</td>
<td></td>
</tr>
</tbody>
</table>

```c
void ColorTable( enum target, enum internalformat,
sizei width, enum format, enum type, void *data );
```

target must be one of the regular color table names listed in table 3.4 to define the table. A proxy table name is a special case discussed later in this section. width, format, type, and data specify an image in memory with the same meaning and allowed values as the corresponding arguments to DrawPixels (see section 3.7.4), with height taken to be 1. The maximum allowable width of a table is implementation-dependent, but must be at least 32. The formats COLOR_INDEX, DEPTH_COMPONENT, DEPTH_STENCIL, and_STENCIL_INDEX and the type BITMAP are not allowed.

The specified image is taken from memory and processed just as if DrawPixels were called, stopping after the final expansion to RGBA. The R, G, B, and A components of each pixel are then scaled by the four COLOR_TABLE_SCALE parameters and biased by the four COLOR_TABLE_BIAS parameters. These parameters are set by calling ColorTableParameterfv as described below. If fragment color clamping is enabled or internalformat is fixed-point, components are clamped to [0, 1]. Otherwise, components are not modified.

Components are then selected from the resulting R, G, B, and A values to obtain a table with the base internal format specified by (or derived from) internalformat, in the same manner as for textures (section 3.9.1). internalformat must be one of the formats in table 3.15 or tables 3.16-3.18, with the exception of the RED, RG, DEPTH_COMPONENT, and DEPTH_STENCIL base and sized internal formats in those tables, all sized internal formats with non-fixed internal data types (see section 3.9), and sized internal format RGB9_E5.

The color lookup table is redefined to have width entries, each with the specified internal format. The table is formed with indices 0 through width – 1. Table
location \(i\) is specified by the \(i\)th image pixel, counting from zero.

The error `INVALID_VALUE` is generated if \(width\) is not zero or a non-negative power of two. The error `TABLE_TOO_LARGE` is generated if the specified color lookup table is too large for the implementation.

The scale and bias parameters for a table are specified by calling

```c
void ColorTableParameter\{if\}v( enum target, enum pname,
    T params );
```

\(target\) must be a regular color table name. \(pname\) is one of `COLOR_TABLE_SCALE` or `COLOR_TABLE_BIAS`. \(params\) points to an array of four values: red, green, blue, and alpha, in that order.

A GL implementation may vary its allocation of internal component resolution based on any `ColorTable` parameter, but the allocation must not be a function of any other factor, and cannot be changed once it is established. Allocations must be invariant; the same allocation must be made each time a color table is specified with the same parameter values. These allocation rules also apply to proxy color tables, which are described later in this section.

### Alternate Color Table Specification Commands

Color tables may also be specified using image data taken directly from the frame-buffer, and portions of existing tables may be respecified.

The command

```c
void CopyColorTable( enum target, enum internalformat,
    int x, int y,sizei width );
```

defines a color table in exactly the manner of `ColorTable`, except that table data are taken from the framebuffer, rather than from client memory. \(target\) must be a regular color table name. \(x\), \(y\), and \(width\) correspond precisely to the corresponding arguments of `CopyPixels` (refer to section 4.3.3); they specify the image’s \(width\) and the lower left \((x,y)\) coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to `CopyPixels` with argument `type` set to `COLOR` and `height` set to 1, stopping after the final expansion to RGBA.

Subsequent processing is identical to that described for `ColorTable`, beginning with scaling by `COLOR_TABLE_SCALE`. Parameters `target`, `internalformat` and `width` are specified using the same values, with the same meanings, as the equivalent arguments of `ColorTable`. `format` is taken to be `RGBA`.

Two additional commands,
3.7. PIXEL RECTANGLES

void ColorSubTable(enum target, sizei start, sizei count, enum format, enum type, void *data);
void CopyColorSubTable( enum target, sizei start, int x, int y, sizei count );

respecify only a portion of an existing color table. No change is made to the internalformat or width parameters of the specified color table, nor is any change made to table entries outside the specified portion. target must be a regular color table name.

ColorSubTable arguments format, type, and data match the corresponding arguments to ColorTable, meaning that they are specified using the same values, and have the same meanings. Likewise, CopyColorSubTable arguments x, y, and count match the x, y, and width arguments of CopyColorTable. Both of the ColorSubTable commands interpret and process pixel groups in exactly the manner of their ColorTable counterparts, except that the assignment of R, G, B, and A pixel group values to the color table components is controlled by the internalformat of the table, not by an argument to the command.

Arguments start and count of ColorSubTable and CopyColorSubTable specify a subregion of the color table starting at index start and ending at index start + count – 1. Counting from zero, the nth pixel group is assigned to the table entry with index count + n. The error INVALID_VALUE is generated if start + count > width.

Calling CopyColorTable or CopyColorSubTable 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).

Color Table State and Proxy State

The state necessary for color tables can be divided into two categories. For each of the three tables, there is an array of values. Each array has associated with it a width, an integer describing the internal format of the table, six integer values describing the resolutions of each of the red, green, blue, alpha, luminance, and intensity components of the table, and two groups of four floating-point numbers to store the table scale and bias. Each initial array is null (zero width, internal format RGBA, with zero-sized components). The initial value of the scale parameters is (1,1,1,1) and the initial value of the bias parameters is (0,0,0,0).

In addition to the color lookup tables, partially instantiated proxy color lookup tables are maintained. Each proxy table includes width and internal format state values, as well as state for the red, green, blue, alpha, luminance, and intensity...
component resolutions. Proxy tables do not include image data, nor do they include scale and bias parameters. When \texttt{ColorTable} is executed with \texttt{target} specified as one of the proxy color table names listed in table 3.4, the proxy state values of the table are recomputed and updated. If the table is too large, no error is generated, but the proxy format, width and component resolutions are set to zero. If the color table would be accommodated by \texttt{ColorTable} called with \texttt{target} set to the corresponding regular table name (\texttt{COLOR\_TABLE} is the regular name corresponding to \texttt{PROXY\_COLOR\_TABLE}, for example), the proxy state values are set exactly as though the regular table were being specified. Calling \texttt{ColorTable} with a proxy \texttt{target} has no effect on the image or state of any actual color table.

There is no image associated with any of the proxy targets. They cannot be used as color tables, and they must never be queried using \texttt{GetColorTable}. The error \texttt{INVALID\_ENUM} is generated if this is attempted.

\textbf{Convolution Filter Specification}

A two-dimensional convolution filter image is specified by calling

\begin{verbatim}
void ConvolutionFilter2D( enum target, enum internalformat,
    sizei width, sizei height, enum format, enum type,
    void *data );
\end{verbatim}

\texttt{target} must be \texttt{CONVOLUTION\_2D}. \texttt{width}, \texttt{height}, \texttt{format}, \texttt{type}, and \texttt{data} specify an image in memory with the same meaning and allowed values as the corresponding parameters to \texttt{DrawPixels}. The formats \texttt{COLOR\_INDEX}, \texttt{DEPTH\_COMPONENT}, \texttt{DEPTH\_STENCIL}, and \texttt{STENCIL\_INDEX} and the type \texttt{BITMAP} are not allowed.

The specified image is extracted from memory and processed just as if \texttt{DrawPixels} were called, stopping after the final expansion to RGBA. The R, G, B, and A components of each pixel are then scaled by the four two-dimensional \texttt{CONVOLUTION\_FILTER\_SCALE} parameters and biased by the four two-dimensional \texttt{CONVOLUTION\_FILTER\_BIAS} parameters. These parameters are set by calling \texttt{ConvolutionParameterfv} as described below. No clamping takes place at any time during this process.

Components are then selected from the resulting R, G, B, and A values to obtain a table with the base \texttt{internal format} specified by (or derived from) \texttt{internalformat}, in the same manner as for textures (section 3.9.1). \texttt{internalformat} accepts \texttt{framebuffer\_object}.

The red, green, blue, alpha, luminance, and/or intensity components of the pixels are stored in floating point, rather than integer format. They form a two-dimensional image indexed with coordinates $i, j$ such that $i$ increases from left to right.
right, starting at zero, and $j$ increases from bottom to top, also starting at zero. Image location $i, j$ is specified by the $N$th pixel, counting from zero, where

$$N = i + j \times \text{width}$$

The error INVALID.VALUE is generated if width or height is greater than the maximum supported value. These values are queried with GetConvolutionParameteriv, setting target to CONVOLUTION_2D and pname to MAX_CONVOLUTION_WIDTH or MAX_CONVOLUTION_HEIGHT, respectively.

The scale and bias parameters for a two-dimensional filter are specified by calling

```c
void ConvolutionParameter{if}v( enum target, enum pname,
    T params );
```

with target CONVOLUTION_2D. pname is one of CONVOLUTION_FILTER_SCALE or CONVOLUTION_FILTER_BIAS. params points to an array of four values: red, green, blue, and alpha, in that order.

A one-dimensional convolution filter is defined using

```c
void ConvolutionFilter1D( enum target, enum internalformat,
    sizei width, enum format, enum type, void *data );
```

$target$ must be CONVOLUTION_1D. $internalformat$, width, $format$, and $type$ have identical semantics and accept the same values as do their two-dimensional counterparts. $data$ must point to a one-dimensional image, however.

The image is extracted from memory and processed as if ConvolutionFilter2D were called with a height of 1, except that it is scaled and biased by the one-dimensional CONVOLUTION_FILTER_SCALE and CONVOLUTION_FILTER_BIAS parameters. These parameters are specified exactly as the two-dimensional parameters, except that ConvolutionParameterfv is called with target CONVOLUTION_1D.

The image is formed with coordinates $i$ such that $i$ increases from left to right, starting at zero. Image location $i$ is specified by the $i$th pixel, counting from zero.

The error INVALID.VALUE is generated if width is greater than the maximum supported value. This value is queried using GetConvolutionParameteriv, setting target to CONVOLUTION_1D and pname to MAX_CONVOLUTION_WIDTH.

Special facilities are provided for the definition of two-dimensional separable filters – filters whose image can be represented as the product of two one-dimensional images, rather than as full two-dimensional images. A two-dimensional separable convolution filter is specified with
void SeparableFilter2D( enum target, enum internalformat, sizei width, sizei height, enum format, enum type, void *row, void *column );

target must be SEPARABLE_2D. internalformat specifies the formats of the table entries of the two one-dimensional images that will be retained. row points to a width pixel wide image of the specified format and type. column points to a height pixel high image, also of the specified format and type.

The two images are extracted from memory and processed as if ConvolutionFilter1D were called separately for each, except that each image is scaled and biased by the two-dimensional separable CONVOLUTION_FILTER_SCALE and CONVOLUTION_FILTER_BIAS parameters. These parameters are specified exactly as the one-dimensional and two-dimensional parameters, except that ConvolutionParameteriv is called with target SEPARABLE_2D.

Alternate Convolution Filter Specification Commands

One and two-dimensional filters may also be specified using image data taken directly from the framebuffer.

The command

void CopyConvolutionFilter2D( enum target, enum internalformat, int x, int y, sizei width, sizei height );

defines a two-dimensional filter in exactly the manner of ConvolutionFilter2D, except that image data are taken from the framebuffer, rather than from client memory. target must be CONVOLUTION_2D. x, y, width, and height correspond precisely to the corresponding arguments of CopyPixels (refer to section 4.3.3); they specify the image’s width and height, and the lower left (x, y) coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to CopyPixels with argument type set to COLOR, stopping after the final expansion to RGBA.

Subsequent processing is identical to that described for ConvolutionFilter2D, beginning with scaling by CONVOLUTION_FILTER_SCALE. Parameters target, internalformat, width, and height are specified using the same values, with the same meanings, as the equivalent arguments of ConvolutionFilter2D. format is taken to be RGBA.

The command

void CopyConvolutionFilter1D( enum target, enum internalformat, int x, int y, sizei width );
defines a one-dimensional filter in exactly the manner of \texttt{ConvolutionFilter1D},
except that image data are taken from the framebuffer, rather than from client mem-
ory. \texttt{target} must be \texttt{CONVOLUTION\_1D\_x, y, and width} correspond precisely to the
corresponding arguments of \texttt{CopyPixels} (refer to section 4.3.3); they specify the
image’s \texttt{width} 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 \texttt{CopyPixels} with argument \texttt{type} set to \texttt{COLOR} and \texttt{height} set to 1,
stopping after the final expansion to RGBA.

Subsequent processing is identical to that described for \texttt{ConvolutionFilter1D},
beginning with scaling by \texttt{CONVOLUTION\_FILTER\_SCALE}. Parameters \texttt{target}, \texttt{internalformat},
and \texttt{width} are specified using the same values, with the same mean-
ings, as the equivalent arguments of \texttt{ConvolutionFilter2D}. \texttt{format} is taken to be
\texttt{RGBA}.

Calling \texttt{CopyConvolutionFilter1D} or \texttt{CopyConvolutionFilter2D} will re-
sult in an \texttt{INVALID\_FRAMEBUFFER\_OPERATION} error if the object bound to
\texttt{READ\_FRAMEBUFFER\_BINDING} is not framebuffer complete (see section 4.4.4).

Convolution Filter State

The required state for convolution filters includes a one-dimensional image array,
two one-dimensional image arrays for the separable filter, and a two-dimensional
image array. Each filter has associated with it a width and height (two-dimensional
and separable only), an integer describing the internal format of the filter, and two
groups of four floating-point numbers to store the filter scale and bias.

Each initial convolution filter is null (zero width and height, internal format
\texttt{RGBA}, with zero-sized components). The initial value of all scale parameters is
(1,1,1,1) and the initial value of all bias parameters is (0,0,0,0).

Color Matrix Specification

Setting the matrix mode to \texttt{COLOR} causes the matrix operations described in sec-
tion 2.12.2 to apply to the top matrix on the color matrix stack. All matrix opera-
tions have the same effect on the color matrix as they do on the other matrices.

Histogram Table Specification

The histogram table is specified with

\begin{verbatim}
void Histogram( enum target, sizei width,
               enum internalformat, boolean sink );
\end{verbatim}
target must be HISTOGRAM if a histogram table is to be specified. target value PROXY,HISTOGRAM is a special case discussed later in this section. width specifies the number of entries in the histogram table, and internalformat specifies the format of each table entry. The maximum allowable width of the histogram table is implementation-dependent, but must be at least 32. sink specifies whether pixel groups will be consumed by the histogram operation (TRUE) or passed on to the minmax operation (FALSE).

If no error results from the execution of Histogram, the specified histogram table is redefined to have width entries, each with the specified internal format. The entries are indexed 0 through width − 1. Each component in each entry is set to zero. The values in the previous histogram table, if any, are lost.

The error INVALID_VALUE is generated if width is not zero or a non-negative power of two. The error TABLE_TOO_LARGE is generated if the specified histogram table is too large for the implementation. internalformat accepts the same values as the corresponding argument of ColorTable, with the exception of the values 1, 2, 3, and 4.

A GL implementation may vary its allocation of internal component resolution based on any Histogram parameter, but the allocation must not be a function of any other factor, and cannot be changed once it is established. In particular, allocations must be invariant; the same allocation must be made each time a histogram is specified with the same parameter values. These allocation rules also apply to the proxy histogram, which is described later in this section.

**Histogram State and Proxy State**

The state necessary for histogram operation is an array of values, with which is associated a width, an integer describing the internal format of the histogram, five integer values describing the resolutions of each of the red, green, blue, alpha, and luminance components of the table, and a flag indicating whether or not pixel groups are consumed by the operation. The initial array is null (zero width, internal format RGBA, with zero-sized components). The initial value of the flag is false.

In addition to the histogram table, a partially instantiated proxy histogram table is maintained. It includes width, internal format, and red, green, blue, alpha, and luminance component resolutions. The proxy table does not include image data or the flag. When Histogram is executed with target set to PROXY,HISTOGRAM, the proxy state values are recomputed and updated. If the histogram array is too large, no error is generated, but the proxy format, width, and component resolutions are set to zero. If the histogram table would be accommodated by Histogram called with target set to HISTOGRAM, the proxy state values are set exactly as though the actual histogram table were being specified. Calling Histogram with target
PROXY_HISTOGRAM has no effect on the actual histogram table.

There is no image associated with PROXY_HISTOGRAM. It cannot be used as a histogram, and its image must never queried using GetHistogram. The error INVALID_ENUM results if this is attempted.

### Minmax Table Specification

The minmax table is specified with

```c
void Minmax(enum target, enum internalformat, boolean sink);
```

*target* must be MINMAX. *internalformat* specifies the format of the table entries. *sink* specifies whether pixel groups will be consumed by the minmax operation (TRUE) or passed on to final conversion (FALSE).

*internalformat* accepts the same values as the corresponding argument of ColorTable, with the exception of the values 1, 2, 3, and 4, as well as the INTENSITY base and sized internal formats. The resulting table always has 2 entries, each with values corresponding only to the components of the internal format.

The state necessary for minmax operation is a table containing two elements (the first element stores the minimum values, the second stores the maximum values), an integer describing the internal format of the table, and a flag indicating whether or not pixel groups are consumed by the operation. The initial state is a minimum table entry set to the maximum representable value and a maximum table entry set to the minimum representable value. Internal format is set to RGBA and the initial value of the flag is false.

### 3.7.4 Rasterization of Pixel Rectangles

The process of drawing pixels encoded in buffer object or client memory is diagrammed in figure 3.7. We describe the stages of this process in the order in which they occur.

Pixels are drawn using

```c
void DrawPixels(sizei width, sizei height, enum format, enum type, void *data);
```

*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 drawn. *data* refers to the data to be drawn. The correspondence between the *type* token values and the GL data types they indicate is

Version 3.0 (with changes marked - August 11, 2008
Figure 3.7. Operation of DrawPixels. Output is RGBA pixels if the GL is in RGBA mode, color index pixels otherwise. Operations in dashed boxes may be enabled or disabled. RGBA and color index pixel paths are shown; depth and stencil pixel paths are not shown.
3.7. PIXEL RECTANGLES

given in table 3.5. If the GL is in color index mode and format is not one of COLOR_INDEX, STENCIL_INDEX, DEPTH_COMPONENT, or DEPTH_STENCIL, then the error INVALID_OPERATION occurs. Results of rasterization are undefined if any of the selected draw buffers of the draw framebuffer have an integer format and no fragment shader is active. If format contains integer components, as shown in table 3.6, an INVALID_OPERATION error is generated. If type is BITMAP and format is not COLOR_INDEX or STENCIL_INDEX then the error INVALID_ENUM occurs. 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 3.6 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.

Calling DrawPixels will result in an INVALID_FRAMEBUFFER_OPERATION error if the object bound to DRAW_FRAMEBUFFER_BINDING is not framebuffer complete (see section 4.4.4).

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 3.6 summarizes the format of groups obtained from memory; it also indicates those formats that yield indices and those that yield floating-point or integer components.

If a pixel unpack buffer is bound (as indicated by a non-zero value of PIXEL_UNPACK_BUFFER_BINDING), data is an offset into the pixel unpack buffer and the pixels are unpacked from the buffer relative to this offset; otherwise, data is a pointer to client memory and the pixels are unpacked from client memory relative to the pointer. If a pixel unpack buffer object is bound and unpacking the pixel data according to the process described below would access memory beyond the size of the pixel unpack buffer’s memory size, an INVALID_OPERATION error results. If a pixel unpack buffer object is bound and data is not evenly divisible by the number of basic machine units needed to store in memory the corresponding GL data type from table 3.5 for the type parameter, 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 3.7. The modified bit orderings are defined only if the GL data type

Version 3.0 (with changes marked - August 11, 2008
### Table 3.5: DrawPixels and ReadPixels 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 3.7.4.

<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>BITMAP</td>
<td>ubyte</td>
<td>Yes</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>

Version 3.0 (with changes marked - August 11, 2008)
### 3.7. PIXEL RECTANGLES

Table 3.6: **DrawPixels** and **ReadPixels** formats. The second column gives a description of and the number and order of elements in a group. Unless specified as an index, formats yield components. Components are floating-point unless prefixed with the letter 'i', which indicates they are integer.

<table>
<thead>
<tr>
<th>Format Name</th>
<th>Element Meaning and Order</th>
<th>Target Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLOR_INDEX</td>
<td>Color Index</td>
<td>Color</td>
</tr>
<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>ALPHA</td>
<td>A</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>LUMINANCE</td>
<td>Luminance</td>
<td>Color</td>
</tr>
<tr>
<td>LUMINANCE_ALPHA</td>
<td>Luminance, 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>ALPHA_INTEGER</td>
<td>iA</td>
<td>Color</td>
</tr>
<tr>
<td>RG_INTEGER</td>
<td>iR, iG</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>

Version 3.0 (with changes marked - August 11, 2008)
Table 3.7: 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.

<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>

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 the pointer passed to DrawPixels. If the value of UNPACK_ROW_LENGTH is not positive, then the number of groups in a row is width; otherwise the number of groups is UNPACK_ROW_LENGTH. If p indicates the location in memory of the first element of the first row, then the first element of the Nth row is indicated by

\[ p + Nk \] (3.12)

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

\[ k = \begin{cases} nl & s \geq a, \\ a/s \left\lfloor snl/a \right\rfloor & s < a \end{cases} \] (3.13)

where n is the number of elements in a group, l is the number of groups in the row, a is the value of UNPACK_ALIGNMENT, and s is the size, in units of GL ubyte, of an element. If the number of bits per element is not 1, 2, 4, or 8 times the number of bits in a GL ubyte, then \( k = nl \) for all values of a.

There is a mechanism for selecting a sub-rectangle of groups from a larger containing rectangle. This mechanism relies on three integer parameters: UNPACK_ROW_LENGTH, UNPACK_SKIP_ROWS, and UNPACK_SKIP_PIXELS. Before obtaining the first group from memory, the pointer supplied to DrawPixels is effectively 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 3.8.

Calling DrawPixels with a type matching one of the types in table 3.8 is a packed_float.
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 the type is float, two 32-bit words; the first word contains the float component, and the second word contains packed 24-bit and 8-bit components. 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 3.8. The error INVALID_OPERATION is generated if a mismatch occurs. This constraint also holds for all other functions that accept or return pixel data using type and format parameters to define the type and format of that data.

Bitfield locations of the first, second, third, and fourth components of each packed pixel type are illustrated in tables 3.9, 3.10, and 3.11. 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.
### Table 3.8: Packed pixel formats.

<table>
<thead>
<tr>
<th>Type Parameter</th>
<th>GL Data Type</th>
<th>Number of Components</th>
<th>Matching Pixel Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNSIGNED_BYTE_3_3_2</td>
<td>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>

Version 3.0 (with changes marked - August 11, 2008)
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 3.9: UNSIGNED_BYTE formats. Bit numbers are indicated for each component.
### 3.7. PIXEL RECTANGLES

<table>
<thead>
<tr>
<th>Unsigned Short Formats</th>
<th>Component Order</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UNSIGNED_SHORT_5_6_5:</strong></td>
<td>1st Component, 2nd, 3rd</td>
</tr>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>UNSIGNED_SHORT_5_6_5_REV:</strong></th>
<th>3rd, 2nd, 1st Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>UNSIGNED_SHORT_4_4_4:</strong></th>
<th>1st Component, 2nd, 3rd, 4th</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>UNSIGNED_SHORT_4_4_4_REV:</strong></th>
<th>4th, 3rd, 2nd, 1st Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>UNSIGNED_SHORT_5_5_5_1:</strong></th>
<th>1st Component, 2nd, 3rd, 4th</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>UNSIGNED_SHORT_1_5_5_5_REV:</strong></th>
<th>4th, 3rd, 2nd, 1st Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

**Table 3.10:** **UNSIGNED_SHORT** formats

---

Version 3.0 (with changes marked - August 11, 2008)
### Table 3.11: UNSIGNED\_INT formats

<table>
<thead>
<tr>
<th>UNSIGNED_INT_0_8_8_8:</th>
<th>1st Component</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
</tr>
</thead>
<tbody>
<tr>
<td>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</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UNSIGNED_INT_8_8_8_REV:</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>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</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UNSIGNED_INT_10_10_10_2:</th>
<th>1st Component</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
</tr>
</thead>
<tbody>
<tr>
<td>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</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UNSIGNED_INT_2_10_10_REV:</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>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</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UNSIGNED_INT_24_8:</th>
<th>1st Component</th>
<th>2nd</th>
</tr>
</thead>
<tbody>
<tr>
<td>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</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UNSIGNED_INT_10F_11F_11F_REV:</th>
<th>3rd</th>
<th>2nd</th>
<th>1st Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>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</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UNSIGNED_INT_5_9_9_9_REV:</th>
<th>4th</th>
<th>3rd</th>
<th>2nd</th>
<th>1st Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>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</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Version 3.0 (with changes marked - August 11, 2008)
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 3.12: Packed pixel field assignments.

The assignment of component to fields in the packed pixel is as described in table 3.12.

Byte swapping, if enabled, is performed before the component 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.

Calling DrawPixels with 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 3.11. The 1st, 2nd, and 3rd components are called |f_red| (11 bits), |f_green| (11 bits), and |f_blue| (10 bits) respectively.

|f_red| and |f_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_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.

Calling DrawPixels with 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 3.11. The 1st, 2nd, 3rd, and 4th components are called |p_red|, |p_green|, |p_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:

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

where \(B = 15\) (the exponent bias) and \(N = 9\) (the number of mantissa bits).

Version 3.0 (with changes marked - August 11, 2008)
3.7. PIXEL RECTANGLES

Calling DrawPixels with a type of BITMAP is a special case in which the data are a series of GL ubyte values. Each ubyte value specifies 8 1-bit elements with its 8 least-significant bits. The 8 single-bit elements are ordered from most significant to least significant if the value of UNPACK_LSB_FIRST is FALSE; otherwise, the ordering is from least significant to most significant. The values of bits other than the 8 least significant in each ubyte are not significant.

The first element of the first row is the first bit (as defined above) of the ubyte pointed to by the pointer passed to DrawPixels. The first element of the second row is the first bit (again as defined above) of the ubyte at location \( p + k \), where \( k \) is computed as

\[
k = a \lceil \frac{l}{8a} \rceil
\]

There is a mechanism for selecting a sub-rectangle of elements from a BITMAP image as well. Before obtaining the first element from memory, the pointer supplied to DrawPixels is effectively advanced by UNPACK_SKIP_ROWS \( \times k \) ubytes. Then UNPACK_SKIP_PIXELS 1-bit elements are ignored, and the subsequent width 1-bit elements are obtained, without advancing the ubyte pointer, after which the pointer is advanced by \( k \) ubytes. height sets of width elements are obtained this way.

### 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 according to the appropriate formula in table 2.10 (section 2.19). For packed pixel types, each element in the group is converted by computing \( c / (2^N - 1) \), where \( c \) is the unsigned integer value of the bitfield containing the element and \( N \) is the number of bits in the bitfield.

### Conversion to RGB

This step is applied only if the format is LUMINANCE or LUMINANCE_ALPHA. If the format is LUMINANCE, then each group of one element is converted to a group of R, G, and B (three) elements by copying the original single element into each of the three new elements. If the format is LUMINANCE_ALPHA, then each group of two elements is converted to a group of R, G, B, and A (four) elements by copying...
the first original element into each of the first three new elements and copying the second original element to the A (fourth) new element.

**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.

**Pixel Transfer Operations**

This step is actually a sequence of steps. Because the pixel transfer operations are performed equivalently during the drawing, copying, and reading of pixels, and during the specification of texture images (either from memory or from the framebuffer), they are described separately in section 3.7.5. After the processing described in that section is completed, groups are processed as described in the following sections.

**Final Conversion**

For a color index, final conversion consists of masking the bits of the index to the left of the binary point by \(2^n - 1\), where \(n\) is the number of bits in an index buffer. For integer RGBA components, no conversion is performed. For floating-point RGBA components, if fragment color clamping is enabled, each element is clamped to \([0, 1]\), and may be converted to fixed-point according to the rules given in section 2.19.9. If fragment color clamping is disabled, RGBA components are unmodified. Fragment color clamping is controlled using ClampColor, as described in section 2.19.6, with a target of CLAMP_FRAGMENT_COLOR.

For a depth component, an element is processed according to the depth buffer’s representation. For fixed-point depth buffers, the element is first clamped to the range \([0, 1]\) and then converted to fixed-point as if it were a window z value (see section 2.12.1). Conversion is not necessary when the depth buffer uses a floating-point representation, but clamping is.

Stencil indices are masked by \(2^n - 1\), where \(n\) is the number of bits in the stencil buffer.

The state required for fragment color clamping is a three-valued integer. The initial value of fragment color clamping is FIXED_ONLY.

Version 3.0 (with changes marked - August 11, 2008)
3.7. PIXEL RECTANGLES

Conversion to Fragments

The conversion of a group to fragments is controlled with

```c
void PixelZoom( float z_x, float z_y );
```

Let \((x_{rp}, y_{rp})\) be the current raster position (section 2.18). (If the current raster position is invalid, then `DrawPixels` is ignored; pixel transfer operations do not update the histogram or minmax tables, and no fragments are generated. However, the histogram and minmax tables are updated even if the corresponding fragments are later rejected by the pixel ownership (section 4.1.1) or scissor (section 4.1.2) tests.) If a particular group (index or components) is the \(n\)th in a row and belongs to the \(m\)th row, consider the region in window coordinates bounded by the rectangle with corners

\[
(x_{rp} + z_x n, y_{rp} + z_y m) \quad \text{and} \quad (x_{rp} + z_x (n + 1), y_{rp} + z_y (m + 1))
\]

(either \(z_x\) or \(z_y\) may be negative). A fragment representing group \((n, m)\) is produced for each framebuffer pixel inside, or on the bottom or left boundary, of this rectangle.

A fragment arising from a group consisting of color data takes on the color index or color components of the group and the current raster position’s associated depth value, while a fragment arising from a depth component takes that component’s depth value and the current raster position’s associated color index or color components. In both cases, the fog coordinate is taken from the current raster position’s associated raster distance, the secondary color is taken from the current raster position’s associated secondary color, and texture coordinates are taken from the current raster position’s associated texture coordinates. Groups arising from `DrawPixels` with a format of `DEPTH_STENCIL` or `STENCIL_INDEX` are treated specially and are described in section 4.3.1.

3.7.5 Pixel Transfer Operations

The GL defines six kinds of pixel groups:

1. **Floating-point RGBA component:** Each group comprises four color components in floating-point format: red, green, blue, and alpha.
2. **Integer RGBA component:** Each group comprises four color components in integer format: red, green, blue, and alpha.
3. **Depth component:** Each group comprises a single depth component.
3.7. PIXEL RECTANGLES

4. **Color index**: Each group comprises a single color index.

5. **Stencil index**: Each group comprises a single stencil index.

6. **Depth/stencil**: Each group comprises a single depth component and a single stencil index.

Each operation described in this section is applied sequentially to each pixel group in an image. Many operations are applied only to pixel groups of certain kinds; if an operation is not applicable to a given group, it is skipped. None of the operations defined in this section affect integer RGBA component pixel groups.

### Arithmetic on Components

This step applies only to RGBA component and depth component groups, and to the depth components in depth/stencil groups. Each component is multiplied by an appropriate signed scale factor: \( \text{RED\_SCALE} \) for an R component, \( \text{GREEN\_SCALE} \) for a G component, \( \text{BLUE\_SCALE} \) for a B component, and \( \text{ALPHA\_SCALE} \) for an A component, or \( \text{DEPTH\_SCALE} \) for a depth component. Then the result is added to the appropriate signed bias: \( \text{RED\_BIAS} \), \( \text{GREEN\_BIAS} \), \( \text{BLUE\_BIAS} \), \( \text{ALPHA\_BIAS} \), or \( \text{DEPTH\_BIAS} \).

### Arithmetic on Indices

This step applies only to color index and stencil index groups, and to the stencil indices in depth/stencil groups. If the index is a floating-point value, it is converted to fixed-point, with an unspecified number of bits to the right of the binary point and at least \( \lceil \log_2(\text{MAX\_PIXEL\_MAP\_TABLE}) \rceil \) bits to the left of the binary point. Indices that are already integers remain so; any fraction bits in the resulting fixed-point value are zero.

The fixed-point index is then shifted by \( |\text{INDEX\_SHIFT}| \) bits, left if \( \text{INDEX\_SHIFT} > 0 \) and right otherwise. In either case the shift is zero-filled. Then, the signed integer offset \( \text{INDEX\_OFFSET} \) is added to the index.

### RGBA to RGBA Lookup

This step applies only to RGBA component groups, and is skipped if \( \text{MAP\_COLOR} \) is \text{FALSE}. First, each component is clamped to the range \([0, 1]\). There is a table associated with each of the R, G, B, and A component elements: \( \text{PIXEL\_MAP\_R\_TO\_R} \) for R, \( \text{PIXEL\_MAP\_G\_TO\_G} \) for G, \( \text{PIXEL\_MAP\_B\_TO\_B} \) for B, and \( \text{PIXEL\_MAP\_A\_TO\_A} \) for A. Each element is multiplied by an integer one less than the size of the corresponding table, and, for each element, an address is found by rounding this value.
to the nearest integer. For each element, the addressed value in the corresponding table replaces the element.

**Color Index Lookup**

This step applies only to color index groups. If the GL command that invokes the pixel transfer operation requires that RGBA component pixel groups be generated, then a conversion is performed at this step. RGBA component pixel groups are required if

1. The groups will be rasterized, and the GL is in RGBA mode, or
2. The groups will be loaded as an image into texture memory, or
3. The groups will be returned to client memory with a format other than COLOR_INDEX.

If RGBA component groups are required, then the integer part of the index is used to reference 4 tables of color components: PIXEL_MAP_I_TO_R, PIXEL_MAP_I_TO_G, PIXEL_MAP_I_TO_B, and PIXEL_MAP_I_TO_A. Each of these tables must have $2^n$ entries for some integer value of $n$ ($n$ may be different for each table). For each table, the index is first rounded to the nearest integer; the result is ANDed with $2^n - 1$, and the resulting value used as an address into the table. The indexed value becomes an R, G, B, or A value, as appropriate. The group of four elements so obtained replaces the index, changing the group’s type to RGBA component.

If RGBA component groups are not required, and if MAP_COLOR is enabled, then the index is looked up in the PIXEL_MAP_I_TO_I table (otherwise, the index is not looked up). Again, the table must have $2^n$ entries for some integer $n$. The index is first rounded to the nearest integer; the result is ANDed with $2^n - 1$, and the resulting value used as an address into the table. The value in the table replaces the index. The floating-point table value is first rounded to a fixed-point value with unspecified precision. The group’s type remains color index.

**Stencil Index Lookup**

This step applies only to stencil index groups, and to the stencil indices in framebuffer_object depth/stencil groups. If MAP_STENCIL is enabled, then the index is looked up in the PIXEL_MAP_S_TO_S table (otherwise, the index is not looked up). The table must have $2^n$ entries for some integer $n$. The table value is first rounded to the nearest integer; the result is ANDed with $2^n - 1$, and the resulting value used as an address into the table. The integer value in the table replaces the index.

Version 3.0 (with changes marked - August 11, 2008)
Color Table Lookup

This step applies only to RGBA component groups. Color table lookup is only done if COLOR_TABLE is enabled. If a zero-width table is enabled, no lookup is performed.

The internal format of the table determines which components of the group will be replaced (see table 3.13). The components to be replaced are converted to indices by clamping to \([0, 1]\), multiplying by an integer one less than the width of the table, and rounding to the nearest integer. Components are replaced by the table entry at the index.

The required state is one bit indicating whether color table lookup is enabled or disabled. In the initial state, lookup is disabled.

Convolution

This step applies only to RGBA component groups. If CONVOLUTION_1D is enabled, the one-dimensional convolution filter is applied only to the one-dimensional texture images passed to TexImage1D, TexSubImage1D, CopyTexImage1D, and CopyTexSubImage1D. If CONVOLUTION_2D is enabled, the two-dimensional convolution filter is applied only to the two-dimensional images passed to DrawPixels, CopyPixels, ReadPixels, TexImage2D, TexSubImage2D, CopyTexImage2D, CopyTexSubImage2D, and CopyTexSubImage3D. If SEPARABLE_2D is enabled, and CONVOLUTION_2D is disabled, the separable two-dimensional convolution filter is instead applied these images.

The convolution operation is a sum of products of source image pixels and convolution filter pixels. Source image pixels always have four components: red,
3.7. PIXEL RECTANGLES

<table>
<thead>
<tr>
<th>Base Filter Format</th>
<th>R</th>
<th>G</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA</td>
<td>$R_s$</td>
<td>$G_s$</td>
<td>$B_s$</td>
<td>$A_s \ast A_f$</td>
</tr>
<tr>
<td>LUMINANCE</td>
<td>$R_s \ast L_f$</td>
<td>$G_s \ast L_f$</td>
<td>$B_s \ast L_f$</td>
<td>$A_s$</td>
</tr>
<tr>
<td>LUMINANCE, ALPHA</td>
<td>$R_s \ast L_f$</td>
<td>$G_s \ast L_f$</td>
<td>$B_s \ast L_f$</td>
<td>$A_s \ast A_f$</td>
</tr>
<tr>
<td>INTENSITY</td>
<td>$R_s \ast I_f$</td>
<td>$G_s \ast I_f$</td>
<td>$B_s \ast I_f$</td>
<td>$A_s \ast I_f$</td>
</tr>
<tr>
<td>RGB</td>
<td>$R_s \ast R_f$</td>
<td>$G_s \ast G_f$</td>
<td>$B_s \ast B_f$</td>
<td>$A_s$</td>
</tr>
<tr>
<td>RGBA</td>
<td>$R_s \ast R_f$</td>
<td>$G_s \ast G_f$</td>
<td>$B_s \ast B_f$</td>
<td>$A_s \ast A_f$</td>
</tr>
</tbody>
</table>

Table 3.14: Computation of filtered color components depending on filter image format. $C \ast F$ indicates the convolution of image component $C$ with filter $F$.

green, blue, and alpha, denoted in the equations below as $R_s$, $G_s$, $B_s$, and $A_s$. Filter pixels may be stored in one of five formats, with 1, 2, 3, or 4 components. These components are denoted as $R_f$, $G_f$, $B_f$, $A_f$, $L_f$, and $I_f$ in the equations below. The result of the convolution operation is the 4-tuple $R,G,B,A$. Depending on the internal format of the filter, individual color components of each source image pixel are convolved with one filter component, or are passed unmodified. The rules for this are defined in table 3.14.

The convolution operation is defined differently for each of the three convolution filters. The variables $W_f$ and $H_f$ refer to the dimensions of the convolution filter. The variables $W_s$ and $H_s$ refer to the dimensions of the source pixel image.

The convolution equations are defined as follows, where $C$ refers to the filtered result, $C_f$ refers to the one- or two-dimensional convolution filter, and $C_{\text{row}}$ and $C_{\text{column}}$ refer to the two one-dimensional filters comprising the two-dimensional separable filter. $C_s'$ depends on the source image color $C_s$ and the convolution border mode as described below. $C_r$, the filtered output image, depends on all of these variables and is described separately for each border mode. The pixel indexing nomenclature is described in the Convolution Filter Specification subsection of section 3.7.3.

One-dimensional filter:

$$C[i'] = \sum_{n=0}^{W_f-1} C_s'[i' + n] \ast C_f[n]$$

Two-dimensional filter:

$$C[i', j'] = \sum_{n=0}^{W_f-1} \sum_{m=0}^{H_f-1} C_s'[i' + n, j' + m] \ast C_f[n, m]$$

Two-dimensional separable filter:

$$C[i', j'] = C_{\text{row}}[i'] \ast C_{\text{column}}[j']$$
If $W_f$ of a one-dimensional filter is zero, then $C[i]$ is always set to zero. Likewise, if either $W_f$ or $H_f$ of a two-dimensional filter is zero, then $C[i, j]$ is always set to zero.

The convolution border mode for a specific convolution filter is specified by calling

```c
void ConvolutionParameter{if}( enum target, enum pname, T param );
```

where `target` is the name of the filter, `pname` is `CONVOLUTION_BORDER_MODE`, and `param` is one of `REDUCE`, `CONSTANT_BORDER` or `REPLICATE_BORDER`.

**Border Mode** REDUCE

The width and height of source images convolved with border mode REDUCE are reduced by $W_f - 1$ and $H_f - 1$, respectively. If this reduction would generate a resulting image with zero or negative width and/or height, the output is simply null, with no error generated. The coordinates of the image that results from a convolution with border mode REDUCE are zero through $W_s - W_f$ in width, and zero through $H_s - H_f$ in height. In cases where errors can result from the specification of invalid image dimensions, it is these resulting dimensions that are tested, not the dimensions of the source image. (A specific example is `TexImage1D` and `TexImage2D`, which specify constraints for image dimensions. Even if `TexImage1D` or `TexImage2D` is called with a null pixel pointer, the dimensions of the resulting texture image are those that would result from the convolution of the specified image).

When the border mode is REDUCE, $C'_s$ equals the source image color $C_s$ and $C_r$ equals the filtered result $C$.

For the remaining border modes, define $C_w = \lfloor W_f/2 \rfloor$ and $C_h = \lfloor H_f/2 \rfloor$. The coordinates $(C_w, C_h)$ define the center of the convolution filter.

**Border Mode** CONSTANT_BORDER

If the convolution border mode is CONSTANT_BORDER, the output image has the same dimensions as the source image. The result of the convolution is the same as if the source image were surrounded by pixels with the same color as the
current convolution border color. Whenever the convolution filter extends beyond one of the edges of the source image, the constant-color border pixels are used as input to the filter. The current convolution border color is set by calling `ConvolutionParameterfv` or `ConvolutionParameteriv` with `pname` set to `CONVOLUTION_BORDER_COLOR` and `params` containing four values that comprise the RGBA color to be used as the image border. Integer color components are interpreted linearly such that the largest positive integer maps to 1.0, and the smallest negative integer maps to -1.0. Floating point color components are not clamped when they are specified.

For a one-dimensional filter, the result color is defined by

\[ C_r[i] = C[i - C_w] \]

where \( C[i'] \) is computed using the following equation for \( C_s[i'] \):

\[ C_s[i'] = \begin{cases} C_s[i'], & 0 \leq i' < W_s \\ C_c, & \text{otherwise} \end{cases} \]

and \( C_c \) is the convolution border color.

For a two-dimensional or two-dimensional separable filter, the result color is defined by

\[ C_r[i, j] = C[i - C_w, j - C_h] \]

where \( C[i', j'] \) is computed using the following equation for \( C_s[i', j'] \):

\[ C_s[i', j'] = \begin{cases} C_s[i', j'], & 0 \leq i' < W_s, 0 \leq j' < H_s \\ C_c, & \text{otherwise} \end{cases} \]

**Border Mode** `REPLICATE_BORDER`  

The convolution border mode `REPLICATE_BORDER` also produces an output image with the same dimensions as the source image. The behavior of this mode is identical to that of the `CONSTANT_BORDER` mode except for the treatment of pixel locations where the convolution filter extends beyond the edge of the source image. For these locations, it is as if the outermost one-pixel border of the source image was replicated. Conceptually, each pixel in the leftmost one-pixel column of the source image is replicated \( C_w \) times to provide additional image data along the left edge, each pixel in the rightmost one-pixel column is replicated \( C_w \) times to provide additional image data along the right edge, and each pixel value in the top and bottom one-pixel rows is replicated to create \( C_h \) rows of image data along...
the top and bottom edges. The pixel value at each corner is also replicated in order to provide data for the convolution operation at each corner of the source image.

For a one-dimensional filter, the result color is defined by

$$C_r[i] = C[i - C_w]$$

where $$C[i']$$ is computed using the following equation for $$C_s[i']$$:

$$C_s[i'] = C_s[\text{clamp}(i', W_s)]$$

and the clamping function $$\text{clamp}(val, max)$$ is defined as

$$\text{clamp}(val, max) = \begin{cases} 
0, & val < 0 \\
val, & 0 \leq val < max \\
max - 1, & val \geq max
\end{cases}$$

For a two-dimensional or two-dimensional separable filter, the result color is defined by

$$C_r[i, j] = C[i - C_w, j - C_h]$$

where $$C[i', j']$$ is computed using the following equation for $$C_s[i', j']$$:

$$C_s[i', j'] = C_s[\text{clamp}(i', W_s), \text{clamp}(j', H_s)]$$

If a convolution operation is performed, each component of the resulting image is scaled by the corresponding PixelTransfer parameters: POST_CONVOLUTION_RED_SCALE for an R component, POST_CONVOLUTION_GREEN_SCALE for a G component, POST_CONVOLUTION_BLUE_SCALE for a B component, and POST_CONVOLUTION_ALPHA_SCALE for an A component. The result is added to the corresponding bias: POST_CONVOLUTION_RED_BIAS, POST_CONVOLUTION_GREEN_BIAS, POST_CONVOLUTION_BLUE_BIAS, or POST_CONVOLUTION_ALPHA_BIAS.

The required state is three bits indicating whether each of one-dimensional, two-dimensional, or separable two-dimensional convolution is enabled or disabled, an integer describing the current convolution border mode, and four floating-point values specifying the convolution border color. In the initial state, all convolution operations are disabled, the border mode is REDUCE, and the border color is $$(0, 0, 0, 0)$$.
3.7. PIXEL RECTANGLES

Post Convolution Color Table Lookup

This step applies only to RGBA component groups. Post convolution color table lookup is enabled or disabled by calling `Enable` or `Disable` with the symbolic constant `POST_CONVOLUTION_COLOR_TABLE`. The post convolution table is defined by calling `ColorTable` with a `target` argument of `POST_CONVOLUTION_COLOR_TABLE`. In all other respects, operation is identical to color table lookup, as defined earlier in section 3.7.5.

The required state is one bit indicating whether post convolution table lookup is enabled or disabled. In the initial state, lookup is disabled.

Color Matrix Transformation

This step applies only to RGBA component groups. The components are transformed by the color matrix. Each transformed component is multiplied by an appropriate signed scale factor: `POST_COLOR_MATRIX_RED_SCALE` for an R component, `POST_COLOR_MATRIX_GREEN_SCALE` for a G component, `POST_COLOR_MATRIX_BLUE_SCALE` for a B component, and `POST_COLOR_MATRIX_ALPHA_SCALE` for an A component. The result is added to a signed bias: `POST_COLOR_MATRIX_RED_BIAS`, `POST_COLOR_MATRIX_GREEN_BIAS`, `POST_COLOR_MATRIX_BLUE_BIAS`, or `POST_COLOR_MATRIX_ALPHA_BIAS`. The resulting components replace each component of the original group.

That is, if $M_c$ is the color matrix, a subscript of $s$ represents the scale term for a component, and a subscript of $b$ represents the bias term, then the components

$$
\begin{pmatrix}
R \\
G \\
B \\
A
\end{pmatrix}
$$

are transformed to

$$
\begin{pmatrix}
R' \\
G' \\
B' \\
A'
\end{pmatrix} =
\begin{pmatrix}
0 & 0 & 0 & 0 \\
0 & G_s & 0 & 0 \\
0 & 0 & B_s & 0 \\
0 & 0 & 0 & A_s
\end{pmatrix} M_c
\begin{pmatrix}
R \\
G \\
B \\
A
\end{pmatrix} +
\begin{pmatrix}
R_b \\
G_b \\
B_b \\
A_b
\end{pmatrix}.
$$

Post Color Matrix Color Table Lookup

This step applies only to RGBA component groups. Post color matrix color table lookup is enabled or disabled by calling `Enable` or `Disable`
with the symbolic constant POST_COLOR_MATRIX_COLOR_TABLE. The post color matrix table is defined by calling ColorTable with a target argument of POST_COLOR_MATRIX_COLOR_TABLE. In all other respects, operation is identical to color table lookup, as defined in section 3.7.5.

The required state is one bit indicating whether post color matrix lookup is enabled or disabled. In the initial state, lookup is disabled.

**Histogram**

This step applies only to RGBA component groups. Histogram operation is enabled or disabled by calling Enable or Disable with the symbolic constant HISTOGRAM.

If the width of the table is non-zero, then indices \( R_i, G_i, B_i, A_i \) are derived from the red, green, blue, and alpha components of each pixel group (without modifying these components) by clamping each component to \([0, 1]\), multiplying by one less than the width of the histogram table, and rounding to the nearest integer. If the format of the HISTOGRAM table includes red, luminance, the red or luminance component of histogram entry \( R_i \) is incremented by one. If the format of the HISTOGRAM table includes green, the green component of histogram entry \( G_i \) is incremented by one. The blue and alpha components of histogram entries \( B_i \) and \( A_i \) are incremented in the same way. If a histogram entry component is incremented beyond its maximum value, its value becomes undefined; this is not an error.

If the Histogram sink parameter is FALSE, histogram operation has no effect on the stream of pixel groups being processed. Otherwise, all RGBA pixel groups are discarded immediately after the histogram operation is completed. Because histogram precedes minmax, no minmax operation is performed. No pixel fragments are generated, no change is made to texture memory contents, and no pixel values are returned. However, texture object state is modified whether or not pixel groups are discarded.

**Minmax**

This step applies only to RGBA component groups. Minmax operation is enabled or disabled by calling Enable or Disable with the symbolic constant MINMAX.

If the format of the minmax table includes red or luminance, the red component value replaces the red or luminance value in the minimum table element if and only if it is less than that component. Likewise, if the format includes red or luminance and the red component of the group is greater than the red or luminance value in the maximum element, the red group component replaces the red or lumi-
nance maximum component. If the format of the table includes green, the green group component conditionally replaces the green minimum and/or maximum if it is smaller or larger, respectively. The blue and alpha group components are similarly tested and replaced, if the table format includes blue and/or alpha. The internal type of the minimum and maximum component values is floating point, with at least the same representable range as a floating point number used to represent colors (section 2.1.1). There are no semantics defined for the treatment of group component values that are outside the representable range.

If the Minmax sink parameter is FALSE, minmax operation has no effect on the stream of pixel groups being processed. Otherwise, all RGBA pixel groups are discarded immediately after the minmax operation is completed. No pixel fragments are generated, no change is made to texture memory contents, and no pixel values are returned. However, texture object state is modified whether or not pixel groups are discarded.

3.7.6 Pixel Rectangle Multisample Rasterization

If MULTISAMPLE is enabled, and the value of SAMPLE_BUFFERS is one, then pixel rectangles are rasterized using the following algorithm. Let \((X_{rp}, Y_{rp})\) be the current raster position. (If the current raster position is invalid, then DrawPixels is ignored.) If a particular group (index or components) is the \(n\)th in a row and belongs to the \(m\)th row, consider the region in window coordinates bounded by the rectangle with corners

\[
(X_{rp} + Z_x * n, Y_{rp} + Z_y * m)
\]

and

\[
(X_{rp} + Z_x * (n + 1), Y_{rp} + Z_y * (m + 1))
\]

where \(Z_x\) and \(Z_y\) are the pixel zoom factors specified by PixelZoom, and may each be either positive or negative. A fragment representing group \((n, m)\) is produced for each framebuffer pixel with one or more sample points that lie inside, or on the bottom or left boundary, of this rectangle. Each fragment so produced takes its associated data from the group and from the current raster position, in a manner consistent with the discussion in the Conversion to Fragments subsection of section 3.7.4. All depth and color sample values are assigned the same value, taken either from their group (for depth and color component groups) or from the current raster position (if they are not). All sample values are assigned the same fog coordinate and the same set of texture coordinates, taken from the current raster position.

Version 3.0 (with changes marked - August 11, 2008)
3.8 Bitmaps

Bitmaps are rectangles of zeros and ones specifying a particular pattern of fragments to be produced. Each of these fragments has the same associated data. These data are those associated with the current raster position.

Bitmaps are sent using

```c
void Bitmap(sizei w, sizei h, float xbo, float ybo,
            float xbi, float ybi, ubyte *data);
```

where 

- `w` and `h` comprise the integer width and height of the rectangular bitmap, respectively.
- `(xbo, ybo)` gives the floating-point `x` and `y` values of the bitmap’s origin.
- `(xbi, ybi)` gives the floating-point `x` and `y` increments that are added to the raster position after the bitmap is rasterized.
- `data` is a pointer to a bitmap.

A bitmap is unpacked from memory according to the procedure given in section 3.7.4 for `DrawPixels`; it is as if the `width` and `height` passed to that command were equal to `w` and `h`, respectively, the `type` were `BITMAP`, and the `format` were `COLOR_INDEX`. The unpacked values (before any conversion or arithmetic would have been performed) form a stipple pattern of zeros and ones.

See figure 3.9.

A bitmap sent using `Bitmap` is rasterized as follows. First, if the current raster position is invalid (the valid bit is reset), the bitmap is ignored. Otherwise, a rectangular array of fragments is constructed, with lower left corner at

\[(x_{ll}, y_{ll}) = (\lfloor x_{rp} - x_{bo} \rfloor, \lfloor y_{rp} - y_{bo} \rfloor)\]

and upper right corner at \((x_{ll} + w, y_{ll} + h)\) where `w` and `h` are the width and height of the bitmap, respectively. Fragments in the array are produced if the corresponding bit in the bitmap is 1 and not produced otherwise. The associated data for each fragment are those associated with the current raster position. Once the fragments have been produced, the current raster position is updated:

\[(x_{rp}, y_{rp}) \leftarrow (x_{rp} + x_{bi}, y_{rp} + y_{bi}).\]

The `z` and `w` values of the current raster position remain unchanged.

Calling `Bitmap` will result in an `INVALID_FRAMEBUFFER_OPERATION` error if the object bound to `DRAW_FRAMEBUFFER_BINDING` is not framebuffer complete (see section 4.4.4).

Version 3.0 (with changes marked - August 11, 2008)
3.8. BITMAPS

Figure 3.9. A bitmap and its associated parameters. $x_{bi}$ and $y_{bi}$ are not shown.

**Bitmap Multisample Rasterization**

If `MULTISAMPLE` is enabled, and the value of `SAMPLE_BUFFERS` is one, then bitmaps are rasterized using the following algorithm. If the current raster position is invalid, the bitmap is ignored. Otherwise, a screen-aligned array of pixel-size rectangles is constructed, with its lower left corner at $(X_{rp}, Y_{rp})$, and its upper right corner at $(X_{rp} + w, Y_{rp} + h)$, where $w$ and $h$ are the width and height of the bitmap. Rectangles in this array are eliminated if the corresponding bit in the bitmap is 0, and are retained otherwise. Bitmap rasterization produces a fragment for each framebuffer pixel with one or more sample points either inside or on the bottom or left edge of a retained rectangle.

Coverage bits that correspond to sample points either inside or on the bottom or left edge of a retained rectangle are 1, other coverage bits are 0. The associated data for each sample are those associated with the current raster position. Once the fragments have been produced, the current raster position is updated exactly as it is in the single-sample rasterization case.
3.9 Texturing

Texturing maps a portion of one or more specified images onto each primitive for which texturing is enabled. This mapping is accomplished by using the color of an image at the location indicated by a texture coordinate set’s \((s, t, r, q)\) coordinates.

The internal data type of a texture may be fixed-point, floating-point, signed integer or unsigned integer, depending on the internal format of the texture. The correspondence between the internal format and the internal data type is given in Tables 3.16-3.18. Fixed-point and floating-point textures return a floating-point value and integer textures return signed or unsigned integer values. When a fragment shader is active, the shader is responsible for interpreting the result of a texture lookup as the correct data type, otherwise the result is undefined. When not using a fragment shader, floating-point texture values are assumed, and the results of using integer textures in this case are undefined.

Six types of texture are supported; each is a collection of images built from one-, two-, or three-dimensional array 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. Finally, a cube map is a special two-dimensional array texture 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.

Implementations must support texturing using at least two images at a time. Each fragment or vertex carries multiple sets of texture coordinates \((s, t, r, q)\) which are used to index separate images to produce color values which are collectively used to modify the resulting transformed vertex or fragment color. Texturing is specified only for RGBA mode; its use in color index mode is undefined. The following subsections (up to and including section 3.9.7) specify the GL operation with a single texture and section 3.9.17 specifies the details of how multiple texture units interact.

The GL provides two ways to specify the details of how texturing of a primitive is effected. The first is referred to as fixed-function fragment shading, or simply fixed-function, and is described in this section. The second is referred to as a fragment shader, and is described in section 3.12. The specification of the image to be texture mapped and the means by which the image is filtered when applied to the primitive are common to both methods and are discussed in this section. The fixed-function method for determining what RGBA value is produced is also described in this section. If a fragment shader is active, the method for determining the RGBA value is specified by an application-supplied fragment shader as described in the OpenGL Shading Language Specification.

Version 3.0 (with changes marked - August 11, 2008)
3.9. TEXTURING

When no fragment shader is active, the coordinates used for texturing are 
\((s/q, t/q, r/q)\), derived from the original texture coordinates \((s, t, r, q)\). If the \(q\) 
texture coordinate is less than or equal to zero, the coordinates used for texturing 
are undefined. When a fragment shader is active, the \((s, t, r, q)\) coordinates are 
available to the fragment shader. The coordinates used for texturing in a fragment 
shader are defined by the OpenGL Shading Language Specification.

3.9.1 Texture Image Specification

The command

\[
\text{void TexImage3D}(\text{enum target}, \text{int level}, \text{int internalformat},
\text{sizei width, sizei height, sizei depth, int border},
\text{enum format, enum type, void *data});
\]

is used to specify a three-dimensional texture image. \text{target} must be one of 
\text{T\(EX\)URE\_3D} for a three-dimensional texture or \text{T\(EX\)URE\_2D\_ARRAY} for an 
two-dimensional array texture. Additionally, \text{target} may be either \text{PROXY\_T\(EX\)URE\_3D} 
for a three-dimensional proxy texture, or \text{PROXY\_T\(EX\)URE\_2D\_ARRAY} for a two-
dimensional proxy array texture, as discussed in section 3.9.11. \text{format, type, and} 
data match the corresponding arguments to \text{DrawPixels} (refer to section 3.7.4); 
they 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. The 
\text{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 \text{width} and \text{height} parameters to \text{T\(ex\)Image3D}. The values of \text{UNPACK\_ROW\_LENGTH} and \text{UNPACK\_ALIGNMENT} control the row-to-row spacing in these images in the same manner as \text{DrawPixels}. If the value of 
the integer parameter \text{UNPACK\_IMAGE\_HEIGHT} is not positive, then the number of 
rows in each two-dimensional image is \text{height}; otherwise the number of rows 
is \text{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 re-
lies on the integer parameter \text{UNPACK\_SKIP\_IMAGES}. If \text{UNPACK\_SKIP\_IMAGES} 
is positive, the pointer is advanced by \text{UNPACK\_SKIP\_IMAGES} times the number of 
elements in one two-dimensional image before obtaining the first group from mem-
ory. Then \text{depth} two-dimensional images are processed, each having a subimage 
extracted in the same manner as \text{DrawPixels}.

The selected groups are processed exactly as for \text{DrawPixels}, stopping just be-
fore final conversion. If the \text{internalformat} of the texture is signed or unsigned 
texture_integer
integer, the components are clamped to the representable range of the internal format. For signed formats, this is $[-2^{n-1}, 2^{n-1} - 1]$ where $n$ is the number of bits per component; for unsigned formats, the range is $[0, 2^n - 1]$. For color component groups, if the internalformat of the texture is fixed-point, the R, G, B, and A values are clamped to $[0, 1]$. For depth component groups, the depth value is clamped to $[0, 1]$. Otherwise, values are not modified. Stencil index values are framebuffer_object 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 3.15 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 3.15, as one of the sized internal format symbolic constants listed in tables 3.16-3.18, as one of the generic compressed internal format symbolic constants listed in table 3.19, or as one of the specific compressed internal format symbolic constants (if listed in table 3.19). internalformat may (for backwards compatibility with the 1.0 version of the GL) also take on the integer values 1, 2, 3, and 4, which are equivalent to symbolic constants LUMINANCE, LUMINANCE_ALPHA, RGB, and RGBA respectively. 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_CUBE_MAP, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_1D_ARRAY, PROXY_TEXTURE_2D_ARRAY, 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 tables 3.16-3.17 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 3.6; if the internal format is not integer and format is an integer format; or if format is an integer format and

Version 3.0 (with changes marked - August 11, 2008
3.9. TEXTURING

<table>
<thead>
<tr>
<th>Base Internal Format</th>
<th>RGBA, Depth, and Stencil Values</th>
<th>Internal Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA</td>
<td>A</td>
<td>A</td>
</tr>
<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>LUMINANCE</td>
<td>R</td>
<td>L</td>
</tr>
<tr>
<td>LUMINANCE_ALPHA</td>
<td>R,A</td>
<td>L,A</td>
</tr>
<tr>
<td>INTENSITY</td>
<td>R</td>
<td>I</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 3.15: Conversion from RGBA, depth, and stencil pixel components to internal texture, table, or filter components. See section 3.9.13 for a description of the texture components R, G, B, A, L, I, D, and S.

type is FLOAT.

The GL provides no specific compressed internal formats but does provide a mechanism to obtain token values for 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 or borders), `internalformat` is replaced by the corresponding base internal format and the texture image will not be compressed by the GL.
3.9. TEXTURING

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 3.15; 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 3.16-3.18 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 3.16-3.18 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 internal formats. Requesting one of these internal formats for any texture type will allocate exactly the internal component sizes and types shown for that format in tables 3.16-3.17:

- **Color formats:**
  - 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, RGB8I, and RG8UI.
  - R32F, R32I, R32UI, R16F, R16I, R16UI, R16, R8, R8I, and R8UI.
  - ALPHA8.

- **Color formats (texture-only):**
  - RGB32F, RGB32I, and RGB32UI.
  - RGB16F, RGB16I, RGB16UI, and RGB16.
  - RGB8, RGB8I, RGB8UI, and SRGB8.

Version 3.0 (with changes marked - August 11, 2008)
3.9. TEXTURING

- RGB9_E5.
- COMPRESSED_RG_RGTC2 and COMPRESSED_SIGNED_RG_RGTC2.
- 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:

\[
\begin{align*}
red_c &= \max(0, \min(shared_{\text{exp max}}, red)) \\
green_c &= \max(0, \min(shared_{\text{exp max}}, green)) \\
blue_c &= \max(0, \min(shared_{\text{exp max}}, blue))
\end{align*}
\]

where

\[
shared_{\text{exp max}} = \frac{(2^N - 1)}{2^N} 2^{E_{\text{max}} - B}.
\]

N is the number of mantissa bits per component (9), B is the exponent bias (15), and \( E_{\text{max}} \) 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:

\[
exp_p = \max(-B - 1, \lceil \log_2(max_c) \rceil) + 1 + B
\]

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

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

Version 3.0 (with changes marked - August 11, 2008)
3.9. TEXTURING

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

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

\[
\begin{align*}
\text{red}_s &= \lfloor \frac{red_c}{2^{exp_s-B-N}} + 0.5 \rfloor \\
\text{green}_s &= \lfloor \frac{green_c}{2^{exp_s-B-N}} + 0.5 \rfloor \\
\text{blue}_s &= \lfloor \frac{blue_c}{2^{exp_s-B-N}} + 0.5 \rfloor
\end{align*}
\]

The resulting \(\text{red}_s\), \(\text{green}_s\), \(\text{blue}_s\), 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 format \texttt{RGB} is allowed to store the components “as is” if the implementation can determine the current pixel transfer state acts as an identity transform on the components.

<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>ALPHA4</td>
<td>ALPHA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>ALPHA8</td>
<td>ALPHA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>ALPHA12</td>
<td>ALPHA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>ALPHA16</td>
<td>ALPHA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>R8</td>
<td>RED</td>
<td>8</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>RGB</td>
<td>RG</td>
<td>8</td>
<td>8</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>R3_G3_B2</td>
<td>RGB</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB4</td>
<td>RGB</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB5</td>
<td>RGB</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB8</td>
<td>RGB</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB10</td>
<td>RGB</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB12</td>
<td>RGB</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB16</td>
<td>RGB</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sized internal color formats continued on next page
### 3.9. TEXTURING

<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>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>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>SRGB8</td>
<td>RGB</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRGB8_ALPHA8</td>
<td>RGBA</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>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>
<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>i8</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>i16</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>i32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB8I</td>
<td>RG</td>
<td>i8</td>
<td>i8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB8UI</td>
<td>RG</td>
<td>i8</td>
<td>i8</td>
<td>i8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG16I</td>
<td>RG</td>
<td>i16</td>
<td>i16</td>
<td>i16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RG16UI</td>
<td>RG</td>
<td>i16</td>
<td>i16</td>
<td>i16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB32I</td>
<td>RG</td>
<td>i32</td>
<td>i32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB32UI</td>
<td>RG</td>
<td>i32</td>
<td>i32</td>
<td>i32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB8I</td>
<td>RG</td>
<td>i8</td>
<td>i8</td>
<td>i8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB8UI</td>
<td>RGB</td>
<td>i8</td>
<td>i8</td>
<td>i8</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>
</tbody>
</table>

Version 3.0 (with changes marked - August 11, 2008)
3.9. TEXTURING

Sized internal color formats continued from previous page

<table>
<thead>
<tr>
<th>Sized Internal Format</th>
<th>Base Internal Format</th>
<th>R bits</th>
<th>G bits</th>
<th>B bits</th>
<th>A bits</th>
<th>Shared bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB16UI</td>
<td>RGB</td>
<td>ui16</td>
<td>ui16</td>
<td>ui16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB32I</td>
<td>RGB</td>
<td>i32</td>
<td>i32</td>
<td>i32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGB32UI</td>
<td>RGB</td>
<td>ui32</td>
<td>ui32</td>
<td>ui32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGBA8I</td>
<td>RGBA</td>
<td>i8</td>
<td>i8</td>
<td>i8</td>
<td>i8</td>
<td></td>
</tr>
<tr>
<td>RGBA8UI</td>
<td>RGBA</td>
<td>ui8</td>
<td>ui8</td>
<td>ui8</td>
<td>ui8</td>
<td></td>
</tr>
<tr>
<td>RGBA16I</td>
<td>RGBA</td>
<td>i16</td>
<td>i16</td>
<td>i16</td>
<td>i16</td>
<td></td>
</tr>
<tr>
<td>RGBA16UI</td>
<td>RGBA</td>
<td>ui16</td>
<td>ui16</td>
<td>ui16</td>
<td>ui16</td>
<td></td>
</tr>
<tr>
<td>RGBA32I</td>
<td>RGBA</td>
<td>i32</td>
<td>i32</td>
<td>i32</td>
<td>i32</td>
<td></td>
</tr>
<tr>
<td>RGBA32UI</td>
<td>RGBA</td>
<td>ui32</td>
<td>ui32</td>
<td>ui32</td>
<td>ui32</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.16: 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, and no prefix is fixed-point.

<table>
<thead>
<tr>
<th>Sized Internal Format</th>
<th>Base Internal Format</th>
<th>A bits</th>
<th>L bits</th>
<th>I bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUMINANCE4</td>
<td>LUMINANCE</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUMINANCE8</td>
<td>LUMINANCE</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUMINANCE12</td>
<td>LUMINANCE</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUMINANCE16</td>
<td>LUMINANCE</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUMINANCE4_ALPHA4</td>
<td>LUMINANCE_ALPHA</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>LUMINANCE6_ALPHA2</td>
<td>LUMINANCE_ALPHA</td>
<td>2</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>LUMINANCE8_ALPHA8</td>
<td>LUMINANCE_ALPHA</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>LUMINANCE12_ALPHA4</td>
<td>LUMINANCE_ALPHA</td>
<td>4</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>LUMINANCE12_ALPHA12</td>
<td>LUMINANCE_ALPHA</td>
<td>12</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>LUMINANCE16_ALPHA16</td>
<td>LUMINANCE_ALPHA</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>INTENSITY4</td>
<td>INTENSITY</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTENSITY8</td>
<td>INTENSITY</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTENSITY12</td>
<td>INTENSITY</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTENSITY16</td>
<td>INTENSITY</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sized internal luminance formats continued on next page

Version 3.0 (with changes marked - August 11, 2008)
3.9. **TEXTURING**

<table>
<thead>
<tr>
<th>Sized Internal Format</th>
<th>Base Internal Format</th>
<th>D bits</th>
<th>S bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPTH_COMPONENT16</td>
<td>DEPTH_COMPONENT</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>DEPTH_COMPONENT24</td>
<td>DEPTH_COMPONENT</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>DEPTH_COMPONENT32</td>
<td>DEPTH_COMPONENT</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>DEPTH_COMPONENT32F</td>
<td>DEPTH_COMPONENT</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>DEPTH24_STENCIL8</td>
<td>DEPTH_STENCIL</td>
<td>24</td>
<td>8</td>
</tr>
<tr>
<td>DEPTH32F_STENCIL8</td>
<td>DEPTH_STENCIL</td>
<td>32</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 3.18: Correspondence of sized internal depth and stencil formats to base internal formats, internal data type, and desired component resolutions for each sized internal format. The component resolution prefix indicates the internal data type: \( f \) is floating point, \( i \) is signed integer, \( ui \) is unsigned integer, and no prefix is fixed-point.

<table>
<thead>
<tr>
<th>Sized Internal Format</th>
<th>Base Internal Format</th>
<th>A bits</th>
<th>L bits</th>
<th>I bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLUMINANCE</td>
<td>LUMINANCE</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLUMINANCE_ALPHA8</td>
<td>LUMINANCE_ALPHA</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.17: Correspondence of sized internal luminance and intensity 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 3.15. The specified image is compressed using a (possibly lossy) compression algorithm chosen by the GL.

A GL implementation may vary its allocation of internal component resolution or compressed internal format based on any `TexImage3D`, `TexImage2D` (see below), or `TexImage1D` (see below) parameter (except `target`), but the allocation and chosen compressed image format must not be a function of any other state and cannot be changed once they are established. In addition, the choice of a compressed

Version 3.0 (with changes marked - August 11, 2008)
### Table 3.19: Generic and specific compressed internal formats.

<table>
<thead>
<tr>
<th>Compressed Internal Format</th>
<th>Base Internal Format</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPRESSED_ALPHA</td>
<td>ALPHA</td>
<td>Generic</td>
</tr>
<tr>
<td>COMPRESSED_LUMINANCE</td>
<td>LUMINANCE</td>
<td>Generic</td>
</tr>
<tr>
<td>COMPRESSED_LUMINANCE_ALPHA</td>
<td>LUMINANCE_ALPHA</td>
<td>Generic</td>
</tr>
<tr>
<td>COMPRESSED_INTENSITY</td>
<td>INTENSITY</td>
<td>Generic</td>
</tr>
<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_SLUMINANCE</td>
<td>LUMINANCE</td>
<td>Generic</td>
</tr>
<tr>
<td>COMPRESSED_SLUMINANCE_ALPHA</td>
<td>LUMINANCE_ALPHA</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>

The specific \*RGTC\* formats are described in appendix C.1.
image format may not be affected by the data parameter. Allocations must be invariant; the same allocation and compressed image format must be chosen each time a texture image is specified with the same parameter values. These allocation rules also apply to proxy textures, which are described in section 3.9.11.

The image itself (referred to by data) is a sequence of groups of values. The first group is the lower left back corner of the texture image. Subsequent groups fill out rows of width width from left to right; height rows are stacked from bottom to top forming a single two-dimensional image slice; and depth slices are stacked from back to front. When the final R, G, B, and A components have been computed for a group, they are assigned to components of a texel as described by table 3.15. Counting from zero, each resulting Nth texel is assigned internal integer coordinates \((i,j,k)\), where

\[
i = (N \mod width) - w_b
\]

\[
j = (\lfloor N/width \rfloor \mod height) - h_b
\]

\[
k = (\lfloor N/width \times height \rfloor \mod depth) - d_b
\]

and \(w_b, h_b,\) and \(d_b\) are the specified border width, height, and depth. \(w_b\) and \(h_b\) are the specified border value; \(d_b\) is the specified border value if target is TEXTURE_3D, or zero if target is TEXTURE_2D_ARRAY. Thus the last two-dimensional image slice of the three-dimensional image is indexed with the highest value of \(k\).

Each color component is converted (by rounding to nearest) to a fixed-point value with \(n\) bits, where \(n\) is the number of bits of storage allocated to that component in the image array. We assume that the fixed-point representation used represents each value \(k/(2^n - 1)\), where \(k \in \{0, 1, \ldots, 2^n - 1\}\), as \(k\) (e.g. 1.0 is represented in binary as a string of all ones).

The level argument to TexImage3D is an integer level-of-detail number. Levels of detail are discussed below, under Mipmapping. The main texture image has a level of detail number of 0. If a level-of-detail less than zero is specified, the error INVALID_VALUE is generated.

The border argument to TexImage3D is a border width. The significance of borders is described below. The border width affects the dimensions of the texture image: let

\[
w_s = w_t + 2w_b
\]

\[
h_s = h_t + 2h_b
\]

\[
d_s = d_t + 2d_b
\]

(3.15)
3.9. TEXTURING

where \( w_s, h_s, \) and \( d_s \) are the specified image width, depth, and depth, and \( w_t, h_t, \) or \( d_t \) 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 \( \text{INVALID\_VALUE} \) is generated.

An image with zero width, height, or depth indicates the null texture. If the null texture is specified for the level-of-detail specified by texture parameter \( \text{TEXTURE\_BASE\_LEVEL} \) (see section 3.9.4), it is as if texturing were disabled.

Currently, the maximum border width \( b_t \) is 1. If \( \text{border} \) is less than zero, or greater than \( b_t \) then the error \( \text{INVALID\_VALUE} \) is generated.

The maximum allowable width, height, or depth of a texture 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 \( 2^{k-lod} + 2b_t \) for image arrays of level-of-detail 0 through \( k \), where \( k \) is the log base 2 of \( \text{MAX\_3D\_TEXTURE\_SIZE} \), \( lod \) is the level-of-detail of the image array, and \( b_t \) is the maximum border width. It may be zero for image arrays of any level-of-detail greater than \( k \). The error \( \text{INVALID\_VALUE} \) 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 \( \text{INVALID\_OPERATION} \) error results.

In a similar fashion, the maximum allowable width of a texture array for a one- or two-dimensional, or one- or two-dimensional array texture, and the maximum allowable height of a two-dimensional or two-dimensional array texture, must be at least \( 2^{k-lod} + 2b_t \) for image arrays of level 0 through \( k \), where \( k \) is the log base 2 of \( \text{MAX\_TEXTURE\_SIZE} \). The maximum allowable width and height of a cube map texture must be the same, and must be at least \( 2^{k-lod} + 2b_t \) for image arrays level 0 through \( k \), where \( k \) is the log base 2 of \( \text{MAX\_CUBE\_MAP\_TEXTURE\_SIZE} \). The maximum number of layers for one- and two-dimensional array textures (height or depth, respectively) must be at least \( \text{MAX\_ARRAY\_TEXTURE\_LAYER} \) for all levels.

An implementation may allow an image array of level 0 to be created only if that single image array can be supported. Additional constraints on the creation of image arrays of level 1 or greater are described in more detail in section 3.9.10.

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. \( \text{target} \) must be one of \( \text{TEXTURE\_2D} \) for a two-dimensional texture, \( \text{TEXTURE\_1D\_ARRAY} \) for a one-dimensional array texture, or one of \( \text{TEXTURE\_CUBE\_MAP\_POSITIVE\_X} \),

Version 3.0 (with changes marked - August 11, 2008
3.9. TEXTURING

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, or PROXY_TEXTURE_CUBE_MAP for a cube map proxy texture in the special case discussed in section 3.9.11. 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

- The border depth, $d_b$, is zero, and the depth of the image is always 1 regardless of the value of border.
- The border height, $h_b$, is zero if target is TEXTURE_1D_ARRAY, and border otherwise.
- Convolution will be performed on the image (possibly changing its width and height) if SEPARABLE_2D or CONVOLUTION_2D is enabled.
- UNPACK_SKIP_IMAGES is ignored.

A two-dimensional 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 enabling cube map texturing or 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.

Finally, the command

```c
void TexImage1D( enum target, int level,
    int internalformat,sizei width, int border,
    enum format, enum type, void *data );
```

Version 3.0 (with changes marked - August 11, 2008)
is used to specify a one-dimensional texture image. `target` must be either `TEXTURE_1D` or `PROXY_TEXTURE_1D` in the special case discussed in section 3.9.11.)

For the purposes of decoding the texture image, `TexImage1D` is equivalent to calling `TexImage2D` with corresponding arguments and `height` of 1, except that

- The border height and depth (\(h_b\) and \(d_b\)) are always zero, regardless of the value of `border`.
- Convolution will be performed on the image (possibly changing its `width`) only if `CONVOLUTION_1D` is enabled.

The image indicated to the GL by the image pointer is decoded and copied into the GL’s internal memory. This copying effectively places the decoded image inside a border of the maximum allowable width \(b_t\) whether or not a border has been specified (see figure 3.10)\(^1\). If no border or a border smaller than the maximum allowable width has been specified, then the image is still stored as if it were surrounded by a border of the maximum possible width. Any excess border (which surrounds the specified image, including any border) is assigned unspecified values. A two-dimensional texture has a border only at its left, right, top, and bottom ends, and a one-dimensional texture has a border only at its left and right ends.

We shall refer to the (possibly border augmented) 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 3.15. A two-dimensional texel array has depth \(d_s = 1\), with height \(h_s\) and width \(w_s\) as above, and 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 that fragment’s associated \((s, t, r)\) coordinates, but may not correspond to any actual texel. See figure 3.10.

If the `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 `target`, `level`, `internalformat`, `border`, `width`, `height`, and `depth`, but with unspecified image contents. In this case no pixel values are accessed in client memory, and no pixel processing is performed. Errors are generated, however, exactly as though the `data` pointer were valid. Otherwise if the pixel unpack buffer object is non-zero, the `data` argument is treatedly normally to refer to the beginning of the pixel unpack buffer object’s data.

\(^1\) Figure 3.10 needs to show a three-dimensional texture image.

Version 3.0 (with changes marked - August 11, 2008)
Figure 3.10. 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.
3.9. TEXTURING

3.9.2 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);
```

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_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, or TEXTURE_CUBE_MAP_NEGATIVE_Z. \( x, y, width, \) and \( height \) correspond precisely to the corresponding arguments to CopyPixels (refer to section 4.3.3); they specify the image’s \( width \) and \( height \), and the lower left \( (x, y) \) coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to CopyPixels with argument type set to COLOR, DEPTH, or DEPTH_STENCIL, depending on \( internalformat \), stopping after pixel transfer processing is complete. 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, except that \( internalformat \) may not be specified as 1, 2, 3, or 4. 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.

Version 3.0 (with changes marked - August 11, 2008
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`, except that `internalformat` may not be specified as 1, 2, 3, or 4. 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,
                    enum format, enum type, void *data );
void TexSubImage2D( enum target, int level,
                    int xoffset, int yoffset,
                    sizei width, sizei height,
                    enum format, enum type, void *data );
void TexSubImage1D( enum target, int level,
                    int xoffset,
                    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` texture_array

Version 3.0 (with changes marked - August 11, 2008)
and \texttt{CopyTexSubImage2D} must be one of \texttt{TEXTURE\_2D}, \texttt{TEXTURE\_1D\_ARRAY}, \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}, or \texttt{TEXTURE\_CUBE\_MAP\_NEGATIVE\_Z}, and the \texttt{target} arguments of \texttt{TexSubImage3D} and \texttt{CopyTexSubImage3D} must be \texttt{TEXTURE\_3D} or \texttt{TEXTURE\_2D\_ARRAY}. The \texttt{level} parameter of each command specifies the level of the texel array that is modified. If \texttt{level} is less than zero or greater than the base 2 logarithm of the maximum texture width, height, or depth, the error \texttt{INVALID\_VALUE} is generated.

\texttt{TexSubImage3D} arguments \texttt{width}, \texttt{height}, \texttt{depth}, \texttt{format}, \texttt{type}, and \texttt{data} match the corresponding arguments to \texttt{TexImage3D}, meaning that they are specified using the same values, and have the same meanings. Likewise, \texttt{TexSubImage2D} arguments \texttt{width}, \texttt{height}, \texttt{format}, \texttt{type}, and \texttt{data} match the corresponding arguments to \texttt{TexImage2D}, and \texttt{TexSubImage1D} arguments \texttt{width}, \texttt{format}, \texttt{type}, and \texttt{data} match the corresponding arguments to \texttt{TexImage1D}.

\texttt{CopyTexSubImage3D} and \texttt{CopyTexSubImage2D} arguments \texttt{x}, \texttt{y}, \texttt{width}, and \texttt{height} match the corresponding arguments to \texttt{CopyTexImage2D}. \texttt{CopyTexSubImage1D} arguments \texttt{x}, \texttt{y}, and \texttt{width} match the corresponding arguments to \texttt{CopyTexImage1D}. Each of the \texttt{TexSubImage} commands interprets and processes pixel groups in exactly the manner of its \texttt{TexImage} counterpart, except that the assignment of R, G, B, A, depth, and stencil index pixel group values to the texture framebuffer buffer components is controlled by the \texttt{internalformat} of the texel array, not by an argument to the command. The same constraints and errors apply to the \texttt{TexSubImage} commands’ \texttt{format} and \texttt{internalformat} of the texel array being re-specified as apply to the \texttt{format} and \texttt{internalformat} arguments of its \texttt{TexImage} counterparts.

Arguments \texttt{xoffset}, \texttt{yoffset}, and \texttt{zoffset} of \texttt{TexSubImage3D} and \texttt{CopyTexSubImage3D} specify the lower left texel coordinates of a \texttt{width}-wide by \texttt{height}-high by \texttt{depth}-deep rectangular subregion of the texel array. The \texttt{depth} argument associated with \texttt{CopyTexSubImage3D} is always 1, because framebuffer memory is two-dimensional - only a portion of a single \texttt{s}, \texttt{t} slice of a three-dimensional texture is replaced by \texttt{CopyTexSubImage3D}.

Negative values of \texttt{xoffset}, \texttt{yoffset}, and \texttt{zoffset} correspond to the coordinates of border texels, addressed as in figure 3.10. Taking \texttt{w_x}, \texttt{h_x}, \texttt{d_x}, \texttt{w_y}, \texttt{h_y}, and \texttt{d_y} to be the specified \texttt{width}, \texttt{height}, \texttt{depth}, and border \texttt{width}, \texttt{border height}, and \texttt{border depth} of the texel array, and taking \texttt{x}, \texttt{y}, \texttt{z}, \texttt{w}, \texttt{h}, and \texttt{d} to be the \texttt{xoffset}, \texttt{yoffset}, \texttt{zoffset}, \texttt{width}, \texttt{height}, and \texttt{depth} argument values, any of the following relationships

\texttt{2} Because the framebuffer is inherently two-dimensional, there is no \texttt{CopyTexImage3D} command.
3.9. TEXTURING

generates the error INVALID_VALUE:

\[ x < -w_b \]
\[ x + w > w_s - w_b \]
\[ y < -h_b \]
\[ y + h > h_s - h_b \]
\[ z < -d_b \]
\[ z + d > d_s - d_b \]

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

\[ i = x + (n \mod w) \]
\[ j = y + (\lfloor \frac{n}{w} \rfloor \mod h) \]
\[ k = z + (\lfloor \frac{n \times width \times height}{d} \rfloor \mod d) \]

Arguments \texttt{xoffset} and \texttt{yoffset} of \texttt{TexSubImage2D} and \texttt{CopyTexSubImage2D} specify the lower left texel coordinates of a width-wide by height-high rectangular subregion of the \texttt{texel array}. Negative values of \texttt{xoffset} and \texttt{yoffset} correspond to the coordinates of border texels, addressed as in figure 3.10. Taking \( w_s, h_s, \) and \( b_s \) to be the specified width, height, and border width of the \texttt{texel array}, and taking \( x, y, w, \) and \( h \) to be the \texttt{xoffset}, \texttt{yoffset}, \texttt{width}, and \texttt{height} argument values, any of the following relationships generates the error INVALID_VALUE:

\[ x < -b_s \]
\[ x + w > w_s - b_s \]
\[ y < -b_s \]
\[ y + h > h_s - b_s \]

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

\[ i = x + (n \mod w) \]
\[ j = y + (\lfloor \frac{n}{w} \rfloor \mod h) \]

Version 3.0 (with changes marked - August 11, 2008)
The \textit{xoffset} argument of \texttt{TexSubImage1D} and \texttt{CopyTexSubImage1D} specifies the left texel coordinate of a \textit{width}-wide subregion of the texel array. Negative values of \textit{xoffset} correspond to the coordinates of border texels. Taking \(w_s\) and \(b_s\) to be the specified width and border width of the texel array, and \(x\) and \(w\) to be the \textit{xoffset} and \textit{width} argument values, either of the following relationships generates the error \texttt{INVALID\_VALUE}:

\[
\begin{align*}
&x < -b_s \\
&x + w > w_s - b_s
\end{align*}
\]

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

\[i = x + (n \mod w)\]

Texture images with compressed internal formats may be stored in such a way that it is not possible to modify an image with subimage commands without having to decompress and recompress the texture image. Even if the image were modified in this manner, it may not be possible to preserve the contents of some of the texels outside the region being modified. To avoid these complications, the GL does not support arbitrary modifications to texture images with compressed internal formats. Calling \texttt{TexSubImage3D}, \texttt{CopyTexSubImage3D}, \texttt{TexSubImage2D}, \texttt{CopyTexSubImage2D}, \texttt{TexSubImage1D}, or \texttt{CopyTexSubImage1D} will result in an \texttt{INVALID\_OPERATION} error if \textit{xoffset}, \textit{yoffset}, or \textit{zoffset} is not equal to \(-b_s\) (border width). In addition, the contents of any texel outside the region modified by such a call are undefined. These restrictions may be relaxed for specific compressed internal formats whose images are easily modified.

If the internal format of the texture image being modified is one of the specific \texttt{RGTC} formats described in table 3.19, the texture is stored using one of the \texttt{RGTC} texture image encodings (see appendix C.1). Since \texttt{RGTC} images are easily edited along \(4 \times 4\) texel boundaries, the limitations on subimage location and size are relaxed for \texttt{TexSubImage2D}, \texttt{TexSubImage3D}, \texttt{CopyTexSubImage2D}, and \texttt{CopyTexSubImage3D}. These commands will generate an \texttt{INVALID\_OPERATION} error if one of the following conditions occurs:

- \textit{width} is not a multiple of four or equal to \texttt{TEXTURE\_WIDTH}, unless \textit{xoffset} and \textit{yoffset} are both zero.

- \textit{height} is not a multiple of four or equal to \texttt{TEXTURE\_HEIGHT}, unless \textit{xoffset} and \textit{yoffset} are both zero.
3.9. TEXTURING

- $xoffset$ or $yoffset$ is not a multiple of four.

The contents of any $4 \times 4$ 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. Calling `CopyTexSubImage3D`, `CopyTexImage2D`, `CopyTexSubImage2D`, `CopyTexImage1D`, or `CopyTexSubImage1D` will result in an `INVALID_FRAMEBUFFER_OPERATION` error if the object bound to `READ_FRAMEBUFFER_BINDING` is not framebuffer complete (see section 4.4.4).

3.9.3 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`. `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.

`internalformat` must be a supported specific compressed internal format. An `INVALID_ENUM` error will be generated if any other values, including any of the six generic compressed internal formats, is specified.

Version 3.0 (with changes marked - August 11, 2008)
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 and pixel transfer 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 or formats described in table 3.19, 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

Version 3.0 (with changes marked - August 11, 2008
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**.

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 **TexSubImage1D**, **TexSubImage2D**, and **TexSubImage3D**. **data** points to compressed image data stored in the compressed image format corresponding to **format**. Since the core GL provides no specific image formats, using any of these six generic compressed internal formats as **format** will result in an **INVALID_ENUM** error.

The image pointed to by **data** and the **imageSize** parameter are interpreted as though they were provided to **CompressedTexImage1D**, **CompressedTexImage2D**, and **CompressedTexImage3D**. These commands do not provide for image format conversion, so an **INVALID_OPERATION** error results if **format** does not match the internal format of the texture image being modified. If the **imageSize** parameter is not consistent with the format, dimensions, and contents of the compressed image (too little or too much data), an **INVALID_VALUE** error results.

As with **CompressedTexImage** calls, compressed internal formats may have additional restrictions on the use of the compressed image specification calls or parameters. Any such restrictions will be documented in the specification defining the compressed internal format; violating these restrictions will result in an **INVALID_OPERATION** error.

Any restrictions imposed by specific compressed internal formats will be invariant, meaning that if the GL accepts and stores a texture image in compressed form, providing the same image to **CompressedTexSubImage1D**, **CompressedTexSubImage2D**, **CompressedTexSubImage3D** will not result in an **INVALID_OPERATION** error if the following restrictions are satisfied:

Version 3.0 (with changes marked - August 11, 2008)
• *data* points to a compressed texture image returned by *GetCompressedTexImage* (section 6.1.4).

• *target*, *level*, and *format* match the *target*, *level* and *format* parameters provided to the *GetCompressedTexImage* call returning *data*.

• *width*, *height*, *depth*, *format*, and *imageSize* match the values of *TEXTURE_WIDTH*, *TEXTURE_HEIGHT*, *TEXTURE_DEPTH*, *TEXTURE_INTERNAL_FORMAT*, and *TEXTURE_COMPRESSED_IMAGE_SIZES* for image level *level* in effect at the time of the *GetCompressedTexImage* call returning *data*.

• *width*, *height*, *depth*, *format* match the values of *TEXTURE_WIDTH*, *TEXTURE_HEIGHT*, *TEXTURE_DEPTH*, and *TEXTURE_INTERNAL_FORMAT* currently in effect for image level *level*.

• *xoffset*, *yoffset*, and *zoffset* are all $-b$, where $b$ is the value of *TEXTURE_BORDER* currently in effect for image level *level*.

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

Calling *CompressedTexSubImage3D*, *CompressedTexSubImage2D*, or *CompressedTexSubImage1D* will result in an *INVALID_OPERATION* error if *xoffset*, *yoffset*, or *zoffset* is not equal to $-b$ (border width), or if *width*, *height*, and *depth* do not match the values of *TEXTURE_WIDTH*, *TEXTURE_HEIGHT*, or *TEXTURE_DEPTH*, respectively. The contents of any texel outside the region modified by the call are undefined. These restrictions may be relaxed for specific compressed internal formats whose images are easily modified.

If *internalformat* is one of the specific RGTC or formats described in table 3.19, the texture is stored using one of the RGTC compressed texture image encodings (see appendix C.1). If *internalformat* is an RGTC format, *CompressedTexSubImage1D* will generate an *INVALID_ENUM* error; *CompressedTexSubImage2D* will generate an *INVALID_OPERATION* error if *border* is non-zero; and *CompressedTexSubImage3D* will generate an *INVALID_OPERATION* error if *border* is non-zero or *target* is not *TEXTURE_2D_ARRAY*. Since RGTC images are easily edited along $4 \times 4$ texel boundaries, the limitations on subimage location and size are relaxed for *CompressedTexSubImage2D* and *CompressedTexSubImage3D*. These commands will result in an *INVALID_OPERATION* error if one of the following conditions occurs:

• *width* is not a multiple of four or equal to *TEXTURE_WIDTH*.
3.9. TEXTURING

- **height** is not a multiple of four or equal to **TEXTURE_HEIGHT**.
- **xoffset** or **yoffset** is not a multiple of four.

The contents of any $4 \times 4$ block of texels of an RGTC compressed texture image that does not intersect the area being modified are preserved during valid **TexSubImage** and **CopyTexSubImage** calls.

3.9.4 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{u}iv( enum target, enum pname,
T *params );
```

**target** is the target, either **TEXTURE_1D**, **TEXTURE_2D**, **TEXTURE_3D**, **TEXTURE_1D_ARRAY**, **TEXTURE_2D_ARRAY**, or **TEXTURE_CUBE_MAP**. **pname** is a symbolic constant indicating the parameter to be set; the possible constants and corresponding parameters are summarized in table 3.20. 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 value for **TEXTURE_PRIORITY** is specified as an integer, the conversion for signed integers from table 2.10 is applied to convert this value to floating-point, followed by clamping the value to lie in $[0, 1]$.

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**, the conversion for signed integers from table 2.10 is applied to convert these values to floating-point. Otherwise the values are unmodified and stored as floating-point.

In the remainder of section 3.9, denote by $lod_{\text{min}}$, $lod_{\text{max}}$, $level_{\text{base}}$, and $level_{\text{max}}$ 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.

Version 3.0 (with changes marked - August 11, 2008)
### 3.9. 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, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER, MIRRORED_REPEAT</td>
</tr>
<tr>
<td>TEXTURE_WRAP_T</td>
<td>enum</td>
<td>CLAMP, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER, MIRRORED_REPEAT</td>
</tr>
<tr>
<td>TEXTURE_WRAP_R</td>
<td>enum</td>
<td>CLAMP, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER, MIRRORED_REPEAT</td>
</tr>
<tr>
<td>TEXTURE_WRAP_R</td>
<td>enum</td>
<td>CLAMP, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER, MIRRORED_REPEAT</td>
</tr>
<tr>
<td>TEXTURE_WRAP_T</td>
<td>enum</td>
<td>CLAMP, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER, MIRRORED_REPEAT</td>
</tr>
<tr>
<td>TEXTURE_WRAP_R</td>
<td>enum</td>
<td>CLAMP, 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_PRIORITY</td>
<td>float</td>
<td>any value in [0, 1]</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>DEPTH_TEXTURE_MODE</td>
<td>enum</td>
<td>RED, LUMINANCE, INTENSITY, ALPHA</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>
<tr>
<td>GENERATE_MIPMAP</td>
<td>boolean</td>
<td>TRUE or FALSE</td>
</tr>
</tbody>
</table>

Table 3.20: Texture parameters and their values. Version 3.0 (with changes marked - August 11, 2008)
Table 3.21: Selection of cube map images based on major axis direction of texture coordinates.

If the value of texture parameter `GENERATE_MIPMAP` is `TRUE`, specifying or changing texel arrays may have side effects, which are discussed in the `Automatic Mipmap Generation` discussion of section 3.9.7.

### 3.9.5 Depth Component Textures

Depth textures and the depth components of depth/stencil textures can be treated as `RED`, `LUMINANCE`, `INTENSITY` or `ALPHA` textures during texture filtering and application (see section 3.9.14). The initial state for depth and depth/stencil textures treats them as `LUMINANCE` textures except in a forward-compatible context, where the initial state instead treats them as `RED` textures.

### 3.9.6 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 3.21 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 3.21, an updated \((s \ t)\) is calculated as follows:
3.9. TEXTURING

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

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

This new \((s, t)\) is used to find a texture value in the determined face’s two-dimensional texture image using the rules given in sections 3.9.7 through 3.9.8.

3.9.7 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

\[ \lambda_{\text{base}}(x, y) = \log_2[\rho(x, y)] \]  

(3.16)

\[ \lambda'(x, y) = \lambda_{\text{base}}(x, y) + \text{clamp}(\text{bias}_{\text{texobj}} + \text{bias}_{\text{texunit}} + \text{bias}_{\text{shader}}) \]  

(3.17)

\[
\lambda = \begin{cases} 
\text{lod}_{\text{max}}, & \lambda' > \text{lod}_{\text{max}} \\
\lambda', & \text{lod}_{\text{min}} \leq \lambda' \leq \text{lod}_{\text{max}} \\
\text{lod}_{\text{min}}, & \lambda' < \text{lod}_{\text{min}} \\
\text{undefined}, & \text{lod}_{\text{min}} > \text{lod}_{\text{max}} 
\end{cases}
\]

(3.18)

\(\text{bias}_{\text{texobj}}\) is the value of \(\text{TEXTURE\_LOD\_BIAS}\) for the bound texture object (as described in section 3.9.4). \(\text{bias}_{\text{texunit}}\) is the value of \(\text{TEXTURE\_LOD\_BIAS}\) for the current texture unit (as described in section 3.9.13). \(\text{bias}_{\text{shader}}\) is the value of the optional bias parameter in the texture lookup functions available to fragment shaders. If the texture access is performed in a fragment shader without a provided bias value, the bias from the current texture unit is used.

Version 3.0 (with changes marked - August 11, 2008)
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 \texttt{MAX\_TEXTURE\_LOD\_BIAS}.

If $\lambda(x, y)$ is less than or equal to the constant $c$ (see section 3.9.8) the texture is said to be magnified; if it is greater, the texture is minified. Sampling of minified textures is described in the remainder of this section, while sampling of magnified textures is described in section 3.9.8.

The initial values of $lod_{\text{min}}$ and $lod_{\text{max}}$ are chosen so as to never clamp the normal range of $\lambda$. They may be respecified for a specific texture by calling \texttt{TexParameter} with $pname$ set to \texttt{TEXTURE\_MIN\_LOD} or \texttt{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

$$
\begin{align*}
  u(x, y) &= w_t \times s(x, y) + \delta_u \\
  v(x, y) &= h_t \times t(x, y) + \delta_v \\
  w(x, y) &= d_t \times r(x, y) + \delta_w 
\end{align*}
$$

where $w_t$, $h_t$, and $d_t$ are as defined by equation 3.15 with $w_s$, $h_s$, and $d_s$ equal to the width, height, and depth of the image array whose level is $level_{\text{base}}$. For a one-dimensional or one-dimensional array texture, define $v(x, y) \equiv 0$ and $w(x, y) \equiv 0$; for a two-dimensional, two-dimensional array, or cube map texture, define $w(x, y) \equiv 0$.

$(\delta_u, \delta_v, \delta_w)$ 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, or for fixed-function texture accesses, all three shader offsets are taken to be zero. If any of the offset values are outside the range of the implementation-defined values \texttt{MIN\_PROGRAM\_TEXEL\_OFFSET} and \texttt{MAX\_PROGRAM\_TEXEL\_OFFSET}, results of the texture lookup are undefined.

For a polygon, $\rho$ is given at a fragment with window coordinates $(x, y)$ by

$$
\rho = \max \left\{ \sqrt{\left( \frac{\partial u}{\partial x} \right)^2 + \left( \frac{\partial v}{\partial x} \right)^2 + \left( \frac{\partial w}{\partial x} \right)^2}, \sqrt{\left( \frac{\partial u}{\partial y} \right)^2 + \left( \frac{\partial v}{\partial y} \right)^2 + \left( \frac{\partial w}{\partial y} \right)^2} \right\}
$$

(3.20)

where $\partial u/\partial x$ indicates the derivative of $u$ with respect to window $x$, and similarly for the other derivatives.
3.9. TEXTURING

For a line, the formula is

$$\rho = \sqrt{\left(\frac{\partial u}{\partial x} \Delta x + \frac{\partial u}{\partial y} \Delta y\right)^2} + \left(\frac{\partial v}{\partial x} \Delta x + \frac{\partial v}{\partial y} \Delta y\right)^2 + \left(\frac{\partial w}{\partial x} \Delta x + \frac{\partial w}{\partial y} \Delta y\right)^2 / l,$$

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 $l = \sqrt{\Delta x^2 + \Delta y^2}$. For a point, pixel rectangle, or bitmap, $\rho \equiv 1$.

While it is generally agreed that equations 3.20 and 3.21 give the best results when texturing, they are often impractical to implement. Therefore, an implementation may approximate the ideal $\rho$ with a function $f(x, y)$ subject to these conditions:

1. $f(x, y)$ is continuous and monotonically increasing in each of $|\partial u/\partial x|$, $|\partial u/\partial y|$, $|\partial v/\partial x|$, $|\partial v/\partial y|$, $|\partial w/\partial x|$, and $|\partial w/\partial y|$.

2. Let

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

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

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

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

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

Version 3.0 (with changes marked - August 11, 2008)
3.9. TEXTURING

\[ u'(x, y) = \begin{cases} 
\text{clamp}(u(x, y), 0, w_t), & \text{TEXTURE_WRAP_S is CLAMP} \\
 u(x, y), & \text{otherwise} 
\end{cases} \]

\[ v'(x, y) = \begin{cases} 
\text{clamp}(v(x, y), 0, h_t), & \text{TEXTURE_WRAP_T is CLAMP} \\
v(x, y), & \text{otherwise} 
\end{cases} \]

\[ w'(x, y) = \begin{cases} 
\text{clamp}(w(x, y), 0, h_t), & \text{TEXTURE_WRAP_R is CLAMP} \\
w(x, y), & \text{otherwise} 
\end{cases} \]

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

The value assigned to TEXTURE_MIN_FILTER is used to determine how the texture value for a fragment is selected.

When the value of TEXTURE_MIN_FILTER is 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

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

and the value returned by \( \text{wrap()} \) is defined in table 3.22. For a three-dimensional texture, the texel at location \( (i, j, k) \) becomes the texture value. For two-dimensional, two-dimensional array, or cube map textures, \( k \) is irrelevant, and the texel at location \( (i, j) \) becomes the texture value. For one-dimensional texture or one-dimensional array textures, \( j \) and \( k \) are irrelevant, and the texel at location \( i \) becomes the texture value.

For one- and two-dimensional array textures, the texel is obtained from image layer \( l \), where

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

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

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

<table>
<thead>
<tr>
<th>Wrap mode</th>
<th>Result of wrap(coord)</th>
</tr>
</thead>
</table>
| CLAMP              | \(
\begin{cases}
\text{clamp}(\text{coord}, 0, \text{size} - 1), & \text{for \text{NEAREST} filtering} \\
\text{clamp}(\text{coord}, -1, \text{size}), & \text{for \text{LINEAR} filtering}
\end{cases}
\) |
| CLAMP_TO_EDGE      | \text{clamp}(\text{coord}, 0, \text{size} - 1) |
| CLAMP_TO_BORDER    | \text{clamp}(\text{coord}, -1, \text{size}) |
| REPEAT             | \text{fmod}(\text{coord}, \text{size}) |
| MIRRORED_REPEAT    | \( (\text{size} - 1) - \text{mirror}(\text{fmod}(\text{coord}, 2 \times \text{size}) - \text{size}) \) |

Table 3.22: Texel location wrap mode application. \( \text{fmod}(a, b) \) returns \( a - b \times \lfloor \frac{a}{b} \rfloor \). \( \text{mirror}(a) \) returns \( a \) if \( a \geq 0 \), and \( -(1 + a) \) otherwise. The values of \text{mode} and \text{size} are \text{TEXTURE\_WRAP\_S} and \( w \), \text{TEXTURE\_WRAP\_T} and \( h \), and \text{TEXTURE\_WRAP\_R} and \( d \) when wrapping \( i \), \( j \), or \( k \) coordinates, respectively.

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 \text{TEXTURE\_BORDER\_COLOR} are interpreted as an RGBA color to match the texture’s internal format in a manner consistent with table 3.15. The internal data type of the border values must be consistent with the type returned by the texture as described in section 3.9, or the result is undefined. The border values for texture components stored as fixed-point values are clamped to \([0, 1]\) before they are used. If the texture contains depth components, the first component of \text{TEXTURE\_BORDER\_COLOR} is interpreted as a depth value.

When the value of \text{TEXTURE\_MIN\_FILTER} is \text{LINEAR}, a \( 2 \times 2 \times 2 \) cube of texels in the image array of level \( \text{level} \) is selected. Let

\[
i_0 = \text{wrap}([u' - 0.5]) \\
i_1 = \text{wrap}([u' - 0.5] + 1) \\
alpha = \text{frac}(u' - 0.5)
\]

where \( \text{frac}(x) \) denotes the fractional part of \( x \).

Version 3.0 (with changes marked - August 11, 2008)
3.9. TEXTURING

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

$$
\tau = (1 - \alpha)(1 - \beta)(1 - \gamma)\tau_{i0j0k0} + \alpha(1 - \beta)(1 - \gamma)\tau_{i1j0k0} \\
+ (1 - \alpha)\beta(1 - \gamma)\tau_{i0j1k0} + \alpha\beta(1 - \gamma)\tau_{i1j1k0} \\
+ (1 - \alpha)(1 - \beta)\gamma\tau_{i0j0k1} + \alpha(1 - \beta)\gamma\tau_{i1j0k1} \\
+ (1 - \alpha)\beta\gamma\tau_{i0j1k1} + \alpha\beta\gamma\tau_{i1j1k1}
$$ (3.22)

where $\tau_{ijk}$ is the texel at location $(i, j, k)$ in the three-dimensional texture image.

For a two-dimensional, two-dimensional array, or cube map textures, texture array $\tau=(1 - \alpha)(1 - \beta)\tau_{i0j0} + \alpha(1 - \beta)\tau_{i1j0}$ + $(1 - \alpha)\beta\tau_{i0j1} + \alpha\beta\tau_{i1j1}$

where $\tau_{ij}$ is the texel at location $(i, j)$ in the two-dimensional texture image. For texture array two-dimensional array textures, all texels are obtained from layer $l$, where

$$
l = clamp([r + 0.5], 0, dt - 1).
$$

And for a one-dimensional or one-dimensional array texture, texture array $\tau = (1 - \alpha)\tau_{i0} + \alpha\tau_{i1}$

where $\tau_{i}$ is the texel at location $i$ in the one-dimensional texture. For one-dimensional array textures, both texels are obtained from layer $l$, where

$$
l = clamp([t + 0.5], 0, ht - 1).
$$

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 NEAREST filtering.

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. See chapter 4 for discussion of framebuffer objects and their attachments.

- The current DRAW_FRAMEBUFFER_BINDING names a framebuffer object $F$.
- The texture is attached to one of the attachment points, $A$, of framebuffer object $F$.
• The value of 
\texttt{TEXTURE\_MIN\_FILTER} is \texttt{NEAREST} or \texttt{LINEAR}, and the value of 
\texttt{FRAMEBUFFER\_ATTACHMENT\_TEXTURE\_LEVEL} for attachment point \texttt{A} is 
equal to the value of \texttt{TEXTURE\_BASE\_LEVEL}.

-or-

The value of \texttt{TEXTURE\_MIN\_FILTER} is \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 \texttt{A} is 
within the the inclusive range from \texttt{TEXTURE\_BASE\_LEVEL} to \texttt{q}.

Mipmapping

\texttt{TEXTURE\_MIN\_FILTER} values \texttt{NEAREST\_MIPMAP\_NEAREST}, 
\texttt{NEAREST\_MIPMAP\_LINEAR}, \texttt{LINEAR\_MIPMAP\_NEAREST}, and 
\texttt{LINEAR\_MIPMAP\_LINEAR} each require the use of a \textit{mipmap}. 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 \texttt{level\_base}, excluding its 
border, has dimensions $w_t \times h_t \times d_t$, then there are $[\log_2(\texttt{maxsize})] + 1$ 
levels in the mipmap. where

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

Numbering the levels such that level \texttt{level\_base} is the 0th level, the $i$th array has 
dimensions

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

where

\[\text{Bug 3534}\]
3.9. TEXTURING

\[ w_d = 2^i \]
\[ h_d = \begin{cases} 
1, & \text{for 1D and 1D array textures} \\
2^i, & \text{otherwise} 
\end{cases} \]
\[ d_d = \begin{cases} 
2^i, & \text{for 3D textures} \\
1, & \text{otherwise} 
\end{cases} \]

until the last array is reached with dimension \(1 \times 1 \times 1\).

Each array in a mipmap is defined using \texttt{TexImage3D}, \texttt{TexImage2D}, \texttt{CopyTexImage2D}, \texttt{TexImage1D}, or \texttt{CopyTexImage1D}; the array being set is indicated with the level-of-detail argument \texttt{level}. Level-of-detail numbers proceed from \texttt{level\_base} for the original \texttt{texel} array through \( p = \lceil \log_2(\text{maxsize}) \rceil + \texttt{level\_base} \) 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 \texttt{level\_base} through \( q = \min\{p, \texttt{level\_max}\} \) must be defined, as discussed in section 3.9.10.

The values of \texttt{level\_base} and \texttt{level\_max} may be respecified for a specific texture by calling \texttt{TexParameter} with \texttt{pname} set to \texttt{TEXTURE\_BASE\_LEVEL} or \texttt{TEXTURE\_MAX\_LEVEL} respectively.

The error \texttt{INVALID\_VALUE} is generated if either value is negative.

The mipmap is used in conjunction with the level of detail to approximate the application of an appropriately filtered texture to a fragment. Let \( c \) be the value of \( \lambda \) at which the transition from minification to magnification occurs (since this discussion pertains to minification, we are concerned only with values of \( \lambda \) where \( \lambda > c \)).

For mipmap filters \texttt{NEAREST\_MIPMAP\_NEAREST} and \texttt{LINEAR\_MIPMAP\_NEAREST}, the \( d \)th mipmap array is selected, where

\[ d = \begin{cases} 
\text{\texttt{level\_base}}, & \lambda \leq \frac{1}{2} \\
\lfloor \text{\texttt{level\_base}} + \lambda + \frac{1}{2} \rfloor - 1, & \lambda > \frac{1}{2}, \text{\texttt{level\_base}} + \lambda \leq q + \frac{1}{2} \\
q, & \lambda > \frac{1}{2}, \text{\texttt{level\_base}} + \lambda > q + \frac{1}{2} 
\end{cases} \quad (3.23) \]

The rules for \texttt{NEAREST} or \texttt{LINEAR} filtering are then applied to the selected array. Specifically, the coordinate \((u, v, w)\) is computed as in equation 3.19, 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 \texttt{NEAREST\_MIPMAP\_LINEAR} and \texttt{LINEAR\_MIPMAP\_LINEAR}, the level \( d_1 \) and \( d_2 \) mipmap arrays are selected, where

Version 3.0 (with changes marked - August 11, 2008)
3.9. TEXTURING

\[
d_1 = \begin{cases} 
q, & \text{if } \text{level}_{\text{base}} + \lambda \geq q \\
\lfloor \text{level}_{\text{base}} + \lambda \rfloor, & \text{otherwise}
\end{cases} \quad (3.24)
\]

\[
d_2 = \begin{cases} 
q, & \text{if } \text{level}_{\text{base}} + \lambda \geq q \\
d_1 + 1, & \text{otherwise}
\end{cases} \quad (3.25)
\]

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 3.19, 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 3.19, with \(w_s,\) \(h_s,\) and \(d_s\) equal to the width, height, and depth of the image array whose level is \(d_2\).

The final texture value is then found as

\[
\tau = [1 - \frac{\lambda}{\tau_1} + \frac{\lambda}{\tau_2}].
\]

Automatic Mipmap Generation

If the value of texture parameter \texttt{GENERATE\_MIPMAP} is TRUE, and a change is made to the interior or border texels of the \texttt{level\_base} array of a mipmap by one of the texture image specification operations defined in sections 3.9.1 through 3.9.3, then a \(^3\) complete set of mipmap arrays (as defined in section 3.9.10) will be computed. Array levels \(\text{level}_{\text{base}} + 1\) through \(p\) are replaced with arrays derived from the modified \texttt{level\_base} array, regardless of their previous contents. All other mipmap arrays, including the \texttt{level\_base} array, are left unchanged by this computation.

The internal formats and border widths of the derived mipmap arrays all match those of the \texttt{level\_base} array, and the dimensions of the derived arrays follow the requirements described in section 3.9.10.

The contents of the derived arrays are computed by repeated, filtered reduction of the \texttt{level\_base} array. For one- and two-dimensional array textures, each layer is texture array filtered independently. No particular filter algorithm is required, though a box filter is recommended as the default filter. In some implementations, filter quality may be affected by hints (section 5.6).

Automatic mipmap generation is available only for non-proxy texture image targets.

\(^3\)Automatic mipmap generation is not performed for changes resulting from rendering operations targeting a texel array bound as a color buffer of a framebuffer object.
3.9. 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 3.9.10.

Mipmap generation replaces texel array levels `levelbase + 1` through `q` with arrays derived from the `levelbase` array, as described above for Automatic Mipmap Generation. All other mipmap arrays, including the `levelbase` array, are left unchanged by this computation. For arrays in the range `levelbase + 1` through `q`, inclusive, automatic and manual mipmap generation generate the same derived arrays, given identical `levelbase` arrays.

3.9.8 Texture Magnification

When `\lambda` indicates magnification, the value assigned to `TEXTURE_MAG_FILTER` determines how the texture value is obtained. There are two possible values for `TEXTURE_MAG_FILTER`: NEAREST and LINEAR. NEAREST behaves exactly as NEAREST for `TEXTURE_MIN_FILTER` and LINEAR behaves exactly as LINEAR for `TEXTURE_MIN_FILTER` as described in section 3.9.7, including the texture coordinate wrap modes specified in table 3.22. The level-of-detail `levelbase` texel array is always used for magnification.

Finally, there is the choice of `c`, the minification vs. magnification switch-over point. If the magnification filter is given by LINEAR and the minification filter is given by NEAREST_MIPMAP_NEAREST or NEAREST_MIPMAP_LINEAR, then `c = 0.5`. This is done to ensure that a minified texture does not appear “sharper” than a magnified texture. Otherwise `c = 0`.

3.9.9 Combined Depth/Stencil Textures

If the texture image has a base internal format of `DEPTH_STENCIL`, then the stencil index texture component is ignored. The texture value `\tau` does not include a stencil index component, but includes only the depth component.
3.9. TEXTURING

3.9.10 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 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 $level_{base}$ through $q$ (where $q$ is defined in the Mipmapping discussion of section 3.9.7) were each specified with the same internal format.
- The border widths of each array are the same.
- The dimensions of the arrays follow the sequence described in the Mipmapping discussion of section 3.9.7.
- $level_{base} \leq level_{max}$
- Each dimension of the $level_{base}$ array is positive.
- If the internal format of the arrays is integer (see tables 3.16-3.17, TEXTURE_MAG_FILTER must be NEAREST and TEXTURE_MIN_FILTER must be NEAREST or NEAREST_MIPMAP_NEAREST.

Array levels $k$ where $k < level_{base}$ 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 $level_{base}$ arrays of each of the six texture images making up the cube map have identical, positive, and square dimensions.
- The $level_{base}$ arrays were each specified with the same internal format.
- The $level_{base}$ arrays each have the same border width.

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.
3.9. **TEXTURING**

**Effects of Completeness on Texture Application**

If one-, two-, or three-dimensional texturing (but not cube map texturing) is enabled for a texture unit at the time a primitive is rasterized, if TEXTURE_MIN_FILTER is one that requires a mipmap, and if the texture image bound to the enabled texture target is not complete, then it is as if texture mapping were disabled for that texture unit.

If cube map texturing is enabled for a texture unit at the time a primitive is rasterized, and if the bound cube map texture is not cube complete, then it is as if texture mapping were disabled for that texture unit. Additionally, if TEXTURE_MIN_FILTER is one that requires a mipmap, and if the texture is not mipmap cube complete, then it is as if texture mapping were disabled for that texture unit.

**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 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, excluding borders, 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).

**3.9.11 Texture State and Proxy State**

The state necessary for texture can be divided into two categories. First, there are the nine sets of mipmap arrays (one each for the one-, two-, and three-dimensional texture targets and six for the cube map texture targets) and their number. Each array has associated with it a width, height (two- and three-dimensional and cube map only), and depth (three-dimensional only), a border width, an integer describing the internal format of the image, eight integer values describing the resolutions of each of the red, green, blue, alpha, luminance, intensity, depth, and stencil components of the image, eight 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, zero border width, internal format \( 1 \), with the compressed flag set to \( \text{false} \), a zero compressed size, and zero-sized components). 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

Version 3.0 (with changes marked - August 11, 2008)
magnification filters, the wrap modes for $s$, $t$ (two- and three-dimensional and cube map only), and $r$ (three-dimensional only), the \text{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, a boolean indicating whether automatic mipmap generation should be performed, three integers describing the depth texture mode, compare mode, and compare function, and the priority associated with each set of properties. The value of the resident flag is determined by the GL and may change as a result of other GL operations. The flag may only be queried, not set, by applications (see section 3.9.12). In the initial state, the value assigned to \text{TEXTURE\_MIN\_FILTER} is \text{NEAREST\_MIPMAP\_LINEAR}, and the value for \text{TEXTURE\_MAG\_FILTER} is \text{LINEAR}. $s$, $t$, and $r$ wrap modes are all set to \text{REPEAT}. The values of \text{TEXTURE\_MIN\_LOD} and \text{TEXTURE\_MAX\_LOD} are -1000 and 1000 respectively. The values of \text{TEXTURE\_BASE\_LEVEL} and \text{TEXTURE\_MAX\_LEVEL} are 0 and 1000 respectively. \text{TEXTURE\_PRIORITY} is 1.0, and \text{TEXTURE\_BORDER\_COLOR} is (0,0,0,0). The value of \text{GENERATE\_MIPMAP} is false. The values of \text{DEPTH\_TEXTURE\_MODE}, \text{TEXTURE\_COMPARE\_MODE}, and \text{TEXTURE\_COMPARE\_FUNC} are \text{LUMINANCE}, \text{NONE}, and \text{LEQUAL} respectively. The initial value of \text{TEXTURE\_RESIDENT} is determined by the GL.

In addition to image arrays for one-, two-, and three-dimensional textures, one- and two-dimensional array textures, and the six image arrays for the cube map texture, partially instantiated image arrays are maintained for one-, two-, and three-dimensional textures 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, border width, and internal format state values, as well as state for the red, green, blue, alpha, luminance, intensity, depth, and stencil component resolutions and types. Proxy arrays do not include image data nor texture parameters. When \text{TexImage3D} is executed with \text{target} specified as \text{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 \text{TexImage3D} called with \text{target} set to \text{TEXTURE\_3D}, no error is generated, but the proxy width, height, depth, border width, and component resolutions are set to zero, and the component types are set to \text{NONE}. If the image array would be supported by such a call to \text{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 \text{TexImage1D} is executed with \text{target} specified as \text{PROXY\_TEXTURE\_1D}, \text{TexImage2D} is executed with \text{target} specified as \text{PROXY\_TEXTURE\_2D} or \text{PROXY\_TEXTURE\_1D\_ARRAY}, or
3.9. TEXTURING

TexImage3D is executed with target specified as PROXY_TEXTURE_2D_ARRAY.

The cube map proxy arrays are operated on in the same manner when TexImage2D is executed with the target field specified as PROXY_TEXTURE_CUBE_MAP, with the addition that determining that a given cube map texture is supported with PROXY_TEXTURE_CUBE_MAP indicates that all six of the cube map 2D images are supported. Likewise, if the specified PROXY_TEXTURE_CUBE_MAP is not supported, none of the six cube map 2D images are supported.

There is no image associated with any of the proxy textures. Therefore PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, and PROXY_TEXTURE_3D, and PROXY_TEXTURE_CUBE_MAP cannot be used as textures, and their images must never be queried using GetTexImage. The error INVALID_ENUM is generated if this is attempted. Likewise, there is no non level-related state associated with a proxy texture, and GetTexParameteriv or GetTexParameterfv may not be called with a proxy texture target. The error INVALID_ENUM is generated if this is attempted.

3.9.12 Texture Objects

In addition to the default textures TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, and TEXTURE_CUBE_MAP, named one-, two-, and three-dimensional, one- and two-dimensional array, and cube map 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 TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, or TEXTURE_CUBE_MAP. The binding is effected by calling

```c
voidBindTexture( 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 3.9.11, set to the same initial values. If the new texture object is bound to TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, or TEXTURE_CUBE_MAP, it is and remains a one-, two-, three-dimensional, one- or two-dimensional array, or cube map texture respectively until it is deleted.

BindTexture may also be used to bind an existing texture object to either TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, or TEXTURE_CUBE_MAP. The error INVALID_OPERATION is generated if an attempt is made to bind a texture object of different dimensionality

---

Version 3.0 (with changes marked - August 11, 2008)
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.

While a texture object is bound, GL operations on the target to which it is
bound affect the bound object, and queries of the target to which it is bound return
state from the bound object. If texture mapping of the dimensionality of the target
to which a texture object is bound is enabled, the state of the bound texture object
directs the texturing operation.

In the initial state, TEXTURE_1D, TEXTURE_2D, TEXTURE_3D,
TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, and TEXTURE_CUBE_MAP have one-
two-, three-dimensional, one- and two-dimensional array, and cube map 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 ray, and cube map texture is therefore operated upon, queried,
and applied as TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY,
TEXTURE_2D_ARRAY, or TEXTURE_CUBE_MAP 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 one of the targets TEXTURE_1D, TEXTURE_2D,
TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, or TEXTURE_CUBE_MAP
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 sec-
tion 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.

An implementation may choose to establish a working set of texture objects on
which binding operations are performed with higher performance. A texture object
that is currently part of the working set is said to be resident. The command

Version 3.0 (with changes marked - August 11, 2008
3.9. TEXTURING

boolean AreTexturesResident( sizei n, uint *textures,
  boolean *residences);

returns TRUE if all of the \( n \) texture objects named in textures are resident, or if the implementation does not distinguish a working set. If at least one of the texture objects named in textures is not resident, then FALSE is returned, and the residence of each texture object is returned in residences. Otherwise the contents of residences are not changed. If any of the names in textures are unused or are zero, FALSE is returned, the error INVALID_VALUE is generated, and the contents of residences are indeterminate. The residence status of a single bound texture object can also be queried by calling GetTexParameteriv or GetTexParameterfv with target set to the target to which the texture object is bound, and pname set to TEXTURE_RESIDENT.

AreTexturesResident indicates only whether a texture object is currently resident, not whether it could not be made resident. An implementation may choose to make a texture object resident only on first use, for example. The client may guide the GL implementation in determining which texture objects should be resident by specifying a priority for each texture object. The command

\[
\text{void PrioritizeTextures( sizei n, uint *textures,}
\text{ clampf *priorities );}
\]

sets the priorities of the \( n \) texture objects named in textures to the values in priorities. Each priority value is clamped to the range \([0,1]\) before it is assigned. Zero indicates the lowest priority, with the least likelihood of being resident. One indicates the highest priority, with the greatest likelihood of being resident. The priority of a single bound texture object may also be changed by calling TexParameteri, TexParameterf, TexParameteriv, or TexParameterfv with target set to the target to which the texture object is bound, pname set to TEXTURE_PRIORITY, and param or params specifying the new priority value (which is clamped to the range \([0,1]\) before being assigned). PrioritizeTextures silently ignores attempts to prioritize unused texture object names or zero (default textures).

The texture object name space, including the initial one-, two-, and three-dimensional, one- and two-dimensional array, and cube map texture objects, is texture_array shared among all texture units. A texture object may be bound to more than one texture unit simultaneously. After a texture object is bound, any GL operations on that target object affect any other texture units to which the same texture object is bound.

Texture binding is affected by the setting of the state ACTIVE_TEXTURE.

If a texture object is deleted, it as if all texture units which are bound to that texture object are rebound to texture object zero.

Version 3.0 (with changes marked - August 11, 2008
3.9. TEXTURING

3.9.13 Texture Environments and Texture Functions

The command

```c
void TexEnv{if}( enum target, enum pname, T param );
void TexEnv{if}v( enum target, enum pname, T params );
```

sets parameters of the texture environment that specifies how texture values are interpreted when texturing a fragment, or sets per-texture-unit filtering parameters.

- `target` must be one of POINT_SPRITE, TEXTURE_ENV or TEXTURE_FILTER_CONTROL. `pname` is a symbolic constant indicating the parameter to be set. In the first form of the command, `param` is a value to which to set a single-valued parameter; in the second form, `params` is a pointer to an array of parameters: either a single symbolic constant or a value or group of values to which the parameter should be set.

When `target` is POINT_SPRITE, point sprite rasterization behavior is affected as described in section 3.4.

When `target` is TEXTURE_FILTER_CONTROL, `pname` must be TEXTURE_LOD_BIAS. In this case the parameter is a single signed floating point value, `bias_texunit`, that biases the level of detail parameter $\lambda$ as described in section 3.9.7.

When `target` is TEXTURE_ENV, the possible environment parameters are TEXTURE_ENV_MODE, TEXTURE_ENV_COLOR, COMBINE_RGB, COMBINE_ALPHA, RGB_SCALE, ALPHA_SCALE, SRCn_RGB, SRCn_ALPHA, OPERANDn_RGB, and OPERANDn_ALPHA, where $n = 0, 1, \text{ or } 2$. TEXTURE_ENV_MODE may be set to one of REPLACE, MODULATE, DECAL, BLEND, ADD, or COMBINE. TEXTURE_ENV_COLOR is set to an RGBA color by providing four single-precision floating-point values. If integers are provided for TEXTURE_ENV_COLOR, then they are converted to floating-point as specified in table 2.10 for signed integers.

The value of TEXTURE_ENV_MODE specifies a texture function. The result of this function depends on the fragment and the texel array value. The precise form of the function depends on the base internal formats of the texel arrays that were last specified.

$C_f$ and $A_f$ are the primary color components of the incoming fragment; $C_s$ and $A_s$ are the components of the texture source color, derived from the filtered texture values $R_t$, $G_t$, $B_t$, $A_t$, $L_t$, and $I_t$ as shown in table 3.23; $C_c$ and $A_c$ are the components of the texture environment color; $C_p$ and $A_p$ are the components

In the remainder of section 3.9.13, the notation $C_x$ is used to denote each of the three components $R_x$, $G_x$, and $B_x$ of a color specified by $x$. Operations on $C_x$ are performed independently for each color component. The $A$ component of colors is usually operated on in a different fashion, and is therefore denoted separately by $A_x$. 

Bug 3350 - Add R/RGB formats to table 3.23

Version 3.0 (with changes marked - August 11, 2008)
3.9. TEXTURING

<table>
<thead>
<tr>
<th>Texture Base Internal Format</th>
<th>Texture source color $C_s$</th>
<th>$A_s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA</td>
<td>$(0, 0, 0)$</td>
<td>$A_L$</td>
</tr>
<tr>
<td>LUMINANCE</td>
<td>$(L_t, L_t, L_t)$</td>
<td>1</td>
</tr>
<tr>
<td>LUMINANCE_ALPHA</td>
<td>$(L_t, L_t, L_t)$</td>
<td>$A_L$</td>
</tr>
<tr>
<td>INTENSITY</td>
<td>$(I_t, I_t, I_t)$</td>
<td>$I_t$</td>
</tr>
<tr>
<td>RED</td>
<td>$(R_t, 0, 0)$</td>
<td>1</td>
</tr>
<tr>
<td>RG</td>
<td>$(R_t, G_t, 0)$</td>
<td>1</td>
</tr>
<tr>
<td>RGB</td>
<td>$(R_t, G_t, B_t)$</td>
<td>1</td>
</tr>
<tr>
<td>RGBA</td>
<td>$(R_t, G_t, B_t)$</td>
<td>$A_L$</td>
</tr>
</tbody>
</table>

Table 3.23: Correspondence of filtered texture components to texture source components.

resulting from the previous texture environment (for texture environment 0, $C_p$ and $A_p$ are identical to $C_f$ and $A_f$, respectively); and $C_v$ and $A_v$ are the primary color components computed by the texture function.

If fragment color clamping is enabled, all of these color values, including the results, are clamped to the range $[0, 1]$. If fragment color clamping is disabled, the values are not clamped. The texture functions are specified in tables 3.24, 3.25, and 3.26.

If the value of TEXTURE_ENV_MODE is COMBINE, the form of the texture function depends on the values of COMBINE_RGB and COMBINE_ALPHA, according to table 3.26. The RGB and ALPHA results of the texture function are then multiplied by the values of RGB_SCALE and ALPHA_SCALE, respectively. If fragment color clamping is enabled, the arguments and results used in table 3.26 are clamped to $[0, 1]$. Otherwise, the results are unmodified.

The arguments Arg0, Arg1, and Arg2 are determined by the values of SRCn_RGB, SRCn_ALPHA, OPERANDn_RGB and OPERANDn_ALPHA, where $n = 0$, 1, or 2, as shown in tables 3.27 and 3.28. $C_s^n$ and $A_s^n$ denote the texture source color and alpha from the texture image bound to texture unit $n$.

The state required for the current texture environment, for each texture unit, consists of a six-valued integer indicating the texture function, an eight-valued integer indicating the RGB combiner function and a six-valued integer indicating the ALPHA combiner function, six four-valued integers indicating the combiner RGB and ALPHA source arguments, three four-valued integers indicating the combiner RGB operands, three two-valued integers indicating the combiner ALPHA operands, and four floating-point environment color values. In the initial state, the texture

Version 3.0 (with changes marked - August 11, 2008)
### 3.9. TEXTURING

<table>
<thead>
<tr>
<th>Texture Base</th>
<th>Internal Format</th>
<th>REPLACE Function</th>
<th>MODULATE Function</th>
<th>DECAL Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA</td>
<td>( C_v = C_p )</td>
<td>( C_v = C_p )</td>
<td>( C_v = C_p )</td>
<td>undefined</td>
</tr>
<tr>
<td></td>
<td>( A_v = A_s )</td>
<td>( A_v = A_pA_s )</td>
<td>( A_v = A_pA_s )</td>
<td>undefined</td>
</tr>
<tr>
<td>LUMINANCE</td>
<td>( C_v = C_s )</td>
<td>( C_v = C_pC_s )</td>
<td>undefined</td>
<td></td>
</tr>
<tr>
<td>(or 1)</td>
<td>( A_v = A_p )</td>
<td>( A_v = A_p )</td>
<td>undefined</td>
<td></td>
</tr>
<tr>
<td>LUMINANCE,ALPHA</td>
<td>( C_v = C_s )</td>
<td>( C_v = C_pC_s )</td>
<td>undefined</td>
<td></td>
</tr>
<tr>
<td>(or 2)</td>
<td>( A_v = A_s )</td>
<td>( A_v = A_pA_s )</td>
<td>undefined</td>
<td></td>
</tr>
<tr>
<td>INTENSITY</td>
<td>( C_v = C_s )</td>
<td>( C_v = C_pC_s )</td>
<td>undefined</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( A_v = A_s )</td>
<td>( A_v = A_pA_s )</td>
<td>undefined</td>
<td></td>
</tr>
<tr>
<td>RGB, RG, RED,</td>
<td>( C_v = C_s )</td>
<td>( C_v = C_pC_s )</td>
<td>undefined</td>
<td></td>
</tr>
<tr>
<td>or 3</td>
<td>( A_v = A_p )</td>
<td>( A_v = A_p )</td>
<td>undefined</td>
<td></td>
</tr>
<tr>
<td>RGBA</td>
<td>( C_v = C_s )</td>
<td>( C_v = C_pC_s )</td>
<td>undefined</td>
<td></td>
</tr>
<tr>
<td>or 4</td>
<td>( A_v = A_s )</td>
<td>( A_v = A_pA_s )</td>
<td>undefined</td>
<td></td>
</tr>
</tbody>
</table>

**Table 3.24:** Texture functions REPLACE, MODULATE, and DECAL.

<table>
<thead>
<tr>
<th>Texture Base</th>
<th>BLEND Function</th>
<th>ADD Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA</td>
<td>( C_v = C_p )</td>
<td>( C_v = C_p )</td>
</tr>
<tr>
<td></td>
<td>( A_v = A_pA_s )</td>
<td>( A_v = A_pA_s )</td>
</tr>
<tr>
<td>LUMINANCE</td>
<td>( C_v = C_p(1 - C_s) + C_cC_s )</td>
<td>( C_v = C_p + C_s )</td>
</tr>
<tr>
<td>(or 1)</td>
<td>( A_v = A_p )</td>
<td>( A_v = A_p )</td>
</tr>
<tr>
<td>LUMINANCE,ALPHA</td>
<td>( C_v = C_p(1 - C_s) + C_cC_s )</td>
<td>( C_v = C_p + C_s )</td>
</tr>
<tr>
<td>(or 2)</td>
<td>( A_v = A_pA_s )</td>
<td>( A_v = A_pA_s )</td>
</tr>
<tr>
<td>INTENSITY</td>
<td>( C_v = C_p(1 - C_s) + C_cC_s )</td>
<td>( C_v = C_p + C_s )</td>
</tr>
<tr>
<td></td>
<td>( A_v = A_p(1 - A_s) + A_cA_s )</td>
<td>( A_v = A_p + A_s )</td>
</tr>
<tr>
<td>RGB, RG, RED,</td>
<td>( C_v = C_p(1 - C_s) + C_cC_s )</td>
<td>( C_v = C_p + C_s )</td>
</tr>
<tr>
<td>or 3</td>
<td>( A_v = A_p )</td>
<td>( A_v = A_p )</td>
</tr>
<tr>
<td>RGBA</td>
<td>( C_v = C_p(1 - C_s) + C_cC_s )</td>
<td>( C_v = C_p + C_s )</td>
</tr>
<tr>
<td>or 4</td>
<td>( A_v = A_pA_s )</td>
<td>( A_v = A_pA_s )</td>
</tr>
</tbody>
</table>

**Table 3.25:** Texture functions BLEND and ADD.

Version 3.0 (with changes marked - August 11, 2008)
3.9. TEXTURING

<table>
<thead>
<tr>
<th>COMBINE_RGB</th>
<th>Texture Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPLACE</td>
<td>Arg0</td>
</tr>
<tr>
<td>MODULATE</td>
<td>Arg0 * Arg1</td>
</tr>
<tr>
<td>ADD</td>
<td>Arg0 + Arg1</td>
</tr>
<tr>
<td>ADD_SIGNED</td>
<td>Arg0 + Arg1 - 0.5</td>
</tr>
<tr>
<td>INTERPOLATE</td>
<td>Arg0 * Arg2 + Arg1 * (1 - Arg2)</td>
</tr>
<tr>
<td>SUBTRACT</td>
<td>Arg0 - Arg1</td>
</tr>
<tr>
<td>DOT3_RGB</td>
<td>4 × ((Arg0_r - 0.5) * (Arg1_r - 0.5) + (Arg0_g - 0.5) * (Arg1_g - 0.5) + (Arg0_b - 0.5) * (Arg1_b - 0.5))</td>
</tr>
<tr>
<td>DOT3_RGBA</td>
<td>4 × ((Arg0_r - 0.5) * (Arg1_r - 0.5) + (Arg0_g - 0.5) * (Arg1_g - 0.5) + (Arg0_b - 0.5) * (Arg1_b - 0.5))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COMBINE_ALPHA</th>
<th>Texture Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPLACE</td>
<td>Arg0</td>
</tr>
<tr>
<td>MODULATE</td>
<td>Arg0 * Arg1</td>
</tr>
<tr>
<td>ADD</td>
<td>Arg0 + Arg1</td>
</tr>
<tr>
<td>ADD_SIGNED</td>
<td>Arg0 + Arg1 - 0.5</td>
</tr>
<tr>
<td>INTERPOLATE</td>
<td>Arg0 * Arg2 + Arg1 * (1 - Arg2)</td>
</tr>
<tr>
<td>SUBTRACT</td>
<td>Arg0 - Arg1</td>
</tr>
</tbody>
</table>

Table 3.26: COMBINE texture functions. The scalar expression computed for the DOT3_RGB and DOT3_RGBA functions is placed into each of the 3 (RGB) or 4 (RGBA) components of the output. The result generated from COMBINE_ALPHA is ignored for DOT3_RGBA.
### 3.9. TEXTURING

<table>
<thead>
<tr>
<th>SRCn.RGB</th>
<th>OPERANDn.RGB</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXTURE</td>
<td>SRC_COLOR</td>
<td>$C_s$</td>
</tr>
<tr>
<td></td>
<td>ONE_MINUS_SRC_COLOR</td>
<td>$1 - C_s$</td>
</tr>
<tr>
<td></td>
<td>SRC_ALPHA</td>
<td>$A_s$</td>
</tr>
<tr>
<td></td>
<td>ONE_MINUS_SRC_ALPHA</td>
<td>$1 - A_s$</td>
</tr>
<tr>
<td>TEXTURE$n$</td>
<td>SRC_COLOR</td>
<td>$C_s^n$</td>
</tr>
<tr>
<td></td>
<td>ONE_MINUS_SRC_COLOR</td>
<td>$1 - C_s^n$</td>
</tr>
<tr>
<td></td>
<td>SRC_ALPHA</td>
<td>$A_s^n$</td>
</tr>
<tr>
<td></td>
<td>ONE_MINUS_SRC_ALPHA</td>
<td>$1 - A_s^n$</td>
</tr>
<tr>
<td>CONSTANT</td>
<td>SRC_COLOR</td>
<td>$C_c$</td>
</tr>
<tr>
<td></td>
<td>ONE_MINUS_SRC_COLOR</td>
<td>$1 - C_c$</td>
</tr>
<tr>
<td></td>
<td>SRC_ALPHA</td>
<td>$A_c$</td>
</tr>
<tr>
<td></td>
<td>ONE_MINUS_SRC_ALPHA</td>
<td>$1 - A_c$</td>
</tr>
<tr>
<td>PRIMARY_COLOR</td>
<td>SRC_COLOR</td>
<td>$C_f$</td>
</tr>
<tr>
<td></td>
<td>ONE_MINUS_SRC_COLOR</td>
<td>$1 - C_f$</td>
</tr>
<tr>
<td></td>
<td>SRC_ALPHA</td>
<td>$A_f$</td>
</tr>
<tr>
<td></td>
<td>ONE_MINUS_SRC_ALPHA</td>
<td>$1 - A_f$</td>
</tr>
<tr>
<td>PREVIOUS</td>
<td>SRC_COLOR</td>
<td>$C_p$</td>
</tr>
<tr>
<td></td>
<td>ONE_MINUS_SRC_COLOR</td>
<td>$1 - C_p$</td>
</tr>
<tr>
<td></td>
<td>SRC_ALPHA</td>
<td>$A_p$</td>
</tr>
<tr>
<td></td>
<td>ONE_MINUS_SRC_ALPHA</td>
<td>$1 - A_p$</td>
</tr>
</tbody>
</table>

Table 3.27: Arguments for `COMBINE_RGB` functions.

<table>
<thead>
<tr>
<th>SRCn_ALPHA</th>
<th>OPERANDn_ALPHA</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXTURE</td>
<td>SRC_ALPHA</td>
<td>$A_s$</td>
</tr>
<tr>
<td></td>
<td>ONE_MINUS_SRC_ALPHA</td>
<td>$1 - A_s$</td>
</tr>
<tr>
<td>TEXTURE$n$</td>
<td>SRC_ALPHA</td>
<td>$A_s^n$</td>
</tr>
<tr>
<td></td>
<td>ONE_MINUS_SRC_ALPHA</td>
<td>$1 - A_s^n$</td>
</tr>
<tr>
<td>CONSTANT</td>
<td>SRC_ALPHA</td>
<td>$A_c$</td>
</tr>
<tr>
<td></td>
<td>ONE_MINUS_SRC_ALPHA</td>
<td>$1 - A_c$</td>
</tr>
<tr>
<td>PRIMARY_COLOR</td>
<td>SRC_ALPHA</td>
<td>$A_f$</td>
</tr>
<tr>
<td></td>
<td>ONE_MINUS_SRC_ALPHA</td>
<td>$1 - A_f$</td>
</tr>
<tr>
<td>PREVIOUS</td>
<td>SRC_ALPHA</td>
<td>$A_p$</td>
</tr>
<tr>
<td></td>
<td>ONE_MINUS_SRC_ALPHA</td>
<td>$1 - A_p$</td>
</tr>
</tbody>
</table>

Table 3.28: Arguments for `COMBINE_ALPHA` functions.

Version 3.0 (with changes marked - August 11, 2008)
3.9. TEXTURING

and combiner functions are each MODULATE, the combiner RGB and ALPHA sources
are each TEXTURE, PREVIOUS, and CONSTANT for sources 0, 1, and 2 respectively,
the combiner RGB operands for sources 0 and 1 are each SRC_COLOR, the combiner
RGB operand for source 2, as well as for the combiner ALPHA operands, are each
SRC_ALPHA, and the environment color is (0, 0, 0, 0).

The state required for the texture filtering parameters, for each texture unit,
consists of a single floating-point level of detail bias. The initial value of the bias
is 0.0.

3.9.14 Texture Comparison Modes

Texture values can also be computed according to a specified comparison func-
tion. Texture parameter TEXTURE_COMPARE_MODE specifies the comparison
operands, and parameter TEXTURE_COMPARE_FUNC specifies the comparison func-
tion. The format of the resulting texture sample is determined by the value of
DEPTH_TEXTURE_MODE.

Depth Texture Comparison Mode

If the currently bound texture’s base internal format is DEPTH_COMPONENT or
DEPTH_STENCIL, then TEXTURE_COMPARE_MODE, TEXTURE_COMPARE_FUNC and
DEPTH_TEXTURE_MODE 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, defined as
follows:

- For fixed-function, non-cubemap texture lookups, $D_{ref}$ is the interpolated $r$
texture coordinate.
- For fixed-function, cubemap texture lookups, $D_{ref}$ is the interpolated $q$ tex-
ture coordinate.
- For texture lookups generated by an OpenGL Shading Language lookup
function, $D_{ref}$ is the reference value for depth comparisons provided by the
lookup function.

If the texture’s internal format indicates a fixed-point depth texture, then $D_t$
and $D_{ref}$ are clamped to the range $[0, 1]$; otherwise no clamping is performed.
Then the effective texture value is computed as follows:

If the value of TEXTURE_COMPARE_MODE is NONE, then
If the value of TEXTURE_COMPARE_MODE is COMPARE_REF_TO_TEXTURE, then $r$ depends on the texture comparison function as shown in table 3.29.

<table>
<thead>
<tr>
<th>Texture Comparison Function</th>
<th>Computed result $r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEQUAL</td>
<td>$r = \begin{cases} 1.0, &amp; D_{\text{ref}} \leq D_t \ 0.0, &amp; D_{\text{ref}} &gt; D_t \end{cases}$</td>
</tr>
<tr>
<td>GEQUAL</td>
<td>$r = \begin{cases} 1.0, &amp; D_{\text{ref}} \geq D_t \ 0.0, &amp; D_{\text{ref}} &lt; D_t \end{cases}$</td>
</tr>
<tr>
<td>LESS</td>
<td>$r = \begin{cases} 1.0, &amp; D_{\text{ref}} &lt; D_t \ 0.0, &amp; D_{\text{ref}} \geq D_t \end{cases}$</td>
</tr>
<tr>
<td>GREATER</td>
<td>$r = \begin{cases} 1.0, &amp; D_{\text{ref}} &gt; D_t \ 0.0, &amp; D_{\text{ref}} \leq D_t \end{cases}$</td>
</tr>
<tr>
<td>EQUAL</td>
<td>$r = \begin{cases} 1.0, &amp; D_{\text{ref}} = D_t \ 0.0, &amp; D_{\text{ref}} \neq D_t \end{cases}$</td>
</tr>
<tr>
<td>NOTEQUAL</td>
<td>$r = \begin{cases} 1.0, &amp; D_{\text{ref}} \neq D_t \ 0.0, &amp; D_{\text{ref}} = D_t \end{cases}$</td>
</tr>
<tr>
<td>ALWAYS</td>
<td>$r = 1.0$</td>
</tr>
<tr>
<td>NEVER</td>
<td>$r = 0.0$</td>
</tr>
</tbody>
</table>

Table 3.29: Depth texture comparison functions.

The resulting $r$ is assigned to $R_t$, $L_t$, $I_t$, or $A_t$ if the value of DEPTH_TEXTURE_MODE is respectively RED, LUMINANCE, INTENSITY, or ALPHA.

If the value of TEXTURE_MAG_FILTER is not NEAREST, or the value of TEXTURE_MIN_FILTER is not NEAREST or NEAREST_MIPMAP_NEAREST, then $r$ may be computed by comparing more than one depth texture value to the texture reference value. The details of this are implementation-dependent, but $r$ should be a value in the range $[0, 1]$ which is proportional to the number of comparison passes or failures.

### 3.9.15 sRGB Texture Color Conversion

If the currently bound texture’s internal format is one of SRGB, SRGB8, SRGB_ALPHA, SRGB8_ALPHA8, SLUMINANCE_ALPHA, SLUMINANCE8_ALPHA8,
3.9. TEXTURING

SLUMINANCE, SLUMINANCE8, COMPRESSED_SRGB, COMPRESSED_SRGB_ALPHA, COMPRESSED_SLUMINANCE, or COMPRESSED_SLUMINANCE_ALPHA, the red, green, and blue components are converted from an sRGB color space to a linear color space as part of filtering described in sections 3.9.7 and 3.9.8. 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.

$$
c_l = \begin{cases} 
\frac{c_s}{12.92}, & c_s \leq 0.04045 \\
(c_s + 0.055)^{2.4} / 1.055, & c_s > 0.04045 
\end{cases} \quad (3.26)
$$

Assume $c_s$ is the sRGB component in the range $[0, 1]$.

3.9.16 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, green, blue, and $exp_{shared}$ values (see section 3.9.1) are treated as unsigned integers and are converted to red, green, and blue as follows:

$$
\begin{align*}
\text{red} &= red_s2^{exp_{\text{shared}}-B} \\
\text{green} &= green_s2^{exp_{\text{shared}}-B} \\
\text{blue} &= blue_s2^{exp_{\text{shared}}-B}
\end{align*}
$$

3.9.17 Texture Application

Texturing is enabled or disabled using the generic Enable and Disable commands, respectively, with the symbolic constants TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, or TEXTURE_CUBE_MAP to enable the one-, two, three-dimensional, or cube map texture, respectively. If both two- and one-dimensional textures are enabled, the two-dimensional texture is used. If the three-dimensional and either of the two- or one-dimensional textures is enabled, the three-dimensional texture is used. If the cube map texture and any of the three-, two-, or one-dimensional textures is enabled, then cube map texturing is used.
If all texturing is disabled, a rasterized fragment is passed on unaltered to the next stage of the GL (although its texture coordinates may be discarded). Otherwise, a texture value is found according to the parameter values of the currently bound texture image of the appropriate dimensionality using the rules given in sections 3.9.6 through 3.9.8. This texture value is used along with the incoming fragment in computing the texture function indicated by the currently bound texture environment. The result of this function replaces the incoming fragment’s primary R, G, B, and A values. These are the color values passed to subsequent operations. Other data associated with the incoming fragment remain unchanged, except that the texture coordinates may be discarded.

Note that the texture value may contain R, G, B, A, L, I, or D components, but it does not contain an S component. If the texture’s base internal format is DEPTH_STENCIL, for the purposes of texture application it is as if the base internal format were DEPTH_COMPONENT.

Each texture unit is enabled and bound to texture objects independently from the other texture units. Each texture unit follows the precedence rules for one-, two-, three-dimensional, and cube map textures. Thus texture units can be performing texture mapping of different dimensionalities simultaneously. Each unit has its own enable and binding states.

Each texture unit is paired with an environment function, as shown in figure 3.11. The second texture function is computed using the texture value from the second texture, the fragment resulting from the first texture function computation and the second texture unit’s environment function. If there is a third texture, the fragment resulting from the second texture function is combined with the third texture value using the third texture unit’s environment function and so on. The texture unit selected by ActiveTexture determines which texture unit’s environment is modified by TexEnv calls.

If the value of TEXTURE_ENV_MODE is COMBINE, the texture function associated with a given texture unit is computed using the values specified by SRCn_RGB, SRCn_ALPHA, OPERANDn_RGB and OPERANDn_ALPHA. If TEXTUREn is specified as SRCn_RGB or SRCn_ALPHA, the texture value from texture unit n will be used in computing the texture function for this texture unit.

Texturing is enabled and disabled individually for each texture unit. If texturing is disabled for one of the units, then the fragment resulting from the previous unit is passed unaltered to the following unit. Individual texture units beyond those specified by MAX_TEXTURE_UNITS are always treated as disabled.

If a texture unit is disabled or has an invalid or incomplete texture (as defined in section 3.9.10) bound to it, then blending is disabled for that texture unit. If the texture environment for a given enabled texture unit references a disabled texture unit, or an invalid or incomplete texture that is bound to another unit, then the
3.9. TEXTURING

Figure 3.11. Multitexture pipeline. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation. The input fragment color is successively combined with each texture according to the state of the corresponding texture environment, and the resulting fragment color passed as input to the next texture unit in the pipeline.
results of texture blending are undefined.

The required state, per texture unit, is four bits indicating whether each of one-, two-, three-dimensional, or cube map texturing is enabled or disabled. In the initial state, all texturing is disabled for all texture units.

### 3.10 Color Sum

At the beginning of color sum, a fragment has two RGBA colors: a primary color \( c_{pri} \) (which texturing, if enabled, may have modified) and a secondary color \( c_{sec} \).

If color sum is enabled, the R, G, and B components of these two colors are summed to produce a single post-texturing RGBA color \( c \). The A component of \( c \) is taken from the A component of \( c_{pri} \); the A component of \( c_{sec} \) is unused. If color sum is disabled, then \( c_{pri} \) is assigned to \( c \). If fragment color clamping is enabled, the components of \( c \) are then clamped to the range \([0, 1] \).

Color sum is enabled or disabled using the generic \texttt{Enable} and \texttt{Disable} commands, respectively, with the symbolic constant \texttt{COLOR\_SUM}. If lighting is enabled and if a vertex shader is not active, the color sum stage is always applied, ignoring the value of \texttt{COLOR\_SUM}.

The state required is a single bit indicating whether color sum is enabled or disabled. In the initial state, color sum is disabled.

Color sum has no effect in color index mode, or if a fragment shader is active.

### 3.11 Fog

If enabled, fog blends a fog color with a rasterized fragment’s post-texturing color using a blending factor \( f \). Fog is enabled and disabled with the \texttt{Enable} and \texttt{Disable} commands using the symbolic constant \texttt{FOG}.

This factor \( f \) is computed according to one of three equations:

\[
 f = \exp(-d \cdot c), \quad (3.27)
\]

\[
 f = \exp(-(d \cdot c)^2), \quad \text{or} \quad (3.28)
\]

\[
 f = \frac{e - c}{e - s} \quad (3.29)
\]

If a vertex shader is active, or if the fog source, as defined below, is \texttt{FOG\_COORD}, then \( c \) is the interpolated value of the fog coordinate for this fragment. Otherwise, if the fog source is \texttt{FRAGMENT\_DEPTH}, then \( c \) is the eye-coordinate distance from
the eye, \((0, 0, 0, 1)\) in eye coordinates, to the fragment center. The equation and the fog source, along with either \(d\) or \(e\) and \(s\), is specified with

```c
void Fog(void (enum pname, T param));
void Fog(void (enum pname, T params));
```

If \(pname\) is \texttt{FOG\_MODE}, then \(param\) must be, or \(params\) must point to an integer that is one of the symbolic constants \texttt{EXP}, \texttt{EXP2}, or \texttt{LINEAR}, in which case equation 3.27, 3.28, or 3.29, respectively, is selected for the fog calculation (if, when 3.29 is selected, \(e = s\), results are undefined). If \(pname\) is \texttt{FOG\_COORD\_SRC}, then \(param\) must be, or \(params\) must point to an integer that is one of the symbolic constants \texttt{FRAGMENT\_DEPTH} or \texttt{FOG\_COORD}. If \(pname\) is \texttt{FOG\_DENSITY}, \texttt{FOG\_START}, or \texttt{FOG\_END}, then \(param\) is or \(params\) points to a value that is \(d\), \(s\), or \(e\), respectively. If \(d\) is specified less than zero, the error \texttt{INVALID\_VALUE} results.

An implementation may choose to approximate the eye-coordinate distance from the eye to each fragment center by \(|z_e|\). Further, \(f\) need not be computed at each fragment, but may be computed at each vertex and interpolated as other data are.

No matter which equation and approximation is used to compute \(f\), the result is clamped to \([0, 1]\) to obtain the final \(f\).

\(f\) is used differently depending on whether the GL is in RGBA or color index mode. In RGBA mode, if \(C_r\) represents a rasterized fragment’s R, G, or B value, then the corresponding value produced by fog is

\[
C = fC_r + (1 - f)C_f.
\]

(The rasterized fragment’s A value is not changed by fog blending.) The R, G, B, and A values of \(C_f\) are specified by calling \texttt{Fog} with \(pname\) equal to \texttt{FOG\_COLOR}; in this case \(params\) points to four values comprising \(C_f\). If these are not floating-point values, then they are converted to floating-point using the conversion given in table 2.10 for signed integers. If fragment color clamping is enabled, the components of \(C_r\) and \(C_f\) and the result \(C\) are clamped to the range \([0, 1]\) before the fog blend is performed.

In color index mode, the formula for fog blending is

\[
I = i_r + (1 - f)i_f
\]

where \(i_r\) is the rasterized fragment’s color index and \(i_f\) is a single-precision floating-point value. \((1 - f)i_f\) is rounded to the nearest fixed-point value with the same number of bits to the right of the binary point as \(i_r\), and the integer portion of \(I\) is masked (bitwise ANDed) with \(2^n - 1\), where \(n\) is the number of bits in

Version 3.0 (with changes marked - August 11, 2008)
3.12. FRAGMENT SHADERS

a color in the color index buffer (buffers are discussed in chapter 4). The value of \( i_f \) is set by calling \texttt{Fog} with \texttt{pname} set to \texttt{FOG\_INDEX} and \texttt{param} being or \texttt{params} pointing to a single value for the fog index. The integer part of \( i_f \) is masked with \( 2^n - 1 \).

The state required for fog consists of a three valued integer to select the fog equation, three floating-point values \( d, e, \) and \( s \), an RGBA fog color and a fog color index, a two-valued integer to select the fog coordinate source, and a single bit to indicate whether or not fog is enabled. In the initial state, fog is disabled, \texttt{FOG\_COORD\_SRC} is \texttt{FRAGMENT\_DEPTH}, \texttt{FOG\_MODE} is \texttt{EXP}, \( d = 1.0, e = 1.0 \), and \( s = 0.0 \); \( C_f = (0, 0, 0, 0) \) and \( i_f = 0 \).

Fog has no effect if a fragment shader is active.

3.12 Fragment Shaders

The sequence of operations that are applied to fragments that result from rasterizing a point, line segment, polygon, pixel rectangle or bitmap as described in sections 3.9 through 3.11 is a fixed functionality method for processing such fragments. Applications can more generally describe the operations that occur on such fragments by using a \texttt{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 rasterizing a point, line segment, polygon, pixel rectangle or bitmap. The language used for fragment shaders is described in the OpenGL Shading Language Specification.

A fragment shader only applies when the GL is in RGBA mode. Its operation in color index mode is undefined.

Fragment shaders are created as described in section 2.20.1 using a \texttt{type} parameter of \texttt{FRAGMENT\_SHADER}. They are attached to and used in program objects as described in section 2.20.2.

When the program object currently in use includes a fragment shader, its fragment shader is considered \textit{active}, and is used to process fragments. If the program object has no fragment shader, or no program object is currently in use, the fixed-function fragment processing operations described in previous sections are used.

Results of rasterization are undefined if any of the selected draw buffers of the draw framebuffer have an integer format and no fragment shader is active.

3.12.1 Shader Variables

Fragment shaders can access uniforms belonging to the current shader object. The amount of storage available for fragment shader uniform variables is specified by
3.12. FRAGMENT SHADERS

the implementation dependent constant \texttt{MAX\_FRAGMENT\_UNIFORM\_COMPONENTS}. This value represents the number of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a fragment shader. A uniform matrix will 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 the data associated with a fragment that are used for fixed-function fragment processing, such as the fragment’s position, color, secondary color, texture coordinates, fog coordinate, and eye \(z\) coordinate.

Additionally, when a vertex shader is active, it may define one or more \textit{varying} variables (see section 2.20.3 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.

User-defined varying variables are not saved in the current raster position. When processing fragments generated by the rasterization of a pixel rectangle or bitmap, that values of user-defined varying variables are undefined. Built-in varying variables have well-defined values.

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 \textit{Shader Outputs} subsection of section 3.12.2 describes how to direct these values to buffers.

3.12.2 Shader Execution

If a fragment shader is active, the executable version of the fragment shader is used to process incoming fragment values that are the result of point, line segment, polygon, pixel rectangle or bitmap rasterization rather than the fixed-function fragment processing described in sections 3.9 through 3.11. In particular,

- The texture environments and texture functions described in section 3.9.13 are not applied.

- Texture application as described in section 3.9.17 is not applied.
• Color sum as described in section 3.10 is not applied.

• Fog as described in section 3.11 is not applied.

Texture Access

The Shader Only Texturing subsection of section 2.20.4 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to fragment shaders.

When a texture lookup is performed in a fragment shader, the GL computes the filtered texture value \( \tau \) in the manner described in sections 3.9.7 and 3.9.8, and converts it to a texture source color \( C_s \) according to table 3.23 (section 3.9.13). 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 derivates \( \frac{du}{dx} \), \( \frac{du}{dy} \), \( \frac{dv}{dx} \), \( \frac{dv}{dy} \), \( \frac{dw}{dx} \) 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 3.9.14) used to perform the lookup. The comparison operation is requested in the shader by using any of the shadow sampler types and in the texture using the \text{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 \text{DEPTH\_COMPONENT} or \text{DEPTH\_STENCIL}, and the \text{TEXTURE\_COMPARE\_MODE} is not \text{NONE}.

• The sampler used in a texture lookup function is one of the shadow sampler types, the texture object’s internal format is \text{DEPTH\_COMPONENT} or \text{DEPTH\_STENCIL}, and the \text{TEXTURE\_COMPARE\_MODE} is \text{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 \text{DEPTH\_COMPONENT} or \text{DEPTH\_STENCIL}.

The stencil index texture internal component is ignored if the base internal format is \text{DEPTH\_STENCIL}.

If a fragment shader uses a sampler whose associated texture object is not complete, as defined in section 3.9.10, the texture image unit will return \((R, G, B, A) = (0, 0, 0, 1)\).
3.12. FRAGMENT SHADERS

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 `gl_FragCoord` holds the window coordinates \(x, y, z,\) and \(\frac{1}{w}\) for the fragment. The \(z\) component of `gl_FragCoord` undergoes an implied conversion to floating-point. This conversion must leave the values 0 and 1 invariant. Note that this \(z\) component already has a polygon offset added in, if enabled (see section 3.6.5). The \(\frac{1}{w}\) value is computed from the \(w_c\) coordinate (see section 2.12), which is the result of the product of the projection matrix and the vertex’s eye coordinates.

The built-in variables `gl_Color` and `gl_SecondaryColor` hold the R, G, B, and A components, respectively, of the fragment color and secondary color. If the primary color or the secondary color components are represented by the GL as fixed-point values, they undergo an implied conversion to floating-point. This conversion must leave the values 0 and 1 invariant. Floating-point color components (resulting from a disabled vertex color clamp) are unmodified.

The built-in variable `gl_FrontFacing` is set to `TRUE` if the fragment is generated from a front facing primitive, and `FALSE` otherwise. For fragments generated from polygon, triangle, or quadrilateral primitives (including ones resulting from polygons rendered as points or lines), the determination is made by examining the sign of the area computed by equation 2.6 of section 2.19.1 (including the possible reversal of this sign controlled by `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.

The built-in variable `gl_PrimitiveID` is filled with the number of primitives processed by the rasterizer since the last time `Begin` was called (directly or indirectly via vertex array functions). The first primitive generated after a `Begin` is numbered zero, and the primitive ID counter is incremented after every individual point, line, or polygon primitive is processed. For polygons drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may be drawn. For `QUADS` and `QUAD_STRIP` primitives that are decomposed into triangles, the primitive ID is incremented after each complete quad is processed.

The value of `gl_PrimitiveID` is undefined for fragments generated by `POLYGON` primitives or from `DrawPixels` or `Bitmap` commands. Additionally,
3.12. FRAGMENT SHADERS

`gl_PrimitiveID` is only defined under the same conditions that `gl_VertexID` is defined, as described under “Shader Inputs” in section 2.20.4.

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 built-in variables. The built-in variables are `gl_FragColor`, `gl_FragData[n]`, and `gl_FragDepth`. If fragment color clamping is enabled and the color buffer has a fixed- or floating-point format, the final fragment color, fragment data, or varying out variable values written by a fragment shader are clamped to the range `[0, 1]` and are optionally converted to fixed-point as described in section 2.19.9. Only user-defined varying out variables declared as a floating-point type are clamped and may be converted. If fragment color clamping is disabled, or the color buffer has an integer format, the final fragment color, fragment data, or varying out variable values are not modified. 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 2.12.1). 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 a fixed-point format, color values are assumed to be floating-point and are converted to fixed-point as described in section 2.19.9; 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 `gl_FragColor`, `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 `gl_FragDepth` specifies the depth value for the fragment being processed. If the active fragment shader does not statically assign a value to `gl_FragDepth`, then the depth value generated during rasterization is used by sub-
sequent 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 assigned binding of a varying out variable would cause the GL to reference a non-existant fragment color number (one greater than or equal to \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}
int GetFragDataLocation( uint program, const char *name );
\end{verbatim}

returns the number of the fragment color to which the varying out variable \texttt{name} was bound when the program object \texttt{program} was last linked. \texttt{name} must be a null-terminated string. If \texttt{program} has not been successfully linked, the error \texttt{INVALID\_OPERATION} is generated. If \texttt{name} is not a varying out variable, or if an error occurs, -1 will be returned.
3.13 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. In RGBA mode, the value is multiplied by the fragment’s alpha (A) value to yield a final alpha value. In color index mode, the value is used to set the low order bits of the color index value as described in section 3.3. The coverage value is applied separately to each fragment color.

3.14 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. In RGBA mode, the fade factor is multiplied by the fragment’s alpha value to yield a final alpha value. In color index mode, the fade factor has no effect. The fade factor is applied separately to each fragment color.
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, stencil, and accumulation 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, the back right buffer, and some number of auxiliary buffers. 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.) The contents of the auxiliary buffers are never visible. All color buffers must have the same number of bitplanes, although an implementation or context may choose not to provide right buffers, back buffers, or auxiliary buffers at all. Further, an implementation or context may not provide depth, stencil, or accumulation buffers. If no default framebuffer is associated with the GL framebuffer_object, bug 3536. Removed discussion of height/width. Removed redundant use of “default window-system-provided” and “application created framebuffer object”. framebuffer_object framebuffer_object framebuffer_object framebuffer_object framebuffer_object
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 either a single unsigned integer color index or 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 unsigned fixed-point, floating-point, signed integer, 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 2.12.1 or a floating-point value. Each pixel in a stencil buffer consists of a single unsigned integer value. Each pixel in an accumulation buffer consists of up to four color components fixed-point values. If an accumulation buffer is present, it must have at least as many bitplanes per component as in the color buffers.

The number of bitplanes in the color, depth, stencil, and accumulation 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 4.4.2 describes the mechanism for controlling framebuffer usage.

The default framebuffer is initially used as the draw and read framebuffer, and the initial state of all provided bitplanes is undefined. The format and encoding of buffers in the draw and read framebuffers can be queried as described in section 6.1.3.

\footnote{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.}

Version 3.0 (with changes marked - August 11, 2008
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 modifications and tests.

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

Figure 4.1. Per-fragment operations.
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

```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 determined by the size of the GL window. 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`, and `SAMPLE_COVERAGE_INVERT`. 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.

Finally, 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}.

### 4.1.4 Alpha Test

This step applies only in RGBA mode, and only if the color buffer has a fixed-point or floating-point format. In color index mode, or if the color buffer has an integer format, proceed to the next operation.

The alpha test discards a fragment conditional on the outcome of a comparison between the incoming fragment’s alpha value and a constant value. If multiple colors are written by a fragment shader, the alpha value of fragment color zero is used to determine the result of the alpha test. The comparison is enabled or disabled with the generic \texttt{Enable} and \texttt{Disable} commands using the symbolic constant \texttt{ALPHA\_TEST}. When disabled, it is as if the comparison always passes. The test is controlled with

\begin{verbatim}
void AlphaFunc( enum func, clampf ref );
\end{verbatim}
4.1. PER-FRAGMENT OPERATIONS

func is a symbolic constant indicating the alpha test function; ref is a reference value. When performing the alpha test, the GL will convert the reference value to the same representation as the the fragment’s alpha value (floating-point or fixed-point). For fixed-point, the reference value is converted according to the rules given for an A component in section 2.19.9 and the fragment’s alpha value is rounded to the nearest integer.

The possible constants specifying the test function are NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GEQUAL, GREATER, or NOTEQUAL, meaning pass the fragment never, always, if the fragment’s alpha value is less than, less than or equal to, equal to, greater than or equal to, greater than, or not equal to the reference value, respectively.

The required state consists of the floating-point reference value, an eight-valued integer indicating the comparison function, and a bit indicating if the comparison is enabled or disabled. The initial state is for the reference value to be 0 and the function to be ALWAYS. Initially, the alpha test is disabled.

4.1.5 Stencil Test

The stencil test conditionally discards a fragment based on the outcome of a comparison between the value in the stencil buffer at location \((x_w, y_w)\) and a reference value. The test is enabled or disabled with the Enable and Disable commands, using the symbolic constant STENCIL_TEST. When disabled, the stencil test and associated modifications are not made, and the fragment is always passed.

The stencil test is controlled with

```c
void StencilFunc( enum func, int ref, uint mask );
void StencilFuncSeparate( enum face, enum func, int ref, uint mask );
void StencilOp( enum sfail, enum dpass, enum dppass );
void StencilOpSeparate( enum face, enum sfail, enum dppass, enum dppass );
```

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, lines, bitmaps, image rectangles) 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 sections 2.19.1 and 3.6.1).

Version 3.0 (with changes marked - August 11, 2008)
StencilFuncSeparate and StencilOpSeparate take a face argument which can be FRONT, BACK, or FRONT_AND_BACK and indicates which set of state is affected. StencilFunc and StencilOp set front and back stencil state to identical values.

StencilFunc and StencilFuncSeparate take three arguments that control whether the stencil test passes or fails. \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.

StencilOp and 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}. 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 4.1.6) fails (\texttt{dpfail}), or if it passes (\texttt{dppass}).

If the stencil test fails, the incoming fragment is discarded. The state required consists of the most recent values passed to StencilFunc or StencilFuncSeparate and to StencilOp or StencilOpSeparate, and a bit indicating whether stencil testing is enabled or disabled. In the initial state, stenciling is disabled, the front and back stencil reference value are both zero, the front and back stencil comparison functions are both \texttt{ALWAYS}, and the front and back stencil mask are both all ones. Initially, all three front and back stencil operations are \texttt{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.
4.1.6 Depth Buffer Test

The depth buffer test discards the incoming fragment if a depth comparison fails. The comparison is enabled or disabled with the generic Enable and Disable commands using the symbolic constant DEPTH_TEST. When disabled, the depth comparison and subsequent possible updates to the depth buffer value are bypassed and the fragment is passed to the next operation. The stencil value, however, is modified as indicated below as if the depth buffer test passed. If enabled, the comparison takes place and the depth buffer and stencil value may subsequently be modified.

The comparison is specified with

```c
void DepthFunc(enum func);
```

This command takes a single symbolic constant: one of NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GREATER, GEQUAL, NOTEQUAL. Accordingly, the depth buffer test passes never, always, if the incoming fragment’s \( z_w \) value is less than, less than or equal to, equal to, greater than, greater than or equal to, or not equal to the depth value stored at the location given by the incoming fragment’s \((x_w, y_w)\) coordinates.

If 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.

4.1.7 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.

### 4.1.8 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 is dependent on the incoming fragment’s alpha value and that of the corresponding currently stored pixel. Blending applies only in RGBA mode; and only if the color buffer has a fixed-point or floating-point format. In color index mode, or 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_BUFFERi 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 4.1.11), 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.

*Fixed-point* destination (framebuffer) components are taken to be fixed-point values represented according to the scheme in section 2.19.9 (Final Color Processing). 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 2.19.9.

Prior to blending, each fixed-point color component undergoes an implied conversion to floating-point. 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 3.9.15.

Version 3.0 (with changes marked - August 11, 2008)
Table 4.1: RGB and alpha blend equations.

If `FRAMEBUFFER_SRGB` is disabled or the value of `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` is not SRGB, no linearization is performed.

The resulting linearized R, G, and B and unmodified A values are recombined as the destination color used in blending computations.

Table 4.1 provides the corresponding per-component blend equations for each mode, whether acting on RGB components for `modeRGB` or the alpha component for `modeAlpha`.

In the table, the $s$ subscript on a color component abbreviation (R, G, B, or A) refers to the source color component for an incoming fragment, the $d$ subscript on a color component abbreviation refers to the destination color component at the corresponding framebuffer location, and the $c$ subscript on a color component abbreviation refers to the constant blend color component. A color component abbreviation without a subscript refers to the new color component resulting from blending. Additionally, $S_r$, $S_g$, $S_b$, and $S_a$ are the red, green, blue, and alpha components of the source weighting factors determined by the source blend function, and $D_r$, $D_g$, $D_b$, and $D_a$ are the red, green, blue, and alpha components of the destination weighting factors determined by the destination blend function. Blend functions are described below.
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.

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. PER-FRAGMENT OPERATIONS

<table>
<thead>
<tr>
<th>Function</th>
<th>RGB Blend Factors</th>
<th>Alpha Blend Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZERO</td>
<td>((0, 0, 0))</td>
<td>0</td>
</tr>
<tr>
<td>ONE</td>
<td>((1, 1, 1))</td>
<td>1</td>
</tr>
<tr>
<td>SRC_COLOR</td>
<td>((R_s, G_s, B_s))</td>
<td>(A_s)</td>
</tr>
<tr>
<td>ONE_MINUS_SRC_COLOR</td>
<td>((1, 1, 1) - (R_s, G_s, B_s))</td>
<td>(1 - A_s)</td>
</tr>
<tr>
<td>DST_COLOR</td>
<td>((R_d, G_d, B_d))</td>
<td>(A_d)</td>
</tr>
<tr>
<td>ONE_MINUS_DST_COLOR</td>
<td>((1, 1, 1) - (R_d, G_d, B_d))</td>
<td>(1 - A_d)</td>
</tr>
<tr>
<td>SRC_ALPHA</td>
<td>((A_s, A_s, A_s))</td>
<td>(A_s)</td>
</tr>
<tr>
<td>ONE_MINUS_SRC_ALPHA</td>
<td>((1, 1, 1) - (A_s, A_s, A_s))</td>
<td>(1 - A_s)</td>
</tr>
<tr>
<td>DST_ALPHA</td>
<td>((A_d, A_d, A_d))</td>
<td>(A_d)</td>
</tr>
<tr>
<td>ONE_MINUS_DST_ALPHA</td>
<td>((1, 1, 1) - (A_d, A_d, A_d))</td>
<td>(1 - A_d)</td>
</tr>
<tr>
<td>CONSTANT_COLOR</td>
<td>((R_c, G_c, B_c))</td>
<td>(A_c)</td>
</tr>
<tr>
<td>ONE_MINUS_CONSTANT_COLOR</td>
<td>((1, 1, 1) - (R_c, G_c, B_c))</td>
<td>(1 - A_c)</td>
</tr>
<tr>
<td>CONSTANT_ALPHA</td>
<td>((A_c, A_c, A_c))</td>
<td>(A_c)</td>
</tr>
<tr>
<td>ONE_MINUS_CONSTANT_ALPHA</td>
<td>((1, 1, 1) - (A_c, A_c, A_c))</td>
<td>(1 - A_c)</td>
</tr>
<tr>
<td>SRC_ALPHA_SATURATE</td>
<td>((f, f, f)^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. \(f = \min(A_s, 1 - A_d)\).
### 4.1.9 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

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

(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

\[c_s = c_l.\]

The resulting \(c_s\) values for R, G, and B, and the unmodified A form a new RGBA color value. If the color buffer is fixed-point, each component is clamped to the range \([0, 1]\) and then converted to a fixed-point value in the manner described in section 2.19.9. The resulting four values are sent to the subsequent dithering operation.

### 4.1.10 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. In RGBA mode 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.

In color index mode dithering selects either the largest representable index that is less than or equal to the incoming color value, \(c\), or the smallest representable index that is greater than or equal to \(c\). If one of the two indices 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 color index or 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 $c$; a color index is rounded to the nearest representable index value.

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.

### 4.1.11 Logical Operation

Finally, a logical operation is applied between the incoming fragment’s color or index values and the color or index values stored at the corresponding location in the framebuffer. The result replaces the values in the framebuffer at the fragment’s $(x_w, y_w)$ coordinates. If the selected draw buffers refer to the same framebuffer-attachable image more than once, then the values stored in that image are undefined.

The logical operation on color indices is enabled or disabled with `Enable` or `Disable` using the symbolic constant `INDEX_LOGIC_OP`. (For compatibility with GL version 1.0, the symbolic constant `LOGIC_OP` may also be used.) 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.

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 color index buffer that is selected for writing, or for each red, green, blue, and alpha value of each...
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 \landor 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 \landor 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.

color buffer that is selected for writing. The required state is an integer indicating the logical operation, and two bits 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.

4.1.12 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 alpha test, 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 alpha, 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, and dithering operations are performed for a pixel sample only if that sample’s fragment coverage bit is a value of 1. If the corre-
4.2 WHOLE FRAMEBUFFER OPERATIONS

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 alpha, 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, though a simple average computed independently for each color component is recommended.

4.2 Whole Framebuffer Operations

The preceding sections described the operations that occur as individual fragments are sent to the framebuffer. This section describes operations that control or affect the whole framebuffer.

4.2.1 Selecting 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 values for `buf` depend on whether the GL is using the default framebuffer (i.e., `DRAW_FRAMEBUFFER_BINDING` is zero), or a framebuffer object (i.e.,

Version 3.0 (with changes marked - August 11, 2008)
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 section 4.4.

If the GL is bound to the default framebuffer, then buf must be one 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), and to the auxiliary buffers. Arguments other than AUXi that omit reference to LEFT or RIGHT refer to both left and right buffers. Arguments other than AUXi that omit reference to FRONT or BACK refer to both front and back buffers. AUXi enables drawing only to auxiliary buffer i. Each AUXi adheres to AUXi = AUX0 + i, and i must be in the range 0 to the value of AUX_BUFFERS minus one.

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_ATTACHMENTi enables drawing only to the image attached to the framebuffer at COLOR_ATTACHMENTi. Each COLOR_ATTACHMENTi adheres to COLOR_ATTACHMENTi = COLOR_ATTACHMENT0 + i. The initial value of DRAW_BUFFER for framebuffer objects is COLOR_ATTACHMENT0.

If the GL is bound to 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.

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

Version 3.0 (with changes marked - August 11, 2008
4.2. WHOLE FRAMEBUFFER OPERATIONS

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>Front Left</th>
<th>Front Right</th>
<th>Back Left</th>
<th>Back Right</th>
<th>Aux (i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FRONT_LEFT</td>
<td>(\bullet)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FRONT_RIGHT</td>
<td></td>
<td>(\bullet)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BACK_LEFT</td>
<td></td>
<td></td>
<td>(\bullet)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BACK_RIGHT</td>
<td></td>
<td></td>
<td></td>
<td>(\bullet)</td>
<td></td>
</tr>
<tr>
<td>FRONT</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BACK</td>
<td></td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEFT</td>
<td>(\bullet)</td>
<td></td>
<td>(\bullet)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RIGHT</td>
<td></td>
<td>(\bullet)</td>
<td></td>
<td>(\bullet)</td>
<td></td>
</tr>
<tr>
<td>FRONT_AND_BACK</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td></td>
</tr>
<tr>
<td>AUX(i)</td>
<td></td>
<td></td>
<td>(\bullet)</td>
<td>(\bullet)</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Arguments to DrawBuffer(s) and ReadBuffer when the context is bound to a framebuffer object, and the buffers they indicate. \(i\) in COLOR_ATTACHMENT\(i\) may range from zero to the value of MAX_COLOR_ATTACHMENTS - 1.

<table>
<thead>
<tr>
<th>Symbolic Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>No buffer</td>
</tr>
<tr>
<td>COLOR_ATTACHMENT(i) (see caption)</td>
<td>Output fragment color to image attached at color attachment point (i)</td>
</tr>
</tbody>
</table>

Version 3.0 (with changes marked - August 11, 2008)
4.2. WHOLE FRAMEBUFFER OPERATIONS

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

<table>
<thead>
<tr>
<th>Symbolic Constant</th>
<th>Front Left</th>
<th>Front Right</th>
<th>Back Left</th>
<th>Back Right</th>
<th>Aux $i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FRONT_LEFT</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FRONT_RIGHT</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BACK_LEFT</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BACK_RIGHT</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>AUX$i$</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
</tbody>
</table>

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 a 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 and must be at least 1. The number of draw buffers supported can be queried by calling GetInteger with the symbolic constant MAX_DRAW_BUFFERS. An INVALID_VALUE error is generated if $n$ is greater than MAX_DRAW_BUFFERS.

Except for NONE, a buffer may not appear more then once in the array pointed to by $bufs$. Specifying a buffer more then once will result in the error INVALID_OPERATION.

If fixed-function fragment shading is being performed, DrawBuffers specifies a set of draw buffers into which the fragment color is written.

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-
4.2. WHOLE FRAMEBUFFER OPERATIONS

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_ATTACHMENTm where m is greater than or equal
to the value of MAX_COLOR_ATTACHMENTS, then the error INVALID_OPERATION results.

Indicating a buffer or buffers using DrawBuffer or DrawBuffers causes sub-
sequent pixel color value writes to affect the indicated buffers.

Specifying NONE as the draw buffer for a fragment color will inhibit that frag-
ment color from being written to any buffer.

Monoscopic contexts include only left buffers, while stereoscopic contexts in-
clude 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 FRONT if there are no
back buffers; otherwise it is BACK. 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
calling GLinterv with the symbolic constant DRAW_BUFFERi. DRAW_BUFFER is
equivalent to DRAW_BUFFER0.

4.2.2 Fine Control of Buffer Updates

Writing of bits to each of the logical framebuffers after all per-fragment operations
have been performed may be masked. The commands

void IndexMask(uint mask);

Version 3.0 (with changes marked - August 11, 2008
4.2. WHOLE FRAMEBUFFER OPERATIONS

```c
void ColorMask( boolean r, boolean g, boolean b, boolean a );
void ColorMaski( uint buf, boolean r, boolean g, boolean b, boolean a );
```

control writes to the active draw buffers.

The least significant \( n \) bits of \( mask \), where \( n \) is the number of bits in a color index buffer, specify a mask. Where a 1 appears in this mask, the corresponding bit in the color index buffer (or buffers) is written; where a 0 appears, the bit is not written. This mask applies only in color index mode.

In RGBA mode, \texttt{ColorMask} and \texttt{ColorMaski} are used to mask the writing of R, G, B and A values to the draw buffer or buffers. \texttt{ColorMaski} sets the mask for a particular draw buffer. The mask for \texttt{DRAW_BUFFER} \( i \) is modified by passing \( i \) as the parameter \( buf \). \( r \), \( g \), \( b \), and \( a \) indicate whether R, G, B, or A values, respectively, are written or not (a value of \texttt{TRUE} means that the corresponding value is written).

The mask specified by \( r \), \( g \), \( b \), and \( a \) is applied to the color buffer associated with \texttt{DRAW_BUFFER} \( i \). If \texttt{DRAW_BUFFER} \( i \) is one of \texttt{FRONT}, \texttt{BACK}, \texttt{LEFT}, \texttt{RIGHT}, or \texttt{FRONT_AND_BACK} (specifying multiple color buffers) then the mask is applied to all of the buffers.

\texttt{ColorMask} sets the mask for all draw buffers to the same values as specified by \( r \), \( g \), \( b \), and \( a \).

An \texttt{INVALID_VALUE} error is generated if \( index \) is greater than the value of \texttt{MAX_DRAW_BUFFERS} minus one.

In the initial state, all bits (in color index mode) and all color values (in RGBA mode) are enabled for writing for all draw buffers. The value of the color writemask for draw buffer \( i \) can be queried by calling \texttt{GetBooleaniv} with \texttt{target} \texttt{COLOR_WRITEMASK} and \texttt{index} \( i \). The value of the color writemask for draw buffer zero may also be queried by calling \texttt{GetBooleaniv} with \texttt{value} \texttt{COLOR_WRITEMASK}.

The depth buffer can be enabled or disabled for writing \( z_w \) values using

```c
void DepthMask( boolean mask );
```

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

```c
void StencilMask( uint mask );
void StencilMaskSeparate( enum face, uint mask );
```

Version 3.0 (with changes marked - August 11, 2008)
control the writing of particular bits into the stencil planes.

The least significant $s$ bits of mask comprise an integer mask ($s$ is the number of bits in the stencil buffer), just as for IndexMask. 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 4.1.5). The clear operation always uses the front stencil write mask when clearing the stencil buffer.

The state required for the various masking operations is three integers and a bit: an integer for color indices, an integer 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

```c
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 ACCUM_BUFFER_BIT, indicating the buffers currently enabled for color writing, the depth buffer, the stencil buffer, and the accumulation 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 in RGBA mode. The specified components are stored as floating-point values.

The command

void **ClearIndex**( `float index` );

sets the clear color index. `index` is converted to a fixed-point value with unspecified precision to the left of the binary point; the integer part of this value is then masked with $2^m - 1$, where `m` is the number of bits in a color index value stored in the framebuffer.

The command

void **ClearDepth**( `clampd 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 2.12.1. 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.

The command

void **ClearAccum**( `float r`, `float g`, `float b`, `float a` );

takes four floating-point arguments that are the values, in order, to which to set the R, G, B, and A values of the accumulation buffer (see the next section). These values are clamped to the range $[-1, 1]$ when they are specified.

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. Fixed-point RGBA color buffers are cleared to color values derived by clamping each component of the clear color to $[0, 1]$ and converting to fixed-point according to the rules of section 2.19.9. The result of clearing integer color buffers is undefined.

Version 3.0 (with changes marked - August 11, 2008
4.2. WHOLE FRAMEBUFFER OPERATIONS

The state required for clearing is a clear value for each of the color buffer, the depth buffer, the stencil buffer, and the accumulation buffer. Initially, the RGBA color clear value is $(0, 0, 0, 0)$, the clear color index is 0, and the stencil buffer and accumulation buffer clear values are all 0. The depth buffer clear value is initially 1.0.

Individual buffers of the currently bound draw framebuffer may be cleared with the command

```
ClearBuffer{ if ui }v( enum buffer, const T *value );
```

where `buffer` identifies a buffer to clear, and `value` specifies the value or values to clear it to. If the default framebuffer is bound to the draw framebuffer, `buffer` must be one of `FRONT_LEFT`, `FRONT_RIGHT`, `BACK_LEFT`, `BACK_RIGHT`, `AUX`, `DEPTH_BUFFER`, or `STENCIL_BUFFER`, identifying a color buffer, the depth buffer, or the stencil buffer. If a framebuffer object is bound to the draw framebuffer, `buffer` may be one of the framebuffer attachment points listed in table 4.12.

If `buffer` identifies an RGBA color buffer, `value` points to a four-element vector specifying the R, G, B, and A color to clear the buffer to. The `ClearBufferfv`, `ClearBufferiv`, and `ClearBufferuiv` commands should be used to clear fixed- and floating-point, signed integer, and unsigned integer color buffers respectively. Clamping and conversion for fixed-point color buffers are performed in the same fashion as `ClearColor`.

If `buffer` identifies a depth buffer (`DEPTH_BUFFER` or `DEPTH_ATTACHMENT`), `value` points to the single depth value to clear the buffer to. Clamping and type conversion for fixed-point depth buffers are performed in the same fashion as `ClearDepth`. Only `ClearBufferfv` should be used to clear depth buffers.

If `buffer` identifies a stencil buffer (`STENCIL_BUFFER` or `STENCIL_ATTACHMENT`), `value` points to the single stencil value to clear the buffer to. Masking and type conversion are performed in the same fashion as `ClearStencil`. Only `ClearBufferiv` should be used to clear stencil buffers.

The command

```
void ClearBufferfi( enum buffer, float depth, int stencil );
```

clears both depth and stencil buffers of the currently bound draw framebuffer. `buffer` must be `DEPTH_STENCIL_ATTACHMENT`, `depth` and `stencil` are the values to clear the depth and stencil buffers to, respectively. Clamping and type conversion of `depth` for fixed-point depth buffers is performed in the same fashion as `ClearDepth`. Masking of `stencil` for stencil buffers is performed in the same fashion as `ClearStencil`. `ClearBufferfi` is equivalent to clearing the depth

---

Version 3.0 (with changes marked - August 11, 2008)
4.2. WHOLE FRAMEBUFFER OPERATIONS

and stencil buffers separately, but may be faster when a buffer of internal format
DEPTH_STENCIL is being cleared.

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

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

Errors
ClearBuffer generates an INVALID_ENUM error if buffer does not identify a
valid buffer in or attachment of the currently bound draw framebuffer, or if buffer
is not one of the buffer names specified as valid for that form of ClearBuffer
command.

ClearBuffer generates an INVALID_OPERATION error if buffer identifies a
color index buffer.

Clearing the Multisample Buffer

The color samples of the multisample buffer are cleared when one or more color
buffers are cleared, as specified by the Clear mask bit COLOR_BUFFER_BIT and
the DrawBuffer mode. If the DrawBuffer mode is NONE, the color samples of the
multisample buffer cannot be cleared using Clear.

If the Clear mask bits DEPTH_BUFFER_BIT or STENCIL_BUFFER_BIT are set,
then the corresponding depth or stencil samples, respectively, are cleared.

The ClearBuffer commands also clear color, depth, or stencil samples of mul-
tisample buffers corresponding to the specified buffer.

4.2.4 The Accumulation Buffer

Each portion of a pixel in the accumulation buffer consists of four values: one for
each of R, G, B, and A. The accumulation buffer is controlled exclusively through
the use of

    void Accum( enum op, float value );

(except for clearing it). op is a symbolic constant indicating an accumulation buffer
operation, and value is a floating-point value to be used in that operation. The
possible operations are ACCUM, LOAD, RETURN, MULT, and ADD.

When the scissor test is enabled (section 4.1.2), then only those pixels within
the current scissor box are updated by any Accum operation; otherwise, all pixels
in the window are updated. The accumulation buffer operations apply identically

Version 3.0 (with changes marked - August 11, 2008
4.3. DRAWING, READING, AND COPYING PIXELS

Pixels may be written to and read from the framebuffer using the DrawPixels and ReadPixels commands. CopyPixels can be used to copy a block of pixels from...
4.3. DRAWING, READING, AND COPYING PIXELS

one portion of the framebuffer to another.

4.3.1 Writing to the Stencil or Depth/Stencil Buffers

The operation of DrawPixels was described in section 3.7.4, except if the format argument was STENCIL_INDEX or DEPTH_STENCIL. In this case, all operations described for DrawPixels take place, but window \((x, y)\) coordinates, each with the corresponding stencil index, or depth value and stencil index, are produced in lieu of fragments. Each coordinate-data pair is sent directly to the per-fragment operations, bypassing the texture, fog, and antialiasing application stages of rasterization. Each pair is then treated as a fragment for purposes of the pixel ownership and scissor tests; all other per-fragment operations are bypassed. Finally, each stencil index is written to its indicated location in the framebuffer, subject to the current front stencil mask (set with StencilMask or StencilMaskSeparate). If a depth component is present, and the setting of DepthMask is not FALSE, it is also written to the framebuffer; the setting of DepthTest is ignored.

The error INVALID_OPERATION results if the format argument is STENCIL_INDEX and there is no stencil buffer, or if format is DEPTH_STENCIL and there is not both a depth buffer and a stencil buffer.

4.3.2 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 pixels 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 correspond to those of DrawPixels. 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 frame-
Figure 4.2. Operation of ReadPixels. Operations in dashed boxes may be enabled or disabled, except in the case of “convert RGB to L”, which is only applied when reading color data in luminosity formats. RGBA and color index pixel paths are shown; depth and stencil pixel paths are not shown.
4.3. DRAWING, READING, AND COPYING PIXELS

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type</th>
<th>Initial Value</th>
<th>Valid Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>PACK_SWAP_BYTES</td>
<td>boolean</td>
<td>FALSE</td>
<td>TRUE/FALSE</td>
</tr>
<tr>
<td>PACK_LSB_FIRST</td>
<td>boolean</td>
<td>FALSE</td>
<td>TRUE/FALSE</td>
</tr>
<tr>
<td>PACK_ROW_LENGTH</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>PACK_SKIP_ROWS</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>PACK_SKIP_PIXELS</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>PACK_ALIGNMENT</td>
<td>integer</td>
<td>4</td>
<td>1, 2, 4, 8</td>
</tr>
<tr>
<td>PACK_IMAGE_HEIGHT</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
<tr>
<td>PACK_SKIP_IMAGES</td>
<td>integer</td>
<td>0</td>
<td>(0, ∞)</td>
</tr>
</tbody>
</table>

Table 4.7: PixelStore parameters pertaining to ReadPixels, GetColorTable, GetConvolutionFilter, GetSeparableFilter, GetHistogram, GetMinmax, GetPolygonStipple, and GetTexImage.

buffer is framebuffer complete, and the value of SAMPLE_BUFFERS for the read framebuffer is greater than zero.

Obtaining Pixels from the Framebuffer

If the format is DEPTH_COMPONENT, then values are obtained from the depth buffer. If there is no depth buffer, the error INVALID_OPERATION occurs.

If there is a multisample buffer (the value of SAMPLE_BUFFERS is one), then values are obtained from the depth samples in this buffer. It is recommended that the depth value of the centermost sample be used, though implementations may choose any function of the depth sample values at each pixel.

If the format is DEPTH_STENCIL, then values are taken from both the depth buffer and the stencil buffer. If there is no depth buffer or if there is no stencil buffer, then the error INVALID_OPERATION occurs. If the type parameter is not UNSIGNED_INT_24_8 or FLOAT_32_UNSIGNED_INT_24_8_REV, then the error depth_buffer_float 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 sam-
4.3. DRAWING, READING, AND COPYING PIXELS

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.

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_ATTACHMENTi enables reading from the image attached to the framebuffer at COLOR_ATTACHMENTi. 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 \(0 \leq i < width\) and \(0 \leq j < height\); 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.

Version 3.0 (with changes marked - August 11, 2008)
Otherwise, ReadPixels obtains values from the selected buffer, regardless of how those values were placed there.

If the GL is in RGBA mode, and format is one of RED, GREEN, BLUE, ALPHA, RG, RGBA, BGR, BGRA, LUMINANCE, or LUMINANCE, ALPHA, 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 COLOR_INDEX and the GL is in RGBA mode then the error INVALID OPERATION occurs. If the GL is in color index mode, and format is not DEPTH_COMPONENT, DEPTH_STENCIL, or STENCIL_INDEX, then the color index is obtained at each pixel location.

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 3.15). If G, B, or A values are not present in the internal format, they are taken to be zero, zero, and one respectively.

**Conversion of RGBA values**

This step applies only if the GL is in RGBA mode, and then 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 fixed-point color buffer, each element is taken to be a fixed-point value in \([0, 1]\) with \(m\) bits, where \(m\) is the number of bits in the corresponding color component of the selected buffer (see section 2.19.9). 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 2.12.1). No conversion is necessary if the depth buffer uses a floating-point representation.
4.3. DRAWING, READING, AND COPYING PIXELS

Pixel Transfer Operations

This step is actually the sequence of steps that was described separately in section 3.7.5. After the processing described in that section is completed, groups are processed as described in the following sections.

Conversion to L

This step applies only to RGBA component groups. If the format is either LUMINANCE or LUMINANCE_ALPHA, a value L is computed as

\[ L = R + G + B \]

where R, G, and B are the values of the R, G, and B components. The single computed L component replaces the R, G, and B components in the group.

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.

For a floating-point RGBA color, if type is not one of FLOAT, UNSIGNED_INT_5_9_9_9_REV, or UNSIGNED_INT_10F_11F_11F_REV; if CLAMP_READ_COLOR is TRUE; or if CLAMP_READ_COLOR is FIXED_ONLY and the selected color buffer is a fixed-point buffer, 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 RGBA, 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 3.11.

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 \(red_s\), \(green_s\), \(blue_s\), and \(exp\_shared\) integers as described in section 3.9.1 when internalformat is RGB9_E5. The \(red_s\), \(green_s\),
4.3. DRAWING, READING, AND COPYING PIXELS

<table>
<thead>
<tr>
<th>type Parameter</th>
<th>Index Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNSIGNED_BYTE</td>
<td>$2^8 - 1$</td>
</tr>
<tr>
<td>BITMAP</td>
<td>1</td>
</tr>
<tr>
<td>BYTE</td>
<td>$2^7 - 1$</td>
</tr>
<tr>
<td>UNSIGNED_SHORT</td>
<td>$2^{15} - 1$</td>
</tr>
<tr>
<td>SHORT</td>
<td>$2^{15} - 1$</td>
</tr>
<tr>
<td>UNSIGNED_INT</td>
<td>$2^{31} - 1$</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>

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

$ blues$, 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 3.11.

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 3.5 for the type parameter, an INVALID_OPERATION error results.

Groups of elements are placed in memory just as they are taken from memory for DrawPixels. That is, the $i$th group of the $j$th row (corresponding to the $i$th pixel in the $j$th row) is placed in memory just where the $i$th group of the $j$th row would be taken from for DrawPixels. See Unpacking under section 3.7.4. 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, ALPHA, or LUMINANCE, only the corresponding single element is written. Likewise if the format is RG, LUMINANCE_ALPHA, RGB, or BGR, only the

Version 3.0 (with changes marked - August 11, 2008
### 4.3. DRAWING, 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 = \frac{(2^8 - 1)f}{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)f}{2} )</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)f}{2} )</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_4.4.4.4</td>
<td>ushort</td>
<td>( c = (2^N - 1)f )</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_4.4.4.4_REV</td>
<td>ushort</td>
<td>( c = (2^N - 1)f )</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_5.5.5.1</td>
<td>ushort</td>
<td>( c = (2^N - 1)f )</td>
</tr>
<tr>
<td>UNSIGNED_SHORT_1.5.5.5_REV</td>
<td>ushort</td>
<td>( c = (2^N - 1)f )</td>
</tr>
<tr>
<td>UNSIGNED_INT_8.8.8</td>
<td>uint</td>
<td>( c = (2^N - 1)f )</td>
</tr>
<tr>
<td>UNSIGNED_INT_8.8.8_REV</td>
<td>uint</td>
<td>( c = (2^N - 1)f )</td>
</tr>
<tr>
<td>UNSIGNED_INT_10.10.10_2</td>
<td>uint</td>
<td>( c = (2^N - 1)f )</td>
</tr>
<tr>
<td>UNSIGNED_INT_2.10.10.10_REV</td>
<td>uint</td>
<td>( c = (2^N - 1)f )</td>
</tr>
<tr>
<td>UNSIGNED_INT_24.8</td>
<td>uint</td>
<td>( c = (2^N - 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.

Version 3.0 (with changes marked - August 11, 2008)
corresponding two or three elements are written. Otherwise all the elements of each group are written.

### 4.3.3 Copying Pixels

The command

```c
void CopyPixels(int x, int y,sizei width,sizei height,
    enum type);
```

transfers a rectangle of pixel values from one region of the read framebuffer to another in the draw framebuffer. Pixel copying is diagrammed in figure 4.3.

- `type` is a symbolic constant that must be one of `COLOR`, `STENCIL`, `DEPTH`, or `DEPTH_STENCIL`, indicating that the values to be transferred are colors, stencil values, depth values, or depth/stencil values, respectively. The first four arguments have the same interpretation as the corresponding arguments to `ReadPixels`.

Values are obtained from the framebuffer, converted (if appropriate), then subjected to the pixel transfer operations described in section 3.7.5, just as if `ReadPixels` were called with the corresponding arguments. If the `type` is `STENCIL` or `DEPTH`, then it is as if the `format` for `ReadPixels` were `STENCIL_INDEX` or `DEPTH_COMPONENT`, respectively. If the `type` is `DEPTH_STENCIL`, then it is as if the `format` for `ReadPixels` were specified as described in table 4.10. If the `type` is `COLOR`, then if the GL is in RGBA mode, it is as if the `format` were `RGBA`, while if the GL is in color index mode, it is as if the `format` were `COLOR_INDEX`.

The groups of elements so obtained are then written to the framebuffer just as if `DrawPixels` had been given `width` and `height`, beginning with final conversion of elements. The effective `format` is the same as that already described.

Finally, the behavior of several GL operations is specified as if the arguments were passed to `CopyPixels`. These operations include `CopyTexImage*`, `CopyTexSubImage*`, `CopyColorTable`, `CopyColorSubTable`, and `CopyConvolutionFilter*`. An `INVALID_FRAMEBUFFER_OPERATION` error will be generated if an attempt is made to execute one of these operations, or `CopyPixels`, while 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.

`CopyPixels` will generate an `INVALID_FRAMEBUFFER_OPERATION` error if the object bound to `DRAW_FRAMEBUFFER_BINDING` (see section 4.4) is not framebuffer complete.

If the read buffer contains integer or unsigned integer components, an `INVALID_OPERATION` error is generated.

Bug 3295

Version 3.0 (with changes marked - August 11, 2008)
4.3. DRAWING, READING, AND COPYING PIXELS

Figure 4.3. Operation of CopyPixels. Operations in dashed boxes may be enabled or disabled. Index-to-RGBA lookup is currently never performed. RGBA and color index pixel paths are shown; depth and stencil pixel paths are not shown.

Version 3.0 (with changes marked - August 11, 2008)
### 4.3. DRAWING, READING, AND COPYING PIXELS

<table>
<thead>
<tr>
<th>DEPTH_BITS</th>
<th>STENCIL_BITS</th>
<th>format</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>zero</td>
<td>DEPTH_STENCIL</td>
</tr>
<tr>
<td>zero</td>
<td>non-zero</td>
<td>DEPTH_COMPONENT</td>
</tr>
<tr>
<td>non-zero</td>
<td>zero</td>
<td>STENCIL_INDEX</td>
</tr>
<tr>
<td>non-zero</td>
<td>non-zero</td>
<td>DEPTH_STENCIL</td>
</tr>
</tbody>
</table>

Table 4.10: Effective **ReadPixels** format for **DEPTH_STENCIL** **CopyPixels** operation.

#### Blitting Pixel Rectangles

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. There are some important distinctions from **CopyPixels**, as described below.

*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, just as with **CopyPixels**.

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.

Version 3.0 (with changes marked - August 11, 2008)
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 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 as in **CopyPixels**. However, no pixel transfer operations are applied, and clamping behaves as if **CLAMP_FRAGMENT_COLOR** is set to **FIXED_ONLY**. Format conversion is not supported for all data types. If the read buffer contains floating-point values and any draw buffer does not contain floating-point values, or if the read buffer contains non-floating-point values and any draw buffer contains floating-point values, an **INVALID_OPERATION** error is generated.

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.

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
4.4. FRAMEBUFFER OBJECTS

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 gen-
erated if the dimensions of the source and destination rectangles provided to Blit-
Framebuffer 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 (X0, Y0) and
(X1, Y1) bounds.

4.3.4 Pixel Draw/Read State

The state required for pixel operations consists of the parameters that are set with
PixelStore, PixelTransfer, and PixelMap. This state has been summarized in
tables 3.1, 3.2, and 3.3. Additional state includes the current raster position (sec-
tion 2.18), an integer indicating the current setting of ReadBuffer, and a three-
valued integer controlling clamping during Final Conversion. The initial value of
the read buffer is FRONT if there is no back buffer and BACK otherwise, and the
initial value of read color clamping is FIXED_ONLY. State set with PixelStore is
GL client state.

4.4 Framebuffer Objects

As described in chapter 1 and section 2.1, the GL renders into (and reads values
from) a framebuffer. GL defines two classes of framebuffers: window system-
provided and application-created.

Initially, the GL uses the default framebuffer. The storage, dimensions, allo-
cation, and format of the images attached to this framebuffer are managed entirely

Version 3.0 (with changes marked - August 11, 2008
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 (accumulation and auxiliary buffers are not allowed). For each logical buffer, a framebuffer-attachable image can be attached to the framebuffer to store the rendered output for that logical buffer. Examples of framebuffer-attachable images include texture images and renderbuffer images. Renderbuffers are described further in section 4.4.2

By allowing the images of a renderbuffer to be attached to a framebuffer, the GL provides a mechanism to support off-screen rendering. Further, by allowing the images of a texture to be attached to a framebuffer, the GL provides a mechanism to support render to texture.

### 4.4.1 Binding and Managing Framebuffer Objects

The default framebuffer for rendering and readback operations is provided by the window system. In addition, named framebuffer objects can be created and operated upon. The 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 GenFramebuffers (see below) to DRAW_FRAMEBUFFER or READ_FRAMEBUFFER. The binding is effected by calling

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

with `target` set to the desired framebuffer target and `framebuffer` set to the framebuffer object name. The resulting framebuffer object is a new state vector, comprising all the state values listed in table 6.28, as well as one set of the state values listed in table 6.29 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 6.51 and 6.31 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,

- 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`.

There are no accumulation buffer bitplanes, so the value of the implementation-dependent state variables `ACCUM_RED_BITS`, `ACCUM_GREEN_BITS`, `ACCUM_BLUE_BITS`, and `ACCUM_ALPHA_BITS` are all zero.

There are no `AUX` buffer bitplanes, so the value of the implementation-dependent state variable `AUX_BUFFERS` is zero.

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 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 `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 `GetIntegerv` with the symbolic constants `DRAW_FRAMEBUFFER_BINDING` and `READ_FRAMEBUFFER_BINDING`, respectively. `FRAMEBUFFER_BINDING` is equivalent to `DRAW_FRAMEBUFFER_BINDING`.

---

Version 3.0 (with changes marked - August 11, 2008)
4.4. FRAMEBUFFER OBJECTS

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 6.29.

There are two types of framebuffer-attachable images: the image of a renderbuffer object, and an image of a texture object.

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 6.31. 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 \texttt{RENDERBUFFER}. There is no renderbuffer object corresponding to the name zero, so client attempts to modify or query renderbuffer state for the target \texttt{RENDERBUFFER} while zero is bound will generate GL errors, as described in section 6.1.3.

The current \texttt{RENDERBUFFER} binding can be determined by calling \texttt{GetIntegerv} with the symbolic constant \texttt{RENDERBUFFER\_BINDING}.

\texttt{BindRenderbuffer} fails and an \texttt{INVALID\_OPERATION} error is generated if \texttt{renderbuffer} is not a name returned from a previous call to \texttt{GenRenderbuffers}, or if such a name has since been deleted with \texttt{DeleteRenderbuffers}.

Renderbuffer objects are deleted by calling

\begin{verbatim}
void DeleteRenderbuffers( sizei n, const uint *renderbuffers );
\end{verbatim}

where \texttt{renderbuffers} contains \texttt{n} names of renderbuffer objects to be deleted. After a renderbuffer object is deleted, it has no contents, and its name is again unused. If a renderbuffer that is currently bound to \texttt{RENDERBUFFER} is deleted, it is as though \texttt{BindRenderbuffer} had been executed with the \texttt{target} \texttt{RENDERBUFFER} and \texttt{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 \texttt{renderbuffers} are silently ignored, as is the value zero.

The command

\begin{verbatim}
void GenRenderbuffers( sizei n, uint *renderbuffers );
\end{verbatim}

returns \texttt{n} previously unused renderbuffer object names in \texttt{renderbuffers}. These names are marked as used, for the purposes of \texttt{GenRenderbuffers} only, but they acquire renderbuffer state only when they are first bound, just as if they were unused.

The command

\begin{verbatim}
void RenderbufferStorageMultisample( enum target,  
   sizei samples, enum internalformat, sizei width,  
   sizei height );
\end{verbatim}

establishes the data storage, format, dimensions, and number of samples of a renderbuffer object’s image. \texttt{target} must be \texttt{RENDERBUFFER}, \texttt{internalformat} must be color-renderable, depth-renderable, or stencil-renderable (as defined in section 4.4.4). \texttt{width} and \texttt{height} are the dimensions in pixels of the renderbuffer. If either \texttt{width} or \texttt{height} is greater than \texttt{MAX\_RENDERBUFFER\_SIZE}, or if \texttt{samples} is greater than \texttt{MAX\_SAMPLES}, then the error \texttt{INVALID\_VALUE} is generated. If the GL
4.4. **FRAMEBUFFER OBJECTS**

<table>
<thead>
<tr>
<th>Sized Internal Format</th>
<th>Base Internal Format</th>
<th>$S$ bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>STENCIL_INDEX1</td>
<td>STENCIL_INDEX</td>
<td>1</td>
</tr>
<tr>
<td>STENCIL_INDEX4</td>
<td>STENCIL_INDEX</td>
<td>4</td>
</tr>
<tr>
<td>STENCIL_INDEX8</td>
<td>STENCIL_INDEX</td>
<td>8</td>
</tr>
<tr>
<td>STENCIL_INDEX16</td>
<td>STENCIL_INDEX</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 4.11: Correspondence of sized internal formats to base internal formats for formats that can be used only with renderbuffers.

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 implementation dependent. However, the resulting value for `RENDERBUFFER_SAMPLES` is guaranteed to be greater than or equal to `samples` and no more than the next larger sample count supported by the implementation.

A GL implementation may vary its allocation of internal component resolution based on any `RenderbufferStorage` parameter (except target), but the allocation and chosen internal format must not be a function of any other state and cannot be changed once they are established.

The command

```c
void RenderbufferStorage( enum target, enum internalformat,
                        sizei width,sizei height);
```

is equivalent to calling `RenderbufferStorageMultisample` with `samples` equal to zero.

**Required Renderbuffer Formats**

Implementations are required to support the same internal formats for renderbuffers as the required formats for textures enumerated in section 3.9.1, with the exception of `MAX_SAMPLES`.

Version 3.0 (with changes marked - August 11, 2008)
4.4. FRAMEBUFFER OBJECTS

of the color formats labelled “texture-only”. Requesting one of these internal formats for a renderbuffer will allocate exactly the internal component sizes and types shown for that format in tables 3.16-3.18.

Implementations must support creation of renderbuffers in these required formats with up to the value of \texttt{MAX\_SAMPLES} multisamples.

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

\begin{verbatim}
void FramebufferRenderbuffer( enum target,
    enum attachment, enum renderbuffertarget,
    uint renderbuffer);
\end{verbatim}

\texttt{target} must be \texttt{DRAW\_FRAMEBUFFER}, \texttt{READ\_FRAMEBUFFER}, or \texttt{FRAMEBUFFER}. Bug 3572 \texttt{FRAMEBUFFER} is equivalent to \texttt{DRAW\_FRAMEBUFFER}. An \texttt{INVALID\_OPERATION} error is generated if the value of the corresponding binding is zero. \texttt{attachment} should be set to one of the attachment points of the framebuffer listed in table 4.12.

\texttt{renderbuffertarget} must be \texttt{RENDERBUFFER} and \texttt{renderbuffer} should be set to the name of the renderbuffer object to be attached to the framebuffer. \texttt{renderbuffer} must be either zero or the name of an existing renderbuffer object of type \texttt{renderbuffertarget}, otherwise an \texttt{INVALID\_OPERATION} error is generated. If \texttt{renderbuffer} is zero, then the value of \texttt{renderbuffertarget} is ignored.

If \texttt{renderbuffer} is not zero and if \texttt{FramebufferRenderbuffer} is successful, then the renderbuffer named \texttt{renderbuffer} will be used as the logical buffer identified by \texttt{attachment} of the framebuffer currently bound to \texttt{target}. The value of \texttt{FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPE} for the specified attachment point is set to \texttt{RENDERBUFFER} and the value of \texttt{FRAMEBUFFER\_ATTACHMENT\_OBJECT\_NAME} is set to \texttt{renderbuffer}. All other state values of the attachment point specified by \texttt{attachment} are set to their default values listed in table 6.29. No change is made to the state of the renderbuffer object and any previous attachment to the \texttt{attachment} logical buffer of the framebuffer object bound to framebuffer \texttt{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 \texttt{FramebufferRenderbuffer} with the \texttt{renderbuffer} name zero will detach the image, if any, identified by \texttt{attachment}, in the framebuffer currently bound to \texttt{target}. All state values of the attachment point specified by \texttt{attachment} in the object bound to \texttt{target} are set to their default values listed in table 6.29. Bug 2999

Version 3.0 (with changes marked - August 11, 2008)
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*i (see caption)</td>
</tr>
<tr>
<td>DEPTH_ATTACHMENT</td>
</tr>
<tr>
<td>STENCIL_ATTACHMENT</td>
</tr>
<tr>
<td>DEPTH_STENCIL_ATTACHMENT</td>
</tr>
</tbody>
</table>

Table 4.12: Framebuffer attachment points. *i* in COLOR_ATTACHMENT*i* may range from zero to the value of `MAX_COLOR_ATTACHMENTS - 1`.

### Attaching Texture Images to a Framebuffer

GL supports copying the rendered contents of the framebuffer into the images of a texture object through the use of the routines `CopyTexImage*` and `CopyTexSubImage*`. Additionally, GL supports rendering directly into the images of a texture object.

To render directly into a texture image, 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`. Bug 3572
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 4.12.

If texture is zero, the image identified by attachment, if any, will be detached
from the framebuffer currently bound to target. textarget, level, and layer are
ignored. All state values of the attachment point specified by attachment are set to
their default values listed in table 6.29.

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_3D, then level must be greater than or equal to zero
and less than or equal to \( \log_2 \) 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 \( \log_2 \) 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 \( \log_2 \) 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 tex-
ture. An INVALID_VALUE error is generated if layer is larger than the value of
MAX_3D_TEXTURE_SIZE-1.

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_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.

For FramebufferTexture3D, if texture is not zero, then textarget must be
TEXTURE_3D.

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 tar-
get. The value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE for the specified attachment point is set to TEXTURE and the value of FRAMEBUFFER_ATTACHMENT_OBJECT_NAME is set to texture. Additionally, the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL for the named attachment point is set to level. If texture is a cube map texture, then the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE for the named attachment point is set to textarget. If texture is a 3D texture, then the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER for the named attachment point is set to layer. All other state values of the attachment point specified by attachment are set to their default values listed in table 6.29. 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).

The command

```c
void FramebufferTextureLayer( enum target,
    enum attachment, uint texture, int level, int layer );
```

operates identically to FramebufferTexture3D, except that it attaches a single layer of a three-dimensional texture or a one- or two-dimensional 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 layer is negative. The error INVALID_OPERATION is generated if texture is non-zero and is not the name of a three dimensional texture 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 FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER is set to layer.

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 Rendering When an Image of a Bound Texture Object is Also Attached to the Framebuffer

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 any of these conditions hold, texturing operations accessing that image will produce undefined results, as described at the end of section 3.9.7. Conditions resulting in such undefined behavior are defined in more detail below. Such undefined texturing operations are likely to leave the final results of the shader or fixed-function 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 feedback loop between the writing of pixels by the GL’s 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 at the end of the Scale Factor and Level of Detail subsection of section 3.9.7.

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 framebuffer at attachment point $A$
- the texture object $T$ is currently bound to a texture unit $U$, and
- the current fixed-function texture state or 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 \text{TEXTURE\_BASE\_LEVEL} for the texture object \( T \)

- the value of \text{TEXTURE\_MIN\_FILTER} for texture object \( T \) is one of \text{NEAREST\_MIPMAP\_NEAREST}, \text{NEAREST\_MIPMAP\_LINEAR}, \text{LINEAR\_MIPMAP\_NEAREST}, \text{or LINEAR\_MIPMAP\_LINEAR}, \) and the value of \text{FRAMEBUFFER\_ATTACHMENT\_TEXTURE\_LEVEL} for attachment point \( A \) is within the range specified by the current values of \text{TEXTURE\_BASE\_LEVEL} to \( q \), inclusive, for the texture object \( T \). (\( q \) is defined in the \text{Mipmapping} discussion of section 3.9.7).

For the purpose of this discussion, it is \textit{possible} to sample from the texture object \( T \) bound to texture unit \( U \) if any of the following are true:

- Programmable fragment processing is disabled and the target of texture object \( T \) is enabled according to the texture target precedence rules of section 3.9.17
- 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 \text{TEXTURE\_BASE\_LEVEL} and \text{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.)

### 4.4.4 Framebuffer Completeness

A framebuffer must be \textit{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 \footnote{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.

Version 3.0 (with changes marked - August 11, 2008)
4.4. FRAMEBUFFER OBJECTS

- The following base internal formats from table 3.15 are color-renderable: ALPHA, RED, RG, RGB, and RGBA. The sized internal formats from table 3.16 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 3.18 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 4.11, or if it is one of the formats from table 3.18 whose base internal format is DEPTH_STENCIL. No other formats are stencil-renderable.

Framebuffer Attachment Completeness

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE for the framebuffer attachment point attachment is not NONE, then it is said that a framebuffer-attachable image, named image, is attached to the framebuffer at the attachment point. image is identified by the state in attachment as described in section 4.4.2.

The framebuffer attachment point attachment is said to be framebuffer attachment complete if the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE for attachment is NONE (i.e., no image is attached), or if all of the following conditions are true:

- image is a component of an existing object with the name specified by FRAMEBUFFER_ATTACHMENT_OBJECT_NAME, and of the type specified by FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE.

- The width and height of image are non-zero.

- If FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is TEXTURE and FRAMEBUFFER_ATTACHMENT_OBJECT_NAME names a three-dimensional texture, then FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER must be smaller than the depth of the texture.

- If FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is TEXTURE and FRAMEBUFFER_ATTACHMENT_OBJECT_NAME names a one- or two-dimensional array texture, then FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER must be smaller than the number of layers in the texture.

Version 3.0 (with changes marked - August 11, 2008
• If attachment is COLOR_ATTACHMENTi, then image must have a color-renderable internal format.

• If attachment is DEPTH_ATTACHMENT, then image must have a depth-renderable internal format.

• If attachment is STENCIL_ATTACHMENT, then image must have a stencil-renderable internal format.

Whole Framebuffer Completeness

Each rule below is followed by an error token enclosed in { brackets }. The meaning of these errors is explained below and under “Effects of Framebuffer Completeness on Framebuffer Operations” later in section 4.4.4.

The framebuffer object target is said to be framebuffer complete if all the following conditions are true:

• target is the default framebuffer, and the default framebuffer exists.

  \{ FRAMEBUFFER_UNDEFINED \}

• All framebuffer attachment points are framebuffer attachment complete.

  \{ FRAMEBUFFER_INCOMPLETE_ATTACHMENT \}

• There is at least one image attached to the framebuffer.

  \{ FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT \}

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

  \{ 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; and, if the attached images are a mix of renderbuffers and textures, the value of RENDERBUFFER_SAMPLES is zero for all attached renderbuffers.

{ FRAMEBUFFER_INCOMPLETE_MULTISAMPLE }

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.

Version 3.0 (with changes marked - August 11, 2008)
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 target can be queried by calling

```c
enum CheckFramebufferStatus( enum target );
```

**target must be** DRAW_FRAMEBUFFER, READ_FRAMEBUFFER, or FRAMEBUFFER. FRAMEBUFFER is equivalent to DRAW_FRAMEBUFFER. If CheckFramebufferStatus is called within a Begin/End pair, an INVALID_OPERATION error is generated. If CheckFramebufferStatus generates an error, zero is returned.

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

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

**Required Framebuffer Formats**

Implementations must support framebuffer objects with up to MAX_COLOR_ATTACHMENTS color attachments, a depth attachment, and a stencil attachment. Each color attachment may be in any of the required color formats for textures and renderbuffers described in sections 3.9.1 and 4.4.2. The depth attachment may be in any of the required depth or combined depth+stencil...
4.4. FRAMEBUFFER OBJECTS

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.

Effects of Framebuffer Completeness on Framebuffer Operations

Attempting to render to or read from a framebuffer which is not framebuffer com-
plete will generate an INVALID_FRAMEBUFFER_OPERATION error. This means
that rendering commands such as Begin, RasterPos, any command that per-
forms an implicit Begin, as well as commands that read the framebuffer such
as ReadPixels, CopyTexImage, and CopyTexSubImage, will generate the error
INVALID_FRAMEBUFFER_OPERATION if called while the framebuffer is not frame-
buffer complete.

4.4.5 Effects of Framebuffer State on Framebuffer Dependent Values

The values of the state variables listed in table 6.51 may change when a change is
made to DRAW_FRAMEBUFFER_BINDING, to the state of the currently bound frame-
buffer object, or to an image attached to the currently bound framebuffer object.

When DRAW_FRAMEBUFFER_BINDING is zero, the values of the state variables
listed in table 6.51 are implementation defined.

When DRAW_FRAMEBUFFER_BINDING is non-zero, if the currently bound
framebuffer object is not framebuffer complete, then the values of the state vari-
ables listed in table 6.51 are undefined.

When 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 6.51 are completely determined by 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 values of RED_BITS, GREEN_BITS, BLUE_BITS, and ALPHA_BITS are defined only if all color attach-
ments of the draw framebuffer have identical formats, in which case the color
component depths of color attachment zero are returned. The values returned for
DEPTH_BITS and STENCIL_BITS are the depth or stencil component depth of the
corresponding attachment of the draw framebuffer, respectively. The actual sizes
of the color, depth, or stencil bit planes can be obtained by querying an attachment
point using GetFramebufferAttachmentParameteriv, or querying the object at-
tached to that point. If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE
at a particular attachment point is RENDERBUFFER, the sizes may be determined by
calling GetRenderbufferParameteriv as described in section 6.1.3. If the value
of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` at a particular attachment point is `TEXTURE`, the sizes may be determined by calling `GetTexParameter`, as described in section 6.1.3.

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

When `DRAW_FRAMEBUFFER_BINDING` is non-zero, an operation that writes to the framebuffer modifies the image attached to the selected logical buffer, and an 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 3.10 as follows:

\[
\begin{align*}
i &= (x_w - b) \\
j &= (y_w - b) \\
k &= (layer - b)
\end{align*}
\]

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 3.15, 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.

---

Version 3.0 (with changes marked - August 11, 2008)
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 3.23 and the internal format of the attached image.
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 evaluators (used to model curves and surfaces), selection (used to locate rendered primitives on the screen), feedback (which returns GL results before rasterization), display lists (used to designate a group of GL commands for later execution by the GL), flushing and finishing (used to synchronize the GL command stream), and hints.

5.1 Evaluators

Evaluators provide a means to use a polynomial or rational polynomial mapping to produce vertex, normal, and texture coordinates, and colors. The values so produced are sent on to further stages of the GL as if they had been provided directly by the client. Transformations, lighting, primitive assembly, rasterization, and per-pixel operations are not affected by the use of evaluators.

Consider the $\mathbb{R}^k$-valued polynomial $p(u)$ defined by

$$p(u) = \sum_{i=0}^{n} B_i^n(u) R_i$$

(5.1)

with $R_i \in \mathbb{R}^k$ and

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i},$$

the $i$th Bernstein polynomial of degree $n$ (recall that $0^0 \equiv 1$ and $\binom{n}{0} \equiv 1$). Each $R_i$ is a control point. The relevant command is

```c
void Map1(fd)( enum target, T u1, T u2, int stride, int order, T points );
```
5.1. EVALUATORS

Table 5.1: Values specified by the target to Map1. Values are given in the order in which they are taken.

<table>
<thead>
<tr>
<th>target</th>
<th>k</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAP1 VERTEX 3</td>
<td>3</td>
<td>(x, y, z) vertex coordinates</td>
</tr>
<tr>
<td>MAP1 VERTEX 4</td>
<td>4</td>
<td>(x, y, z, w) vertex coordinates</td>
</tr>
<tr>
<td>MAP1 INDEX</td>
<td>1</td>
<td>color index</td>
</tr>
<tr>
<td>MAP1 COLOR 4</td>
<td>4</td>
<td>R, G, B, A</td>
</tr>
<tr>
<td>MAP1 NORMAL</td>
<td>3</td>
<td>(x, y, z) normal coordinates</td>
</tr>
<tr>
<td>MAP1 TEXTURE_COORD 1</td>
<td>1</td>
<td>(s) texture coordinate</td>
</tr>
<tr>
<td>MAP1 TEXTURE_COORD 2</td>
<td>2</td>
<td>(s, t) texture coordinates</td>
</tr>
<tr>
<td>MAP1 TEXTURE_COORD 3</td>
<td>3</td>
<td>(s, t, r) texture coordinates</td>
</tr>
<tr>
<td>MAP1 TEXTURE_COORD 4</td>
<td>4</td>
<td>(s, t, r, q) texture coordinates</td>
</tr>
</tbody>
</table>

\(target\) is a symbolic constant indicating the range of the defined polynomial. Its possible values, along with the evaluations that each indicates, are given in table 5.1. \(order\) is equal to \(n + 1\); The error INVALID.VALUE is generated if \(order\) is less than one or greater than MAX_EVAL_ORDER. \(points\) is a pointer to a set of \(n + 1\) blocks of storage. Each block begins with \(k\) single-precision floating-point or double-precision floating-point values, respectively. The rest of the block may be filled with arbitrary data. Table 5.1 indicates how \(k\) depends on \(target\) and what the \(k\) values represent in each case.

\(stride\) is the number of single- or double-precision values (as appropriate) in each block of storage. The error INVALID.VALUE results if \(stride\) is less than \(k\). The order of the polynomial, \(order\), is also the number of blocks of storage containing control points.

\(u_1\) and \(u_2\) give two floating-point values that define the endpoints of the pre-image of the map. When a value \(u'\) is presented for evaluation, the formula used is

\[ p'(u') = p \left( \frac{u' - u_1}{u_2 - u_1} \right). \]

The error INVALID.VALUE results if \(u_1 = u_2\).

\(Map2\) is analogous to \(Map1\), except that it describes bivariate polynomials of the form

\[ p(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u)B_j^m(v)R_{ij}. \]

The form of the \(Map2\) command is
5.1. EVALUATORS

Figure 5.1. Map Evaluation.

```c
void Map2(fd){
    enum target, T u1, T u2, int ustride,
    int uorder, T v1, T v2, int vstride, int vorder, T points);

    target is a range type selected from the same group as is used for Map1, ex-
    cept that the string MAP1 is replaced with MAP2. points is a pointer to
    (n + 1)(m + 1) blocks of storage (uorder = n + 1 and vorder = m + 1; the er-
    ror INVALID_VALUE is generated if either uorder or vorder is less than one or
    greater than MAX_EVAL_ORDER). The values comprising R_{ij} are located

    (ustride)i + (vstride)j

values (either single- or double-precision floating-point, as appropriate) past the
first value pointed to by points. u1, u2, v1, and v2 define the pre-image rectangle
of the map; a domain point \((u', v')\) is evaluated as

\[ p'(u', v') = p\left(\frac{u' - u_1}{u_2 - u_1}, \frac{v' - v_1}{v_2 - v_1}\right). \]

The evaluation of a defined map is enabled or disabled with Enable and
Disable using the constant corresponding to the map as described above. The
evaluator map generates only coordinates for texture unit TEXTURE0. The error
INVALID_VALUE results if either ustride or vstride is less than k, or if u1 is equal
to u2, or if v1 is equal to v2. If the value of ACTIVE_TEXTURE is not TEXTURE0,
calling Map12 generates the error INVALID_OPERATION.

Figure 5.1 describes map evaluation schematically; an evaluation of enabled
maps is effected in one of two ways. The first way is to use

```c
void EvalCoord12(fd)( T arg );
void EvalCoord12(fd)v( T arg );
```

Version 3.0 (with changes marked - August 11, 2008
5.1. EVALUATORS

EvalCoord1 causes evaluation of the enabled one-dimensional maps. The argument is the value (or a pointer to the value) that is the domain coordinate, \( u' \). EvalCoord2 causes evaluation of the enabled two-dimensional maps. The two values specify the two domain coordinates, \( u' \) and \( v' \), in that order.

When one of the EvalCoord commands is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if a corresponding GL command were issued with the resulting coordinates, with one important difference. The difference is that when an evaluation is performed, the GL uses evaluated values instead of current values for those evaluations that are enabled (otherwise, the current values are used). The order of the effective commands is immaterial, except that Vertex (for vertex coordinate evaluation) must be issued last. Use of evaluators has no effect on the current color, normal, or texture coordinates. If ColorMaterial is enabled, evaluated color values affect the result of the lighting equation as if the current color was being modified, but no change is made to the tracking lighting parameters or to the current color.

No command is effectively issued if the corresponding map (of the indicated dimension) is not enabled. If more than one evaluation is enabled for a particular dimension (e.g. MAP1_TEXTURE_COORD_1 and MAP1_TEXTURE_COORD_2), then only the result of the evaluation of the map with the highest number of coordinates is used.

Finally, if either MAP2_VERTEX_3 or MAP2_VERTEX_4 is enabled, then the normal to the surface is computed. Analytic computation, which sometimes yields normals of length zero, is one method which may be used. If automatic normal generation is enabled, then this computed normal is used as the normal associated with a generated vertex. Automatic normal generation is controlled with Enable and Disable with the symbolic constant AUTO_NORMAL. If automatic normal generation is disabled, then a corresponding normal map, if enabled, is used to produce a normal. If neither automatic normal generation nor a normal map are enabled, then no normal is sent with a vertex resulting from an evaluation (the effect is that the current normal is used).

For MAP_VERTEX_3, let \( q = p \). For MAP_VERTEX_4, let \( q = (x/w, y/w, z/w) \), where \((x, y, z, w) = p\). Then let

\[
\mathbf{m} = \frac{\partial q}{\partial u} \times \frac{\partial q}{\partial v}.
\]

Then the generated analytic normal, \( \mathbf{n} \), is given by \( \mathbf{n} = \mathbf{m} \) if a vertex shader is active, or else by \( \mathbf{n} = \frac{\mathbf{m}}{||\mathbf{m}||} \).

The second way to carry out evaluations is to use a set of commands that provide for efficient specification of a series of evenly spaced values to be mapped. This method proceeds in two steps. The first step is to define a grid in the domain.

Version 3.0 (with changes marked - August 11, 2008)
This is done using

```c
void MapGrid1{fd}( int n, T u'_1, T u'_2 );
```

for a one-dimensional map or

```c
void MapGrid2{fd}( int n, T u'_1, T u'_2, int n_v, T v'_1, T v'_2 );
```

for a two-dimensional map. In the case of `MapGrid1`, `u'_1` and `u'_2` describe an interval, while `n` describes the number of partitions of the interval. The error `INVALID_VALUE` results if `n ≤ 0`. For `MapGrid2`, `(u'_1, v'_1)` specifies one two-dimensional point and `(u'_2, v'_2)` specifies another. `n_u` gives the number of partitions between `u'_1` and `u'_2`, and `n_v` gives the number of partitions between `v'_1` and `v'_2`. If either `n_u ≤ 0` or `n_v ≤ 0`, then the error `INVALID_VALUE` occurs.

Once a grid is defined, an evaluation on a rectangular subset of that grid may be carried out by calling

```c
void EvalMesh1( enum mode, int p_1, int p_2 );
```

`mode` is either `POINT` or `LINE`. The effect is the same as performing the following code fragment, with \( \Delta u' = (u'_2 - u'_1)/n \):

```c
Begin( type );
for i = p_1 to p_2 step 1.0
   EvalCoord1( i * \Delta u' + u'_1 );
End( );
```

where `EvalCoord1f` or `EvalCoord1d` is substituted for `EvalCoord1` as appropriate. If `mode` is `POINT`, then `type` is `POINTS`; if `mode` is `LINE`, then `type` is `LINE_STRIP`. The one requirement is that if either \( i = 0 \) or \( i = n \), then the value computed from \( i \ast \Delta u' + u'_1 \) is precisely \( u'_1 \) or \( u'_2 \), respectively.

The corresponding commands for two-dimensional maps are

```c
void EvalMesh2( enum mode, int p_1, int p_2, int q_1, int q_2 );
```

`mode` must be `FILL`, `LINE`, or `POINT`. When `mode` is `FILL`, then these commands are equivalent to the following, with \( \Delta u' = (u'_2 - u'_1)/n \) and \( \Delta v' = (v'_2 - v'_1)/m \):

Version 3.0 (with changes marked - August 11, 2008
for $i = q_1$ to $q_2 - 1$ step 1.0
Begin (QUAD_STRIP);
for $j = p_1$ to $p_2$ step 1.0
  EvalCoord2 ($j * \Delta u' + u'_1$, $i * \Delta v' + v'_1$);
  EvalCoord2 ($j * \Delta u' + u'_1$, $(i + 1) * \Delta v' + v'_1$);
End ();

If \textit{mode} is \textsc{line}, then a call to \textbf{EvalMesh2} is equivalent to

for $i = q_1$ to $q_2$ step 1.0
Begin (LINE_STRIP);
for $j = p_1$ to $p_2$ step 1.0
  EvalCoord2 ($j * \Delta u' + u'_1$, $i * \Delta v' + v'_1$);
End ();
for $i = p_1$ to $p_2$ step 1.0
Begin (LINE_STRIP);
for $j = q_1$ to $q_2$ step 1.0
  EvalCoord2 ($i * \Delta u' + u'_1$, $j * \Delta v' + v'_1$);
End ();

If \textit{mode} is \textsc{point}, then a call to \textbf{EvalMesh2} is equivalent to

Begin (POINTS);
for $i = q_1$ to $q_2$ step 1.0
for $j = p_1$ to $p_2$ step 1.0
  EvalCoord2 ($j * \Delta u' + u'_1$, $i * \Delta v' + v'_1$);
End ();

Again, in all three cases, there is the requirement that $0 * \Delta u' + u'_1 = u'_1$, $n * \Delta u' + u'_1 = u'_2$, $0 * \Delta v' + v'_1 = v'_1$, and $m * \Delta v' + v'_1 = v'_2$.

An evaluation of a single point on the grid may also be carried out:

\begin{verbatim}
void EvalPoint1(int p);
\end{verbatim}

Calling it is equivalent to the command

\begin{verbatim}
EvalCoord1 ($p * \Delta u' + u'_1$);
\end{verbatim}

with $\Delta u'$ and $u'_1$ defined as above.

\begin{verbatim}
void EvalPoint2(int p, int q);
\end{verbatim}

is equivalent to the command

\begin{verbatim}
EvalCoord2 ($p * \Delta u' + u'_1$, $q * \Delta v' + v'_1$);
\end{verbatim}
5.2 Selection

Selection is used to determine which primitives are drawn into some region of a window. The region is defined by the current model-view and perspective matrices.

Selection works by returning an array of integer-valued names. This array represents the current contents of the name stack. This stack is controlled with the commands

```c
void InitNames( void );
void PopName( void );
void PushName( uint name );
void LoadName( uint name );
```

InitNames empties (clears) the name stack. PopName pops one name off the top of the name stack. PushName causes name to be pushed onto the name stack.

\textbf{EvalCoord2} \((p * \Delta u' + u'_1, q * \Delta v' + v'_1)\);

The state required for evaluators potentially consists of 9 one-dimensional map specifications and 9 two-dimensional map specifications, as well as corresponding flags for each specification indicating which are enabled. Each map specification consists of one or two orders, an appropriately sized array of control points, and a set of two values (for a one-dimensional map) or four values (for a two-dimensional map) to describe the domain. The maximum possible order, for either \(u\) or \(v\), is implementation dependent (one maximum applies to both \(u\) and \(v\)), but must be at least 8. Each control point consists of between one and four floating-point values (depending on the type of the map). Initially, all maps have order 1 (making them constant maps). All vertex coordinate maps produce the coordinates \((0, 0, 0, 1)\) (or the appropriate subset); all normal coordinate maps produce \((0, 0, 1)\); RGBA maps produce \((1, 1, 1, 1)\); color index maps produce 1.0; and texture coordinate maps produce \((0, 0, 0, 1)\). In the initial state, all maps are disabled. A flag indicates whether or not automatic normal generation is enabled for two-dimensional maps. In the initial state, automatic normal generation is disabled. Also required are two floating-point values and an integer number of grid divisions for the one-dimensional grid specification and four floating-point values and two integer grid divisions for the two-dimensional grid specification. In the initial state, the bounds of the domain interval for 1-D is 0 and 1.0, respectively; for 2-D, they are \((0, 0)\) and \((1, 0, 1, 0)\), respectively. The number of grid divisions is 1 for 1-D and 1 in both directions for 2-D. If any evaluation command is issued when no vertex map is enabled for the map dimension being evaluated, nothing happens.
LoadName replaces the value on the top of the stack with name. Loading a name onto an empty stack generates the error INVALID_OPERATION. Popping a name off of an empty stack generates STACK_UNDERFLOW; pushing a name onto a full stack generates STACK_OVERFLOW. The maximum allowable depth of the name stack is implementation dependent but must be at least 64.

In selection mode, framebuffer updates as described in chapter 4 are not performed. The GL is placed in selection mode with

```c
int RenderMode( enum mode );
```

`mode` is a symbolic constant: one of RENDER, SELECT, or FEEDBACK. RENDER is the default, corresponding to rendering as described until now. SELECT specifies selection mode, and FEEDBACK specifies feedback mode (described below). Use of any of the name stack manipulation commands while the GL is not in selection mode has no effect.

Selection is controlled using

```c
void SelectBuffer( sizei n, uint *buffer );
```

`buffer` is a pointer to an array of unsigned integers (called the selection array) to be potentially filled with names, and `n` is an integer indicating the maximum number of values that can be stored in that array. Placing the GL in selection mode before `SelectBuffer` has been called results in an error of INVALID_OPERATION as does calling `SelectBuffer` while in selection mode.

In selection mode, if a point, line, polygon, or the valid coordinates produced by a RasterPos command intersects the clip volume (section 2.17) then this primitive (or RasterPos command) causes a selection hit. WindowPos commands always generate a selection hit, since the resulting raster position is always valid. In the case of polygons, no hit occurs if the polygon would have been culled, but selection is based on the polygon itself, regardless of the setting of PolygonMode. When in selection mode, whenever a name stack manipulation command is executed or `RenderMode` is called and there has been a hit since the last time the stack was manipulated or `RenderMode` was called, then a hit record is written into the selection array.

A hit record consists of the following items in order: a non-negative integer giving the number of elements on the name stack at the time of the hit, a minimum depth value, a maximum depth value, and the name stack with the bottommost element first. The minimum and maximum depth values are the minimum and maximum taken over all the window coordinate z values of each (post-clipping) vertex of each primitive that intersects the clipping volume since the last hit record was
written. The minimum and maximum (each of which lies in the range \([0, 1]\)) are each multiplied by \(2^{32} - 1\) and rounded to the nearest unsigned integer to obtain the values that are placed in the hit record. No depth offset arithmetic (section 3.6.5) is performed on these values.

Hit records are placed in the selection array by maintaining a pointer into that array. When selection mode is entered, the pointer is initialized to the beginning of the array. Each time a hit record is copied, the pointer is updated to point at the array element after the one into which the topmost element of the name stack was stored. If copying the hit record into the selection array would cause the total number of values to exceed \(n\), then as much of the record as fits in the array is written and an overflow flag is set.

Selection mode is exited by calling `RenderMode` with an argument value other than `SELECT`. When called while in selection mode, `RenderMode` returns the number of hit records copied into the selection array and resets the `SelectBuffer` pointer to its last specified value. Values are not guaranteed to be written into the selection array until `RenderMode` is called. If the selection array overflow flag was set, then `RenderMode` returns \(-1\) and clears the overflow flag. The name stack is cleared and the stack pointer reset whenever `RenderMode` is called.

The state required for selection consists of the address of the selection array and its maximum size, the name stack and its associated pointer, a minimum and maximum depth value, and several flags. One flag indicates the current `RenderMode` value. In the initial state, the GL is in the `RENDER` mode. Another flag is used to indicate whether or not a hit has occurred since the last name stack manipulation. This flag is reset upon entering selection mode and whenever a name stack manipulation takes place. One final flag is required to indicate whether the maximum number of copied names would have been exceeded. This flag is reset upon entering selection mode. This flag, the address of the selection array, and its maximum size are GL client state.

### 5.3 Feedback

The GL is placed in feedback mode by calling `RenderMode` with `FEEDBACK`. When in feedback mode, framebuffer updates as described in chapter 4 are not performed. Instead, information about primitives that would have otherwise been rasterized is returned to the application via the *feedback buffer*.

Feedback is controlled using

```c
void FeedbackBuffer(sizei n, enum type, float *buffer);
```

Version 3.0 (with changes marked - August 11, 2008)
5.3. FEEDBACK

`buffer` is a pointer to an array of floating-point values into which feedback information will be placed, and `n` is a number indicating the maximum number of values that can be written to that array. `type` is a symbolic constant describing the information to be fed back for each vertex (see figure 5.2). The error `INVALID_OPERATION` results if the GL is placed in feedback mode before a call to `FeedbackBuffer` has been made, or if a call to `FeedbackBuffer` is made while in feedback mode.

While in feedback mode, each primitive that would be rasterized (or bitmap or call to `DrawPixels` or `CopyPixels`, if the raster position is valid) generates a block of values that get copied into the feedback array. If doing so would cause the number of entries to exceed the maximum, the block is partially written so as to fill the array (if there is any room left at all). The first block of values generated after the GL enters feedback mode is placed at the beginning of the feedback array, with subsequent blocks following. Each block begins with a code indicating the primitive type, followed by values that describe the primitive’s vertices and associated data. Entries are also written for bitmaps and pixel rectangles. Feedback occurs after polygon culling (section 3.6.1) and `PolygonMode` interpretation of polygons (section 3.6.4) has taken place. It may also occur after polygons with more than three edges are broken up into triangles (if the GL implementation renders polygons by performing this decomposition). `x, y, and z` coordinates returned by feedback are window coordinates; if `w` is returned, it is in clip coordinates. No depth offset arithmetic (section 3.6.5) is performed on the `z` values. In the case of bitmaps and pixel rectangles, the coordinates returned are those of the current raster position.

The texture coordinates and colors returned are those resulting from the clipping operations described in section 2.19.8. Only coordinates for texture unit TEXTURE0 are returned even for implementations which support multiple texture units. The colors returned are the primary colors.

The ordering rules for GL command interpretation also apply in feedback mode. Each command must be fully interpreted and its effects on both GL state and the values to be written to the feedback buffer completed before a subsequent command may be executed.

Feedback mode is exited by calling `RenderMode` with an argument value other than `FEEDBACK`. When called while in feedback mode, `RenderMode` returns the number of values placed in the feedback array and resets the feedback array pointer to be `buffer`. The return value never exceeds the maximum number of values passed to `FeedbackBuffer`.

If writing a value to the feedback buffer would cause more values to be written than the specified maximum number of values, then the value is not written and an overflow flag is set. In this case, `RenderMode` returns `-1` when it is called, after which the overflow flag is reset. While in feedback mode, values are not guaranteed
### 5.4. DISPLAY LISTS

<table>
<thead>
<tr>
<th>Type</th>
<th>coordinates</th>
<th>color</th>
<th>texture</th>
<th>total values</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D</td>
<td>x, y</td>
<td>–</td>
<td>–</td>
<td>2</td>
</tr>
<tr>
<td>3D</td>
<td>x, y, z</td>
<td>–</td>
<td>–</td>
<td>3</td>
</tr>
<tr>
<td>3D_COLOR</td>
<td>x, y, z</td>
<td>k</td>
<td>–</td>
<td>3 + k</td>
</tr>
<tr>
<td>3D_COLOR_TEXTURE</td>
<td>x, y, z</td>
<td>k</td>
<td>4</td>
<td>7 + k</td>
</tr>
<tr>
<td>4D_COLOR_TEXTURE</td>
<td>x, y, z, w</td>
<td>k</td>
<td>4</td>
<td>8 + k</td>
</tr>
</tbody>
</table>

Table 5.2: Correspondence of feedback type to number of values per vertex. $k$ is 1 in color index mode and 4 in RGBA mode.

To be written into the feedback buffer before RenderMode is called.

Figure 5.2 gives a grammar for the array produced by feedback. Each primitive is indicated with a unique identifying value followed by some number of vertices. A vertex is fed back as some number of floating-point values determined by the feedback type. Table 5.2 gives the correspondence between feedback buffer and the number of values returned for each vertex.

The command

```c
void PassThrough(float token);
```

may be used as a marker in feedback mode. `token` is returned as if it were a primitive; it is indicated with its own unique identifying value. The ordering of any `PassThrough` commands with respect to primitive specification is maintained by feedback. `PassThrough` may not occur between `Begin` and `End`. It has no effect when the GL is not in feedback mode.

The state required for feedback is the pointer to the feedback array, the maximum number of values that may be placed there, and the feedback type. An overflow flag is required to indicate whether the maximum allowable number of feedback values has been written; initially this flag is cleared. These state variables are GL client state. Feedback also relies on the same mode flag as selection to indicate whether the GL is in feedback, selection, or normal rendering mode.

### 5.4 Display Lists

A display list is simply a group of GL commands and arguments that has been stored for subsequent execution. The GL may be instructed to process a particular display list (possibly repeatedly) by providing a number that uniquely specifies it. Doing so causes the commands within the list to be executed just as if they were given normally. The only exception pertains to commands that rely upon client
5.4. DISPLAY LISTS

feedback-list:
  feedback-item feedback-list
feedback-item

feedback-item:
  point
  line-segment
  polygon
  bitmap
  pixel-rectangle
  passthrough

point:
  POINT_TOKEN vertex

line-segment:
  LINE_TOKEN vertex vertex
  LINE_RESET_TOKEN vertex vertex

polygon:
  POLYGON_TOKEN n polygon-spec

polygon-spec:
  polygon-spec vertex
  vertex vertex vertex

bitmap:
  BITMAP_TOKEN vertex

pixel-rectangle:
  DRAW_PIXEL_TOKEN vertex
  COPY_PIXEL_TOKEN vertex

passthrough:
  PASS_THROUGH_TOKEN f

vertex:
  2D:
  f f
  3D:
  f f f
  3D_COLOR:
  f f f color
  3D_COLOR_TEXTURE:
  f f f color tex
  4D_COLOR_TEXTURE:
  f f f f color tex

color:
  f f f

tex:
  f f f

Figure 5.2: Feedback syntax. f is a floating-point number. n is a floating-point integer giving the number of vertices in a polygon. The symbols ending with _TOKEN are symbolic floating-point constants. The labels under the “vertex” rule show the different data returned for vertices depending on the feedback type. LINE_TOKEN and LINE_RESET_TOKEN are identical except that the latter is returned only when the line stipple is reset for that line segment.
state. When such a command is accumulated into the display list (that is, when
issued, not when executed), the client state in effect at that time applies to the com-
mand. Only server state is affected when the command is executed. As always,
 pointers which are passed as arguments to commands are dereferenced when the
command is issued. (Vertex array pointers are dereferenced when the commands
ArrayElement, DrawArrays, DrawElements, or DrawRangeElements are ac-
cumulated into a display list.)

A display list is begun by calling

```c
void NewList( uint n, enum mode );
```

$n$ is a positive integer to which the display list that follows is assigned, and $mode$ is a
symbolic constant that controls the behavior of the GL during display list creation.
If $mode$ is COMPILE, then commands are not executed as they are placed in the
display list. If $mode$ is COMPILE_AND_EXECUTE then commands are executed as
they are encountered, then placed in the display list. If $n = 0$, then the error
INVALID_VALUE is generated.

After calling NewList all subsequent GL commands are placed in the display
list (in the order the commands are issued) until a call to

```c
void EndList( void );
```

occurs, after which the GL returns to its normal command execution state. It is
only when EndList occurs that the specified display list is actually associated with
the index indicated with NewList. The error INVALID_OPERATION is generated if
EndList is called without a previous matching NewList, or if NewList is called
a second time before calling EndList. The error OUT_OF_MEMORY is generated if
EndList is called and the specified display list cannot be stored because insufficient
memory is available. In this case GL implementations of revision 1.1 or greater
insure that no change is made to the previous contents of the display list, if any,
and that no other change is made to the GL state, except for the state changed by
execution of GL commands when the display list mode is COMPILE_AND_EXECUTE.

Once defined, a display list is executed by calling

```c
void CallList( uint n );
```

$n$ gives the index of the display list to be called. This causes the commands saved
in the display list to be executed, in order, just as if they were issued without using
a display list. If $n = 0$, then the error INVALID_VALUE is generated.

The command
void CallLists(sizei n, enum type, void *lists);

provides an efficient means for executing a number of display lists. $n$ is an integer indicating the number of display lists to be called, and lists is a pointer that points to an array of offsets. Each offset is constructed as determined by lists as follows. First, type may be one of the constants BYTE, UNSIGNED_BYTE, SHORT, UNSIGNED_SHORT, INT, UNSIGNED_INT, or FLOAT indicating that the array pointed to by lists is an array of bytes, unsigned bytes, shorts, unsigned shorts, integers, unsigned integers, or floats, respectively. In this case each offset is found by simply converting each array element to an integer (floating point values are truncated to negative infinity). Further, type may be one of 2_BYTES, 3_BYTES, or 4_BYTES, indicating that the array contains sequences of 2, 3, or 4 unsigned bytes, in which case each integer offset is constructed according to the following algorithm:

\[
\text{offset} \leftarrow 0 \\
\text{for } i = 1 \text{ to } b \\
\quad \text{offset} \leftarrow \text{offset shifted left 8 bits} \\
\quad \text{offset} \leftarrow \text{offset + byte} \\
\quad \text{advance to next byte in the array}
\]

$b$ is 2, 3, or 4, as indicated by type. If $n = 0$, CallLists does nothing.

Each of the $n$ constructed offsets is taken in order and added to a display list base to obtain a display list number. For each number, the indicated display list is executed. The base is set by calling

void ListBase(uint base);

to specify the offset.

Indicating a display list index that does not correspond to any display list has no effect. CallList or CallLists may appear inside a display list. (If the mode supplied to NewList is COMPARE_AND_EXECUTE, then the appropriate lists are executed, but the CallList or CallLists, rather than those lists’ constituent commands, is placed in the list under construction.) To avoid the possibility of infinite recursion resulting from display lists calling one another, an implementation dependent limit is placed on the nesting level of display lists during display list execution. This limit must be at least 64.

Two commands are provided to manage display list indices.

uint GenLists(sizei s);
returns an integer \( n \) such that the indices \( n, \ldots, n+s-1 \) are previously unused (i.e. there are \( s \) previously unused display list indices starting at \( n \)). \texttt{GenLists} also has the effect of creating an empty display list for each of the indices \( n, \ldots, n+s-1 \), so that these indices all become used. \texttt{GenLists} returns 0 if there is no group of \( s \) contiguous previously unused display list indices, or if \( s = 0 \).

\[
\begin{align*}
\text{boolean} \ & \texttt{IsList}( \text{uint list} ); \\
\text{returns} \ & \text{TRUE} \text{ if } list \text{ is the index of some display list.}
\end{align*}
\]

A contiguous group of display lists may be deleted by calling

\[
\begin{align*}
\text{void} \ & \texttt{DeleteLists}( \text{uint list, sizei range} ); \\
\text{where} \ & list \text{ is the index of the first display list to be deleted and range is the number} \\
\text{of display lists to be deleted. All information about the display lists is lost, and the} \\
\text{indices become unused. Indices to which no display list corresponds are ignored.} \\
\text{If} \ & range = 0, \text{nothing happens.}
\end{align*}
\]

Certain commands, when called while compiling a display list, are not compiled into the display list but are executed immediately. These commands fall in several categories including

- \textit{Display lists}: \texttt{GenLists} and \texttt{DeleteLists}.
- \textit{Render modes}: \texttt{FeedbackBuffer}, \texttt{SelectBuffer}, and \texttt{RenderMode}.
- \textit{Client state}: \texttt{EnableClientState}, \texttt{DisableClientState}, \texttt{EnableVertexAttribArray}, \texttt{DisableVertexAttribArray}, \texttt{PushClientAttrib}, and \texttt{PopClientAttrib}.
- \textit{Pixels and textures}: \texttt{PixelStore}, \texttt{ReadPixels}, \texttt{GenTextures}, \texttt{DeleteTextures}, \texttt{AreTexturesResident}, and \texttt{GenerateMipmap}.
- \textit{Occlusion queries}: \texttt{GenQueries} and \texttt{DeleteQueries}.
- \textit{Vertex buffer objects}: \texttt{GenBuffers}, \texttt{DeleteBuffers}, \texttt{BindBuffer}, \texttt{BindBufferRange}, \texttt{BindBufferBase}, \texttt{TransformFeedbackVaryings}, \texttt{BufferData}, \texttt{BufferSubData}, \texttt{MapBuffer}, and \texttt{UnmapBuffer}.
- \textit{Framebuffer} and \textit{renderbuffer} objects: \texttt{GenFramebuffers}, \texttt{BindFramebuffer}, \texttt{DeleteFramebuffers}, \texttt{CheckFramebufferStatus}, \texttt{GenRenderbuffers}, \texttt{BindRenderbuffer}, \texttt{DeleteRenderbuffers}, \texttt{RenderbufferStorage}, \texttt{RenderbufferStorageMultisample}, \texttt{FramebufferTexture1D}, \texttt{FramebufferTexture2D}, \texttt{FramebufferTexture3D}, \texttt{FramebufferTextureLayer}, \texttt{FramebufferRenderbuffer}, and \texttt{BlitFramebuffer}.

\[
\begin{align*}
\text{Version 3.0 (with changes marked - August 11, 2008}
\end{align*}
\]
5.5. Flush and Finish

The command

```c
void Flush(void);
```
5.6. **HINTS** 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.

## 5.6 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 5.3; 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 5.3: Hint targets and descriptions.

<table>
<thead>
<tr>
<th>Target</th>
<th>Hint description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PERSPECTIVE_CORRECTION_HINT</td>
<td>Quality of parameter interpolation</td>
</tr>
<tr>
<td>POINT_SMOOTH_HINT</td>
<td>Point sampling quality</td>
</tr>
<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>FOG_HINT</td>
<td>Fog quality (calculated per-pixel or per-vertex)</td>
</tr>
<tr>
<td>GENERATE_MIPMAP_HINT</td>
<td>Quality and performance of automatic mipmap level generation</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 dFx, dFy and fwidth</td>
</tr>
</tbody>
</table>

Version 3.0 (with changes marked - August 11, 2008)
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 GetFloatv( enum value, float *data );
void GetDoublev( enum value, double *data );
```

The commands obtain boolean, 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 GetBooleani_v( enum target, uint index,  
                    boolean *data );
void GetIntegeri_v( enum target, uint index, int *data );
```
target is the name of the indexed state and index is the index of the particular element being queried. data is a pointer to a scalar or array of the indicated type in which to place the returned data. An INVALID.VALUE error is generated if index is outside the valid range for the indexed state target.

Finally,

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

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

```c
boolean IsEnabledi( enum target, uint index );
```

can be used to determine if the indexed state corresponding to target and index is enabled or disabled. An INVALID.VALUE error is generated if index is outside the valid range for the indexed state target.

### 6.1.2 Data Conversions

If a Get command is issued that returns value types different from the type of the value being obtained, a type conversion is performed. If 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 GetIntegerv (or any of the Get commands below) is called, a boolean value is interpreted as either 1 or 0, and a floating-point value is rounded to the nearest integer, unless the value is an RGBA color component, a DepthRange value, a depth buffer clear value, or a normal coordinate. 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 is interpreted as either 1.0 or 0.0, 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 \(f\). Similarly, points for evaluator maps are returned in the order that they appeared when passed to Map1. Map2 returns \(R_{ij}\) in the \([(uorder)i + j]\)th block of values (see page 298 for \(i, j, uorder,\) and \(R_{ij}\)).
Matrices may be queried and returned in transposed form by calling `GetBooleanv`, `GetIntegerv`, `GetFloatv`, and `GetDoublev` with `pname` set to one of `TRANSPOSE_MODELVIEW_MATRIX`, `TRANSPOSE_PROJECTION_MATRIX`, `TRANSPOSE_TEXTURE_MATRIX`, or `TRANSPOSE_COLOR_MATRIX`. The effect of

```
GetFloatv(TRANSPOSE_MODELVIEW_MATRIX, m);
```

is the same as the effect of the command sequence

```
GetFloatv(MODELVIEW_MATRIX, m);
m = m^T;
```

Similar conversions occur when querying `TRANSPOSE_PROJECTION_MATRIX`, `TRANSPOSE_TEXTURE_MATRIX`, and `TRANSPOSE_COLOR_MATRIX`.

If fragment color clamping is enabled, querying of the texture border color, texture environment color, fog color, alpha test reference value, 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. Client texture state variables such as texture coordinate array pointers are qualified by the value of `CLIENT_ACTIVE_TEXTURE`. Tables 6.5, 6.6, 6.12, 6.19, 6.22, and 6.47 indicate those state variables which are qualified by `ACTIVE_TEXTURE` or `CLIENT_ACTIVE_TEXTURE` during state queries. Queries of texture state variables corresponding to texture coordinate processing units (namely, `TexGen` state and enables, and matrices) will generate an `INVALID_OPERATION` error if the value of `ACTIVE_TEXTURE` is greater than or equal to `MAX_TEXTURE_COORDS`. 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 6.6 through 6.9 define the set of state stored in a vertex array object.
GetLightiv, GetMaterialiv, GetTexEnviv, GetTexGeniv, and GetTexParameteriv convert floating point state to integer values in the same manner as GetIntegerv (see section 6.1.2).

GetClipPlane always returns four double-precision values in *eqn_; these are the coefficients of the plane equation of *plane_ in eye coordinates (these coordinates are those that were computed when the plane was specified).

GetLight places information about *value_ (a symbolic constant) for *light_ (also a symbolic constant) in *data_. POSITION or SPOT_DIRECTION returns values in eye coordinates (again, these are the coordinates that were computed when the position or direction was specified).

GetMaterial, GetTexGen, GetTexEnv, GetTexParameter, and GetBufferParameter are similar to GetLight, placing information about *value_ for the target indicated by their first argument into *data_. The *face_ argument to GetMaterial must be either FRONT or BACK, indicating the front or back material, respectively. The *env_ argument to GetTexEnv must be either POINT_SPRITE, TEXTURE_ENV, or TEXTURE_FILTER_CONTROL. The *coord_ argument to GetTexGen must be one of S, T, R, or Q. For GetTexGen, EYE_LINEAR coefficients are returned in the eye coordinates that were computed when the plane was specified; OBJECT_LINEAR coefficients are returned in object coordinates.

GetTexParameter parameter *target_ may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, or texture_array.
6.1. QUERYING GL STATE

TEXTURE_CUBE_MAP, indicating the currently bound one-, two-, three-dimensional, one- or two-dimensional array, or cube map texture object. GetTexLevelParameter parameter target may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, 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, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_3D, PROXY_TEXTURE_1D_ARRAY, PROXY_TEXTURE_2D_ARRAY, or PROXY_TEXTURE_CUBE_MAP, indicating the one-, two-, or three-dimensional texture, the one- or two-dimensional array texture, one of the six distinct 2D images making up the cube map texture object, or the one-, two-, three-dimensional, one- or two-dimensional array, or cube map proxy state vector. Note that TEXTURE_CUBE_MAP is not a valid target parameter for GetTexLevelParameter, because it does not specify a particular cube map face. value is a symbolic value indicating which texture parameter is to be obtained. For GetTexParameter, value must be either TEXTURE_RESIDENT, or one of the symbolic values in table 3.20. Querying value TEXTURE_BORDER_COLOR with GetTexParameteriv 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. The lod argument to GetTexLevelParameter determines which level-of-detail’s state is returned. If the lod argument is less than zero or if it is larger than the maximum allowable level-of-detail then the error INVALID_VALUE occurs.

For texture images with uncompressed internal formats, queries of value of TEXTURE_RED_TYPE, TEXTURE_GREEN_TYPE, TEXTURE_BLUE_TYPE, TEXTURE_ALPHA_TYPE, TEXTURE_LUMINANCE_TYPE, TEXTURE_DEPTH_TYPE, and TEXTURE_INTENSITY_TYPE return the data type used to store the component. Types NONE, UNSIGNED_NORMALIZED, FLOAT, INT, and UNSIGNED_INT respectively indicate missing, unsigned normalized integer, floating-point, signed unnormalized integer, and unsigned unnormalized integer components. Queries of value of TEXTURE_RED_SIZE, TEXTURE_GREEN_SIZE, TEXTURE_BLUE_SIZE, TEXTURE_ALPHA_SIZE, TEXTURE_LUMINANCE_SIZE, TEXTURE_DEPTH_SIZE, TEXTURE_INTENSITY_SIZE, TEXTURE_STENCIL_SIZE, and TEXTURE_SHARED_SIZE return the actual resolutions of the stored image array components, not the resolutions specified when the image array was defined. For texture images with 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 question. Since the quality of the implementation’s compression algorithm...
6.1. QUERYING GL STATE

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`, or as `TEXTURE_COMPONENTS` for compatibility with GL version 1.0.

For `GetPixelMap`, the `map` must be a map name from table 3.3. For `GetMap`, `map` must be one of the map types described in section 5.1, and `value` must be one of `ORDER`, `COEFF`, or `DOMAIN`. The `GetPixelMapfv`, `GetPixelMapuiv`, and `GetPixelMapusv` commands write all the values in the named pixel map to `data`. `GetPixelMapuiv` and `GetPixelMapusv` convert floating point pixel map values to integers according to the `UNSIGNED_INT` and `UNSIGNED_SHORT` entries, respectively, of table 4.9. 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; otherwise, `data` is a pointer to client memory. All pixel storage and pixel transfer modes are ignored when returning a pixel map. `n` machine units are written where `n` is the size of the pixel map times the size of `FLOAT`, `UNSIGNED_INT`, or `UNSIGNED_SHORT` respectively in basic machine units. If a pixel pack buffer object is bound and `data + n` is greater than the size of the pixel buffer, 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 a `FLOAT`, `UNSIGNED_INT`, or `UNSIGNED_SHORT` respectively, an `INVALID_OPERATION` error results.

For `GetFramebufferAttachmentParameteriv`, `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`, `BACK_RIGHT`, `AUXI`, `DEPTH_BUFFER`, or `STENCIL_BUFFER`, identifying a color buffer, the depth buffer, or the stencil buffer, and `pname` may be `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` or `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME`.

If a framebuffer object is bound to `target`, then `attachment` must be one of the attachment points of the framebuffer listed in table 4.12.
and \textit{pname} may be one of \texttt{FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPE}, \texttt{FRAMEBUFFER\_ATTACHMENT\_OBJECT\_NAME}, \texttt{FRAMEBUFFER\_ATTACHMENT\_TEXTURE\_LEVEL}, \texttt{FRAMEBUFFER\_ATTACHMENT\_TEXTURE\_CUBE\_MAP\_FACE}, \texttt{texture\_array} \texttt{FRAMEBUFFER\_ATTACHMENT\_TEXTURE\_LAYER}.  Bug 2999

If \textit{attachment} is \texttt{DEPTH\_STENCIL\_ATTACHMENT}, and different objects are bound to the depth and stencil attachment points of \textit{target}, the query will fail and generate an \texttt{INVALID\_OPERATION} error. If the same object is bound to both attachment points, information about that object will be returned. Bugs 3550, 3589

In either case, \textit{pname} may also be one of \texttt{FRAMEBUFFER\_ATTACHMENT\_RED\_SIZE}, \texttt{FRAMEBUFFER\_ATTACHMENT\_GREEN\_SIZE}, \texttt{FRAMEBUFFER\_ATTACHMENT\_BLUE\_SIZE}, \texttt{FRAMEBUFFER\_ATTACHMENT\_ALPHA\_SIZE}, \texttt{FRAMEBUFFER\_ATTACHMENT\_DEPTH\_SIZE}, \texttt{FRAMEBUFFER\_ATTACHMENT\_STENCIL\_SIZE}, \texttt{FRAMEBUFFER\_ATTACHMENT\_COMPONENT\_TYPE}, or \texttt{FRAMEBUFFER\_ATTACHMENT\_COLOR\_ENCODING}.  Bugs 3550, 3589

Upon successful return from \texttt{GetFramebufferAttachmentParameteriv}, if \textit{pname} is \texttt{FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPE}, then \textit{param} will contain one of \texttt{NONE}, \texttt{FRAMEBUFFER\_DEFAULT}, \texttt{TEXTURE}, \texttt{RENDERBUFFER}, identifying the type of object which contains the attached image. If \textit{pname} is \texttt{FRAMEBUFFER\_ATTACHMENT\_RED\_SIZE}, \texttt{FRAMEBUFFER\_ATTACHMENT\_GREEN\_SIZE}, \texttt{FRAMEBUFFER\_ATTACHMENT\_BLUE\_SIZE}, \texttt{FRAMEBUFFER\_ATTACHMENT\_ALPHA\_SIZE}, \texttt{FRAMEBUFFER\_ATTACHMENT\_DEPTH\_SIZE}, \texttt{FRAMEBUFFER\_ATTACHMENT\_STENCIL\_SIZE}, \texttt{FRAMEBUFFER\_ATTACHMENT\_COMPONENT\_TYPE}, or \texttt{FRAMEBUFFER\_ATTACHMENT\_COLOR\_ENCODING}, then \textit{param} will contain the number of bits in the corresponding red, green, blue, alpha, depth, or stencil component of the specified \textit{attachment}. Zero is returned if the requested component is not present in \textit{attachment}. If \textit{pname} is \texttt{FRAMEBUFFER\_ATTACHMENT\_COMPONENT\_TYPE}, \textit{param} will contain the format of components of the specified attachment, one of \texttt{FLOAT}, \texttt{INT}, \texttt{UNSIGNED\_INT}, \texttt{UNSIGNED\_NORMALIZED}, or \texttt{INDEX} for floating-point, signed integer, unsigned integer, unsigned fixed-point, or index components respectively. Only color buffers may have index or integer components. If \textit{pname} is \texttt{FRAMEBUFFER\_ATTACHMENT\_COLOR\_ENCODING}, \textit{param} will contain the encoding of components of the specified attachment, one of \texttt{LINEAR} or \texttt{SRGB} for linear or sRGB-encoded components, respectively. Only color buffer components may be sRGB-encoded; such components are treated as described in sections 4.1.8 and 4.1.9. For the default framebuffer, color encoding is determined by the implementation. For framebuffer objects, components are sRGB-encoded if the internal format of a color attachment is one of the color-renderable SRGB formats described in section 3.9.15.

If the value of \texttt{FRAMEBUFFER\_ATTACHMENT\_OBJECT\_TYPE} is \texttt{RENDERBUFFER}, then
6.1. QUERYING GL STATE

- If `pname` is `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME`, `params` will contain the name of the renderbuffer object which contains the attached image.

- Otherwise, an `INVALID_ENUM` error is generated.

If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is `TEXTURE`, then

- If `pname` is `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME`, then `params` will contain the name of the texture object which contains the attached image.

- If `pname` is `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL`, then `params` will contain the mipmap level of the texture object which contains the attached image.

- If `pname` is `FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE` and the texture object named `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` is a cube map texture, then `params` will contain the cube map face of the cube-map texture object which contains the attached image. Otherwise `params` will contain the value zero.

- If `pname` is `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER` and the texture object named `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` is a three-dimensional texture or a one- or two-dimensional array texture, then `params` will contain the number of the texture layer which contains the attached image. Otherwise `params` will contain the value zero.

- Otherwise, an `INVALID_ENUM` error is generated.

For `GetRenderbufferParameteriv`, `target` must be `RENDERBUFFER` and `pname` must be one of the symbolic values in table 6.31. If the renderbuffer currently bound to `target` is zero, then an `INVALID_OPERATION` error is generated.

Upon successful return from `GetRenderbufferParameteriv`, if `pname` is `RENDERBUFFER_WIDTH`, `RENDERBUFFER_HEIGHT`, `RENDERBUFFER_INTERNAL_FORMAT`, or `RENDERBUFFER_SAMPLES`, then `params` will contain the width in pixels, height in pixels, internal format, or number of samples, respectively, of the image of the renderbuffer currently bound to `target`.

If `pname` is `RENDERBUFFER_RED_SIZE`, `RENDERBUFFER_GREEN_SIZE`, `RENDERBUFFER_BLUE_SIZE`, `RENDERBUFFER_ALPHA_SIZE`, `RENDERBUFFER_DEPTH_SIZE`, or `RENDERBUFFER_STENCIL_SIZE`, then `params` will contain the actual resolutions, (not the resolutions specified when the image array was defined), for the red, green, blue, alpha depth, or stencil components, respectively, of the image of the renderbuffer currently bound to `target`.

Otherwise, an `INVALID_ENUM` error is generated.
6.1. Querying GL State

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, and TEXTURE_2D_ARRAY` indicate a one-, two-, or three-dimensional or one- or two-dimensional array 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 3.6, `type` is a pixel type from table 3.5.

Calling `GetTexImage` with a color format (one of `RED, GREEN, BLUE, ALPHA, RG, RGB, BGR, RGBA, BGRA, LUMINANCE, or LUMINANCE_ALPHA`) when the base internal format of the texture image is not a color format; with a format of `DEPTH_COMPONENT` when the base internal format is not `DEPTH_COMPONENT` or `DEPTH_STENCIL`; or with a format of `DEPTH_STENCIL` when the base internal format is not `DEPTH_STENCIL`, causes the error `INVALID_OPERATION`.

`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. No pixel transfer operations are performed on this image, but pixel storage modes that are applicable to `ReadPixels` are applied.

For three-dimensional and two-dimensional array textures, pixel storage oper-
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>ALPHA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Ai</td>
</tr>
<tr>
<td>LUMINANCE (or 1)</td>
<td>Li</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>LUMINANCE_ALPHA (or 2)</td>
<td>Li</td>
<td>0</td>
<td>0</td>
<td>Ai</td>
</tr>
<tr>
<td>INTENSITY</td>
<td>Ii</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RED</td>
<td>Ri</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RG</td>
<td>Ri</td>
<td>Gi</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RGBA (or 3)</td>
<td>Ri</td>
<td>Gi</td>
<td>Bi</td>
<td>1</td>
</tr>
<tr>
<td>RGBA (or 4)</td>
<td>Ri</td>
<td>Gi</td>
<td>Bi</td>
<td>Ai</td>
</tr>
</tbody>
</table>

Table 6.1: Texture, table, and filter return values. \( R_i, G_i, B_i, A_i, L_i, \) and \( I_i \) 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.

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 \( \text{img} \) is not evenly divisible by the number of basic machine units needed to store in memory a FLOAT, UNSIGNED_INT, or UNSIGNED_SHORT respectively, an INVALID_OPERATION error results.

The command

```c
void GetCompressedTexImage( enum target, int lod, 
    void *img );
```

is used to obtain texture images stored in compressed form. The parameters \( \text{target} \), \( \text{lod} \), and \( \text{img} \) are interpreted in the same manner as in \text{GetTexImage}. When called, \text{GetCompressedTexImage} writes \( n \) bytes of compressed image data to the pixel pack buffer or client memory pointed to by \( \text{img} \), where \( n \) is the value of \text{TEXTURE_COMPRESSED_IMAGE_SIZE} for the texture. The compressed image data

Version 3.0 (with changes marked - August 11, 2008)
is formatted according to the definition of the texture’s internal format. All pixel storage and pixel transfer 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

```java
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 Stipple Query

The command

```java
void GetPolygonStipple( void *pattern );
```

obtains the polygon stipple. The pattern is packed into pixel pack buffer or client memory according to the procedure given in section 4.3.2 for `ReadPixels`; it is as if the `height` and `width` passed to that command were both equal to 32, the `type` were `BITMAP`, and the `format` were `COLOR_INDEX`.

### 6.1.6 Color Matrix Query

The scale and bias variables are queried using `GetFloatv` with `pname` set to the appropriate variable name. The top matrix on the color matrix stack is returned by `GetFloatv` called with `pname` set to `COLOR_MATRIX` or `TRANSPOSE_COLOR_MATRIX`. The depth of the color matrix stack, and the maximum depth of the color matrix stack, are queried with `GetIntegerv`, setting `pname` to `COLOR_MATRIX_STACK_DEPTH` and `MAX_COLOR_MATRIX_STACK_DEPTH` respectively.

Version 3.0 (with changes marked - August 11, 2008)
6.1. QUERYING GL STATE

6.1.7 Color Table Query

The current contents of a color table are queried using

```c
void GetColorTable(enum target, enum format, enum type,
                    void *table);
```

target must be one of the regular color table names listed in table 3.4. format and type accept the same values as do the corresponding parameters of GetTexImage, except that a format of DEPTH_COMPONENT causes the error INVALID_ENUM. The one-dimensional color table image is returned to pixel pack buffer or client memory starting at table. No pixel transfer operations are performed on this image, but pixel storage modes that are applicable to ReadPixels are performed. Color components that are requested in the specified format, but which are not included in the internal format of the color lookup table, are returned as zero. The assignments of internal color components to the components requested by format are described in table 6.1.

The functions

```c
void GetColorTableParameter[if]( enum target,
                                 enum pname, T params );
```

are used for integer and floating point query.

target must be one of the regular or proxy color table names listed in table 3.4. pname is one of COLOR_TABLE_SCALE, COLOR_TABLE_BIAS, COLOR_TABLE_FORMAT, COLOR_TABLE_WIDTH, COLOR_TABLE_RED_SIZE, COLOR_TABLE_GREEN_SIZE, COLOR_TABLE_BLUE_SIZE, COLOR_TABLE_ALPHA_SIZE, COLOR_TABLE_LUMINANCE_SIZE, or COLOR_TABLE_INTENSITY_SIZE. The value of the specified parameter is returned in params.

6.1.8 Convolution Query

The current contents of a convolution filter image are queried with the command

```c
void GetConvolutionFilter( enum target, enum format,
                          enum type, void *image );
```

target must be CONVOLUTION_1D or CONVOLUTION_2D. format and type accept the same values as do the corresponding parameters of GetTexImage, except that a format of DEPTH_COMPONENT causes the error INVALID_ENUM. The one-dimensional or two-dimensional images is returned to pixel pack buffer or client memory starting at image.
6.1. QUERYING GL STATE

memory starting at image. Pixel processing and component mapping are identical to those of GetTexImage.

The current contents of a separable filter image are queried using

```c
void GetSeparableFilter( enum target, enum format,
    enum type, void *row, void *column, void *span );
```

`target` must be SEPARABLE_2D. `format` and `type` accept the same values as do the corresponding parameters of GetTexImage. The row and column images are returned to pixel pack buffer or client memory starting at `row` and `column` respectively. `span` is currently unused. Pixel processing and component mapping are identical to those of GetTexImage.

The functions

```c
void GetConvolutionParameter{if}( enum target,
    enum pname, T *params );
```

are used for integer and floating point query. `target` must be CONVOLUTION_1D, CONVOLUTION_2D, or SEPARABLE_2D. `pname` is one of CONVOLUTION_BORDER_COLOR, CONVOLUTION_BORDER_MODE, CONVOLUTION_FILTER_SCALE, CONVOLUTION_FILTER_BIAS, CONVOLUTION_FORMAT, CONVOLUTION_WIDTH, CONVOLUTION_HEIGHT, MAX_CONVOLUTION_WIDTH, or MAX_CONVOLUTION_HEIGHT. The value of the specified parameter is returned in `params`.

6.1.9 Histogram Query

The current contents of the histogram table are queried using

```c
void GetHistogram( enum target, boolean reset,
    enum format, enum type, void *values );
```

`target` must be HISTOGRAM. `type` and `format` accept the same values as do the corresponding parameters of GetTexImage, except that a format of DEPTH_COMPONENT causes the error INVALID_ENUM. The one-dimensional histogram table image is returned to pixel pack buffer or client memory starting at `type`. Pixel processing and component mapping are identical to those of GetTexImage, except that instead of applying the Final Conversion pixel storage mode, component values are simply clamped to the range of the target data type.

If `reset` is TRUE, then all counters of all elements of the histogram are reset to zero. Counters are reset whether returned or not.

No counters are modified if `reset` is FALSE.

Calling
6.1. QUERYING GL STATE

```
void ResetHistogram(enum target);
```

resets all counters of all elements of the histogram table to zero. `target` must be `HISTOGRAM`.

It is not an error to reset or query the contents of a histogram table with zero entries.

The functions

```
void GetHistogramParameter(if)( enum target, enum pname, T params);
```

are used for integer and floating point query. `target` must be `HISTOGRAM` or `PROXY_HISTOGRAM`. `pname` is one of `HISTOGRAM_FORMAT`, `HISTOGRAM_WIDTH`, `HISTOGRAM_RED_SIZE`, `HISTOGRAM_GREEN_SIZE`, `HISTOGRAM_BLUE_SIZE`, `HISTOGRAM_ALPHA_SIZE`, or `HISTOGRAM_LUMINANCE_SIZE`. `pname` may be `HISTOGRAM_SINK` only for `target` `HISTOGRAM`. The value of the specified parameter is returned in `params`.

6.1.10 Minmax Query

The current contents of the minmax table are queried using

```
void GetMinmax(enum target, boolean reset, enum format, enum type, void* values);
```

`target` must be `MINMAX`. `type` and `format` accept the same values as do the corresponding parameters of `GetTexImage`, except that a format of `DEPTH_COMPONENT` causes the error `INVALID_ENUM`. A one-dimensional image of width 2 is returned to pixel pack buffer or client memory starting at `values`. Pixel processing and component mapping are identical to those of `GetTexImage`.

If `reset` is `TRUE`, then each minimum value is reset to the maximum representable value, and each maximum value is reset to the minimum representable value. All values are reset, whether returned or not.

No values are modified if `reset` is `FALSE`.

Calling

```
void ResetMinmax(enum target);
```

resets all minimum and maximum values of `target` to to their maximum and minimum representable values, respectively. `target` must be `MINMAX`.

The functions

Version 3.0 (with changes marked - August 11, 2008)
6.1. QUERYING GL STATE

```c
void GetMinmaxParameter(void)(enum target, enum pname, T params);
```

are used for integer and floating point query. `target` must be `MINMAX`. `pname` is `MINMAX_FORMAT` or `MINMAX_SINK`. The value of the specified parameter is returned in `params`.

### 6.1.11 Pointer and String Queries

The command

```c
void GetPointerv(void)(enum pname, void **params);
```

obtains the pointer or pointers named `pname` in the array `params`. The possible values for `pname` are `SELECTION_BUFFER_POINTER` and `FEEDBACK_BUFFER_POINTER`, which respectively return the pointers set with `SelectBuffer` and `FeedbackBuffer`, and `VERTEX_ARRAY_POINTER`, `NORMAL_ARRAY_POINTER`, `COLOR_ARRAY_POINTER`, `SECONDARY_COLOR_ARRAY_POINTER`, `INDEX_ARRAY_POINTER`, `TEXTURE_COORD_ARRAY_POINTER`, `FOG_COORD_ARRAY_POINTER`, and `EDGE_FLAG_ARRAY_POINTER`, which respectively return the corresponding value stored in the currently bound vertex array object. Each `pname` returns a single pointer value.

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 `VENDOR`, `RENDERER`, `VERSION`, `SHADING_LANGUAGE_VERSION`, and `EXTENSIONS`. The format of the `RENDERER` and `VENDOR` strings is implementation dependent. The `EXTENSIONS` string contains a space separated list of extension names (the extension names themselves do not contain any spaces). The `VERSION` and `SHADING_LANGUAGE_VERSION` strings are laid out as follows:

```
<version number><space><vendor-specific information>
```

1Applications making copies of these static strings should never use a fixed-length buffer, because the strings may grow unpredictably between releases, resulting in buffer overflow when copying. This is particularly true of the `EXTENSIONS` string, which has become extremely long in some GL implementations.

Version 3.0 (with changes marked - August 11, 2008)
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) and the extension names (in the EXTENSIONS string) that can be supported by the current GL context. Thus, if the client and server support different versions and/or extensions, a compatible version and list of extensions is returned.

The GL version 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, and 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

ubyte *GetStringi( enum name, uint index);

target is the name of the indexed state and index is the index of the particular element being queried. target may only be EXTENSIONS, indicating that the extension name corresponding to the indexth supported extension should be returned. index may range from zero to the value of NUM_EXTENSIONS minus one. All extension names, and only the extension names returned in GetString(EXTENSIONS) will be returned as individual names, but there is no defined relationship between the order in which names appear in the non-indexed string and the order in which the appear in the indexed query. There is also 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 target.

### 6.1.12 Asynchronous Queries

The command

boolean IsQuery( uint id);

Version 3.0 (with changes marked - August 11, 2008)
returns \texttt{TRUE} if \textit{id} is the name of a query object. If \textit{id} is zero, or if \textit{id} is a non-zero value that is not the name of a query object, \texttt{IsQuery} returns \texttt{FALSE}.

Information about a query target can be queried with the command

\begin{verbatim}
void GetQueryiv( enum target, enum pname, int *params );
\end{verbatim}

\textit{target} identifies the query target, and must be one of \texttt{SAMPLES\_PASSED} for occlusion queries or \texttt{PRIMITIVES\_GENERATED} and \texttt{TRANSFORM\_FEEDBACK\_PRIMITIVES\_WRITTEN} for primitive queries. If \textit{pname} is \texttt{CURRENT\_QUERY}, the name of the currently active query for \textit{target}, or zero if no query is active, will be placed in \textit{params}.

If \textit{pname} is \texttt{QUERY\_COUNTER\_BITS}, the implementation-dependent number of bits used to hold the query result for \textit{target} will be placed in \textit{params}. The number of query counter bits may be zero, in which case the counter contains no useful information.

For primitive queries (\texttt{PRIMITIVES\_GENERATED} and \texttt{TRANSFORM\_FEEDBACK\_PRIMITIVES\_WRITTEN}) if the number of bits is non-zero, the minimum number of bits allowed is 32.

For occlusion queries (\texttt{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 (\texttt{MAX\_VIEWPORT\_DIMS}). The counter must be able to represent at least two overdraws for every pixel in the viewport. The formula to compute the allowable minimum value (where \textit{n} is the minimum number of bits) is:

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

The state of a query object can be queried with the commands

\begin{verbatim}
void GetQueryObjectiv( uint id, enum pname, int *params );
void GetQueryObjectuiv( uint id, enum pname, uint *params );
\end{verbatim}

If \textit{id} is not the name of a query object, or if the query object named by \textit{id} is currently active, then an \texttt{INVALID\_OPERATION} error is generated.

If \textit{pname} is \texttt{QUERY\_RESULT}, then the query object’s result value is returned as a single integer in \textit{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
6.1. QUERYING GL STATE

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, 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.

6.1.13 Buffer Object Queries

The command

```c
boolean IsBuffer( uint buffer );
```

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

The command

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

queries the data contents of a buffer object. target is ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER, PIXEL_PACK_BUFFER, or PIXEL_UNPACK_BUFFER. 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 `ARRAY_BUFFER`, `ELEMENT_ARRAY_BUFFER`, `PIXEL_PACK_BUFFER`, or `PIXEL_UNPACK_BUFFER` 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 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 `GetIntegerv` 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.

### 6.1.14 Vertex Array Object Queries

The command

```c
boolean IsVertexArray(uint array);
```

returns TRUE if `array` is the name of a vertex array object previously returned by `GenVertexArrays`. 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.

### 6.1.15 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
6.1. QUERYING GL STATE

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 is returned if shader is a vertex shader object, and FRAGMENT_SHADER is returned if shader is a fragment shader object. If pname is DELETE_STATUS, TRUE is returned if the shader has been flagged for deletion and FALSE is returned otherwise. If pname is COMPILER_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, 0 is returned. If \textit{pname} is \texttt{ATTACHED\_SHADERS},
the number of objects attached is returned. If \textit{pname} is \texttt{ACTIVE\_ATTRIBUTES}, the
number of active attributes in \textit{program} is returned. If no active attributes exist,
0 is returned. If \textit{pname} is \texttt{ACTIVE\_ATTRIBUTE\_MAX\_LENGTH}, the length of the
longest active attribute name, including a null terminator, is returned. If no active
attributes exist, 0 is returned. If \textit{pname} is \texttt{ACTIVE\_UNIFORMS}, the number of
active uniforms is returned. If no active uniforms exist, 0 is returned. If \textit{pname}
is \texttt{ACTIVE\_UNIFORM\_MAX\_LENGTH}, the length of the longest active uniform name,
including a null terminator, is returned. If no active uniforms exist, 0 is returned.

If \textit{pname} is \texttt{TRANSFORM\_FEEDBACK\_BUFFER\_MODE}, the buffer mode used when
transform feedback is active is returned. It can be one of \texttt{SEPARATE\_ATTRIBS}
or \texttt{INTERLEAVED\_ATTRIBS}. If \textit{pname} is \texttt{TRANSFORM\_FEEDBACK\_VARYINGS}, the
number of varying variables to capture in transform feedback mode for the pro-
gram is returned. If \textit{pname} is \texttt{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 feed-
back, zero is returned.

The command

\begin{verbatim}
void GetAttachedShaders( uint program, sizei maxCount,
                        sizei *count, uint *shaders );
\end{verbatim}

returns the names of shader objects attached to \textit{program} in \textit{shaders}. The actual
number of shader names written into \textit{shaders} is returned in \textit{count}. If no shaders are
attached, \textit{count} is set to zero. If \textit{count} is \texttt{NULL} then it is ignored. The maximum
number of shader names that may be written into \textit{shaders} is specified by \textit{maxCount}.
The number of objects attached to \textit{program} is given by can be queried by calling
\texttt{GetProgramiv} with \texttt{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
\textit{info log}, can be obtained with the commands

\begin{verbatim}
void GetShaderInfoLog( uint shader, sizei bufSize,
                      sizei *length, char *infoLog );
void GetProgramInfoLog( uint program, sizei bufSize,
                        sizei *length, char *infoLog );
\end{verbatim}

These commands return the info log string in \textit{infoLog}. This string will be null
terminated. The actual number of characters written into \textit{infoLog}, excluding the
null terminator, is returned in \textit{length}. If \textit{length} is \texttt{NULL}, then no length is returned.
The maximum number of characters that may be written into \texttt{infoLog}, including the null terminator, is specified by \texttt{bufSize}. The number of characters in the info log can be queried with \texttt{GetShaderiv} or \texttt{GetProgramiv} with \texttt{INFO\_LOG\_LENGTH}. If \texttt{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 \texttt{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

\begin{verbatim}
void GetShaderSource( uint shader, sizei bufsize, 
                      sizei *length, char *source );
\end{verbatim}

returns in \texttt{source} the string making up the source code for the shader object \texttt{shader}. The string \texttt{source} will be null terminated. The actual number of characters written into \texttt{source}, excluding the null terminator, is returned in \texttt{length}. If \texttt{length} is null, no length is returned. The maximum number of characters that may be written into \texttt{source}, including the null terminator, is specified by \texttt{bufSize}. The string \texttt{source} is a concatenation of the strings passed to the GL using \texttt{ShaderSource}. The length of this concatenation is given by \texttt{SHADER\_SOURCE\_LENGTH}, which can be queried with \texttt{GetShaderiv}.

The commands

\begin{verbatim}
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 );
\end{verbatim}

obtain the vertex attribute state named by \texttt{pname} for the generic vertex attribute numbered \texttt{index} and places the information in the array \texttt{params}. \texttt{pname} must be one of

Version 3.0 (with changes marked - August 11, 2008
6.1. QUERYING GL STATE

The queries except CURRENT_VERTEX_ATTRIB return values stored in the currently bound vertex array object (the value of VERTEX_ATTRIB_ARRAY_BINDING). If the zero object is bound, these values are client state. The error INVALID_VALUE is generated if index is greater than or equal to MAX_VERTEX_ATTRIBS.

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

The query CURRENT_VERTEX_ATTRIB returns the current value for the generic attribute index. GetVertexAttribdv and GetVertexAttribfv read and return the current attribute values as floating-point values; GetVertexAttribiv reads them as floating-point values and converts them to integer values; GetVertexAttribIiv reads and returns them as integers; 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 error INVALID_OPERATION is generated if index is zero, as there is no current value for generic attribute zero.

The command

```c
void GetVertexAttribPointerv( uint index, enum pname, 
   void **pointer);
```

obtains the pointer named pname for the vertex attribute numbered index and places the information in the array pointer. pname must be VERTEX_ATTRIB_ARRAY_POINTER. The value returned is queried from the currently bound vertex array object. If the zero object is bound, the value is queried from client state. An INVALID_VALUE error is generated if index is greater than or equal to the value of MAX_VERTEX_ATTRIBS.

The commands

```c
void GetUniformfv( uint program, int location, 
   float *params);
```

Version 3.0 (with changes marked - August 11, 2008
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` 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, the return parameter `params` will be unmodified.

framebuffer_object

6.1.16 Framebuffer Object Queries

The command

```c
boolean IsFramebuffer(uint framebuffer);
```

returns `TRUE` if `framebuffer` is the name of a framebuffer object. If `framebuffer` is zero, or if `framebuffer` is a non-zero value that is not the name of a framebuffer object, `IsFramebuffer` return `FALSE`.

6.1.17 Renderbuffer Object Queries

The command

```c
boolean IsRenderbuffer(uint renderbuffer);
```

returns `TRUE` if `renderbuffer` is the name of a renderbuffer object. If `renderbuffer` is zero, or if `renderbuffer` is a non-zero value that is not the name of a renderbuffer object, `IsRenderbuffer` return `FALSE`.

6.1.18 Saving and Restoring State

Besides providing a means to obtain the values of state variables, the GL also provides a means to save and restore groups of state variables. The `PushAttrib`, `PushClientAttrib`, `PopAttrib` and `PopClientAttrib` commands are used for this purpose. The commands

```c
void PushAttrib(bitfield mask);
```
6.1. QUERYING GL STATE

```c
void PushClientAttrib(bitfield mask);
```

take a bitwise OR of symbolic constants indicating which groups of state variables to push onto an attribute stack. **PushAttrib** uses a server attribute stack while **PushClientAttrib** uses a client attribute stack. Each constant refers to a group of state variables. The classification of each variable into a group is indicated in the following tables of state variables. The error **STACK_OVERFLOW** is generated if **PushAttrib** or **PushClientAttrib** is executed while the corresponding stack depth is **MAX_ATTRIB_STACKDEPTH** or **MAX_CLIENT_ATTRIB_STACKDEPTH** respectively. Bits set in `mask` that do not correspond to an attribute group are ignored. The special `mask` values **ALL_ATTRIB_BITS** and **CLIENT_ALL_ATTRIB_BITS** may be used to push all stackable server and client state, respectively.

The commands

```c
void PopAttrib(void);
void PopClientAttrib(void);
```
reset the values of those state variables that were saved with the last corresponding **PushAttrib** or **PopClientAttrib**. Those not saved remain unchanged. The error **STACK_UNDERFLOW** is generated if **PopAttrib** or **PopClientAttrib** is executed while the respective stack is empty.

Table 6.2 shows the attribute groups with their corresponding symbolic constant names and stacks.

When **PushAttrib** is called with `TEXTURE_BIT` set, the priorities, border colors, filter modes, wrap modes, and other state of the currently bound texture objects (see table 6.20), as well as the current texture bindings and enables, are pushed onto the attribute stack. (Unbound texture objects are not pushed or restored.) When an attribute set that includes texture information is popped, the bindings and enables are first restored to their pushed values, then the bound texture object’s parameters are restored to their pushed values.

Operations on attribute groups push or pop texture state within that group for all texture units. When state for a group is pushed, all state corresponding to `TEXTURE0` is pushed first, followed by state corresponding to `TEXTURE1`, and so on up to and including the state corresponding to `TEXTUREk` where `k + 1` is the value of `MAX_TEXTURE_UNITS`. When state for a group is popped, texture state is restored in the opposite order that it was pushed, starting with state corresponding to `TEXTUREk` and ending with `TEXTURE0`. Identical rules are observed for client texture state push and pop operations. Matrix stacks are never pushed or popped with **PushAttrib**, **PushClientAttrib**, **PopAttrib**, or **PopClientAttrib**.

The depth of each attribute stack is implementation dependent but must be at least 16. The state required for each attribute stack is potentially 16 copies of each
6.1. QUERYING GL STATE

<table>
<thead>
<tr>
<th>Stack</th>
<th>Attribute</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>server</td>
<td>accum-buffer</td>
<td>ACCUM_BUFFER_BIT</td>
</tr>
<tr>
<td>server</td>
<td>color-buffer</td>
<td>COLOR_BUFFER_BIT</td>
</tr>
<tr>
<td>server</td>
<td>current</td>
<td>CURRENT_BIT</td>
</tr>
<tr>
<td>server</td>
<td>depth-buffer</td>
<td>DEPTH_BUFFER_BIT</td>
</tr>
<tr>
<td>server</td>
<td>enable</td>
<td>ENABLE_BIT</td>
</tr>
<tr>
<td>server</td>
<td>eval</td>
<td>EVAL_BIT</td>
</tr>
<tr>
<td>server</td>
<td>fog</td>
<td>FOG_BIT</td>
</tr>
<tr>
<td>server</td>
<td>hint</td>
<td>HINT_BIT</td>
</tr>
<tr>
<td>server</td>
<td>lighting</td>
<td>LIGHTING_BIT</td>
</tr>
<tr>
<td>server</td>
<td>line</td>
<td>LINE_BIT</td>
</tr>
<tr>
<td>server</td>
<td>list</td>
<td>LIST_BIT</td>
</tr>
<tr>
<td>server</td>
<td>multisample</td>
<td>MULTISAMPLE_BIT</td>
</tr>
<tr>
<td>server</td>
<td>pixel</td>
<td>PIXEL_MODE_BIT</td>
</tr>
<tr>
<td>server</td>
<td>point</td>
<td>POINT_BIT</td>
</tr>
<tr>
<td>server</td>
<td>polygon</td>
<td>POLYGON_BIT</td>
</tr>
<tr>
<td>server</td>
<td>polygon-stipple</td>
<td>POLYGON_STIPPLE_BIT</td>
</tr>
<tr>
<td>server</td>
<td>scissor</td>
<td>SCISSOR_BIT</td>
</tr>
<tr>
<td>server</td>
<td>stencil-buffer</td>
<td>STENCIL_BUFFER_BIT</td>
</tr>
<tr>
<td>server</td>
<td>texture</td>
<td>TEXTURE_BIT</td>
</tr>
<tr>
<td>server</td>
<td>transform</td>
<td>TRANSFORM_BIT</td>
</tr>
<tr>
<td>server</td>
<td>viewport</td>
<td>VIEWPORT_BIT</td>
</tr>
<tr>
<td>server</td>
<td></td>
<td>ALL_ATTRIB_BITS</td>
</tr>
<tr>
<td>client</td>
<td>vertex-array</td>
<td>CLIENT_VERTEX_ARRAY_BIT</td>
</tr>
<tr>
<td>client</td>
<td>pixel-store</td>
<td>CLIENT_PIXEL_STORE_BIT</td>
</tr>
<tr>
<td>client</td>
<td>select</td>
<td>can’t be pushed or pop’d</td>
</tr>
<tr>
<td>client</td>
<td>feedback</td>
<td>can’t be pushed or pop’d</td>
</tr>
<tr>
<td>client</td>
<td></td>
<td>CLIENT_ALL_ATTRIB_BITS</td>
</tr>
</tbody>
</table>

Table 6.2: Attribute groups

Version 3.0 (with changes marked - August 11, 2008)
state variable, 16 masks indicating which groups of variables are stored in each stack entry, and an attribute stack pointer. In the initial state, both attribute stacks are empty.

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 all matrices, where only the top entry on the stack is returned; with clip planes, where only the selected clip plane is returned, with parameters describing lights, where only the value pertaining to the selected light is returned; with textures, where only the selected texture or texture parameter is returned; and with evaluator maps, where only the selected map is returned. Finally, a “−” in the attribute column indicates that the indicated value is not included in any attribute group (and thus can not be pushed or popped with PushAttrib, PushClientAttrib, PopAttrib, or PopClientAttrib).

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 only by using that command.

State table entries which are required only by the imaging subset (see section 3.7.2) are typeset against a gray background.
### Table 6.3: State Variable Types

<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>CI</td>
<td>Color index (floating-point index value)</td>
</tr>
<tr>
<td>T</td>
<td>Texture coordinates (floating-point ((s, t, r, q)) values)</td>
</tr>
<tr>
<td>N</td>
<td>Normal coordinates (floating-point ((x, y, z)) values)</td>
</tr>
<tr>
<td>V</td>
<td>Vertex, including associated data</td>
</tr>
<tr>
<td>Z</td>
<td>Integer</td>
</tr>
<tr>
<td>Z⁺</td>
<td>Non-negative integer</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>P</td>
<td>Position (((x, y, z, w)) floating-point coordinates)</td>
</tr>
<tr>
<td>D</td>
<td>Direction (((x, y, z)) floating-point coordinates)</td>
</tr>
<tr>
<td>M⁴</td>
<td>(4 \times 4) floating-point matrix</td>
</tr>
<tr>
<td>S</td>
<td>NULL-terminated string</td>
</tr>
<tr>
<td>I</td>
<td>Image</td>
</tr>
<tr>
<td>A</td>
<td>Attribute stack entry, including mask</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>

Version 3.0 (with changes marked - August 11, 2008)
### 6.2. STATE TABLES

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Type</th>
<th>Initial Value</th>
<th>Get value</th>
<th>Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Z11</td>
<td>V</td>
<td>-</td>
<td>-</td>
<td>Previous vertex in Begin/End line</td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>B</td>
<td>-</td>
<td>-</td>
<td>Indicates first vertex of a Begin/End line</td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>V</td>
<td>-</td>
<td>-</td>
<td>Line stipple counter</td>
<td></td>
</tr>
<tr>
<td>Z1</td>
<td>Z1</td>
<td>Z1</td>
<td>Z1</td>
<td>Vertices inside of Begin/End polygon</td>
<td></td>
</tr>
<tr>
<td>Z1</td>
<td>V</td>
<td>Z1</td>
<td>Z1</td>
<td>Previous two vertices in a Begin/End triangle strip</td>
<td></td>
</tr>
<tr>
<td>Z1</td>
<td>2×V</td>
<td>2×V</td>
<td>2×V</td>
<td>Number of polygon-vertices</td>
<td></td>
</tr>
<tr>
<td>Z1</td>
<td>V</td>
<td>V</td>
<td>V</td>
<td>Number of vertices so far in triangle strip: 0, 1, or more</td>
<td></td>
</tr>
<tr>
<td>Z1</td>
<td>3×V</td>
<td>3×V</td>
<td>3×V</td>
<td>Triangle strip A/B vertex pointer</td>
<td></td>
</tr>
<tr>
<td>Z1</td>
<td>Z1</td>
<td>Z1</td>
<td>Z1</td>
<td>Vertices of the quad under construction</td>
<td></td>
</tr>
<tr>
<td>Z1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Number of vertices so far in quad strip: 0, 1, 2, or more</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4. GL Internal begin-end state variables (inaccessible)
### Table 6.5. Current Values and Associated Data

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT_COLOR</td>
<td>C</td>
<td>GetFloatv</td>
<td>1,1,1,1</td>
<td>Current color</td>
<td>2.7</td>
<td>current</td>
</tr>
<tr>
<td>CURRENT_SECONDARY_COLOR</td>
<td>C</td>
<td>GetFloatv</td>
<td>0,0,0,1</td>
<td>Current secondary color</td>
<td>2.7</td>
<td>current</td>
</tr>
<tr>
<td>CURRENT_INDEX</td>
<td>CI</td>
<td>GetIntegerv</td>
<td>1</td>
<td>Current color index</td>
<td>2.7</td>
<td>current</td>
</tr>
<tr>
<td>CURRENT_TEXTURE_COORDS</td>
<td>2 * T</td>
<td>GetFloatv</td>
<td>0,0,0,1</td>
<td>Current texture coordinates</td>
<td>2.7</td>
<td>current</td>
</tr>
<tr>
<td>CURRENT_NORMAL</td>
<td>N</td>
<td>GetFloatv</td>
<td>0,0,1</td>
<td>Current normal</td>
<td>2.7</td>
<td>current</td>
</tr>
<tr>
<td>CURRENT_FOG_COORD</td>
<td>R</td>
<td>GetFloatv</td>
<td>0</td>
<td>Current fog coordinate</td>
<td>2.7</td>
<td>current</td>
</tr>
<tr>
<td>-</td>
<td>C</td>
<td>–</td>
<td>–</td>
<td>Color associated with last vertex</td>
<td>2.6</td>
<td>–</td>
</tr>
<tr>
<td>-</td>
<td>CI</td>
<td>–</td>
<td>–</td>
<td>Color index associated with last ver-</td>
<td>2.6</td>
<td>–</td>
</tr>
<tr>
<td>-</td>
<td>T</td>
<td>–</td>
<td>–</td>
<td>Texture coordinates associated with last vertex</td>
<td>2.6</td>
<td>–</td>
</tr>
<tr>
<td>CURRENT_RASTER_POSITION</td>
<td>R⁺</td>
<td>GetFloatv</td>
<td>0,0,1</td>
<td>Current raster position</td>
<td>2.18</td>
<td>current</td>
</tr>
<tr>
<td>CURRENT_RASTER_DISTANCE</td>
<td>R⁺</td>
<td>GetFloatv</td>
<td>0</td>
<td>Current raster distance</td>
<td>2.18</td>
<td>current</td>
</tr>
<tr>
<td>CURRENT_RASTER_COLOR</td>
<td>C</td>
<td>GetFloatv</td>
<td>1,1,1,1</td>
<td>Color associated with raster position</td>
<td>2.18</td>
<td>current</td>
</tr>
<tr>
<td>CURRENT_RASTER_SECONDARY_COLOR</td>
<td>C</td>
<td>GetFloatv</td>
<td>0,0,0,1</td>
<td>Secondary color associated with raster position</td>
<td>2.18</td>
<td>current</td>
</tr>
<tr>
<td>CURRENT_RASTER_INDEX</td>
<td>CI</td>
<td>GetIntegerv</td>
<td>1</td>
<td>Color index associated with raster position</td>
<td>2.18</td>
<td>current</td>
</tr>
<tr>
<td>CURRENT_RASTER_TEXTURE_COORDS</td>
<td>2 * T</td>
<td>GetFloatv</td>
<td>0,0,0,1</td>
<td>Texture coordinates associated with raster position</td>
<td>2.18</td>
<td>current</td>
</tr>
<tr>
<td>CURRENT_RASTER_POSITION_VALID</td>
<td>B</td>
<td>GetBooleanv</td>
<td>TRUE</td>
<td>Raster position valid bit</td>
<td>2.18</td>
<td>current</td>
</tr>
<tr>
<td>EDGE_FLAG</td>
<td>B</td>
<td>GetBooleanv</td>
<td>TRUE</td>
<td>Edge flag</td>
<td>2.6</td>
<td>2</td>
</tr>
</tbody>
</table>

**Version 3.0 (with changes marked - August 11, 2008**
<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>VERTEX.ARRAY</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Vertex array enable</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>VERTEX.ARRAY,SIZE</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>4</td>
<td>Coordinates per vertex</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>VERTEX.ARRAY,TYPE</td>
<td>Z4</td>
<td>GetIntegerv</td>
<td>FLOAT</td>
<td>Type of vertex coordinates</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>VERTEX.ARRAY,STRIDE</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Stride between vertices</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>VERTEX.ARRAY,POINTER</td>
<td>Y</td>
<td>GetPointerv</td>
<td>0</td>
<td>Pointer to the vertex array</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>NORMAL.ARRAY</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Normal array enable</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>NORMAL.ARRAY,TYPE</td>
<td>Z5</td>
<td>GetIntegerv</td>
<td>FLOAT</td>
<td>Type of normal coordinates</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>NORMAL.ARRAY,STRIDE</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Stride between normals</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>NORMAL.ARRAY,POINTER</td>
<td>Y</td>
<td>GetPointerv</td>
<td>0</td>
<td>Pointer to the normal array</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>FOG.COORD.ARRAY</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Fog coord array enable</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>FOG.COORD.ARRAY,TYPE</td>
<td>Z2</td>
<td>GetIntegerv</td>
<td>FLOAT</td>
<td>Type of fog coord components</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>FOG.COORD.ARRAY,STRIDE</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Stride between fog coords</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>FOG.COORD.ARRAY,POINTER</td>
<td>Y</td>
<td>GetPointerv</td>
<td>0</td>
<td>Pointer to the fog coord array</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>COLOR.ARRAY</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Color array enable</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>COLOR.ARRAY,SIZE</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>4</td>
<td>Color components per vertex</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>COLOR.ARRAY,TYPE</td>
<td>Z8</td>
<td>GetIntegerv</td>
<td>FLOAT</td>
<td>Type of color components</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>COLOR.ARRAY,STRIDE</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Stride between colors</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>COLOR.ARRAY,POINTER</td>
<td>Y</td>
<td>GetPointerv</td>
<td>0</td>
<td>Pointer to the color array</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Initial Value</td>
<td>Description</td>
<td>Sec.</td>
<td>Attribute</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------</td>
<td>-------------</td>
<td>---------------</td>
<td>-------------------------------------</td>
<td>------</td>
<td>-----------</td>
</tr>
<tr>
<td>SECONDARY_COLOR_ARRAY</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Secondary color array enable</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>SECONDARY_COLOR_ARRAY_SIZE</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>3</td>
<td>Secondary color components per vertex</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>SECONDARY_COLOR_ARRAY_TYPE</td>
<td>Z_4</td>
<td>GetIntegerv</td>
<td>FLOAT</td>
<td>Type of secondary color components</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>SECONDARY_COLOR_ARRAY_STRIDE</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Stride between secondary colors</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>SECONDARY_COLOR_ARRAY_POINTER</td>
<td>Y</td>
<td>GetPointerv</td>
<td>0</td>
<td>Pointer to the secondary color array</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>INDEX_ARRAY</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Index array enable</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>INDEX_ARRAY_TYPE</td>
<td>Z_4</td>
<td>GetIntegerv</td>
<td>FLOAT</td>
<td>Type of indices</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>INDEX_ARRAY_STRIDE</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Stride between indices</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>INDEX_ARRAY_POINTER</td>
<td>Y</td>
<td>GetPointerv</td>
<td>0</td>
<td>Pointer to the index array</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>TEXTURE_COORD_ARRAY</td>
<td>2*B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Texture coordinate array enable</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>TEXTURE_COORD_ARRAY_SIZE</td>
<td>2*Z+</td>
<td>GetIntegerv</td>
<td>4</td>
<td>Coordinates per element</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>TEXTURE_COORD_ARRAY_TYPE</td>
<td>2*Z_4</td>
<td>GetIntegerv</td>
<td>FLOAT</td>
<td>Type of texture coordinates</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>TEXTURE_COORD_ARRAY_STRIDE</td>
<td>2*Z+</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Stride between texture coordinates</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
<tr>
<td>TEXTURE_COORD_ARRAY_POINTER</td>
<td>Y</td>
<td>GetPointerv</td>
<td>0</td>
<td>Pointer to the texture coordinate array</td>
<td>2.8</td>
<td>vertex-array</td>
</tr>
</tbody>
</table>
### Table 6.8. Vertex Array Object State (cont.)

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Command</th>
<th>Type</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetVertexAttrib</td>
<td>GetVertexAttrib</td>
<td>16 \times Z</td>
<td>FALSE</td>
<td>Vertex attr array enable</td>
<td>2.8</td>
</tr>
<tr>
<td>GetVertexAttrib</td>
<td>GetVertexAttrib</td>
<td>16 \times Z</td>
<td>FALSE</td>
<td>Vertex attr array size</td>
<td>2.8</td>
</tr>
<tr>
<td>GetVertexAttrib</td>
<td>GetVertexAttrib</td>
<td>16 \times Z</td>
<td>FALSE</td>
<td>Vertex attr array stride</td>
<td>2.8</td>
</tr>
<tr>
<td>GetVertexAttrib</td>
<td>GetVertexAttrib</td>
<td>16 \times Z</td>
<td>FALSE</td>
<td>Vertex attr array type</td>
<td>2.8</td>
</tr>
<tr>
<td>GetVertexAttrib</td>
<td>GetVertexAttrib</td>
<td>16 \times Z</td>
<td>FALSE</td>
<td>Vertex attr array normalized</td>
<td>2.8</td>
</tr>
<tr>
<td>GetVertexAttrib</td>
<td>GetVertexAttrib</td>
<td>16 \times Z</td>
<td>FALSE</td>
<td>Vertex attr array has unconverted integers</td>
<td>2.8</td>
</tr>
<tr>
<td>GetVertexAttribPointer</td>
<td>GetVertexAttribPointer</td>
<td>16 \times Z</td>
<td>NULL</td>
<td>Vertex attr array pointer</td>
<td>2.8</td>
</tr>
<tr>
<td>GetVertexAttribPointer</td>
<td>GetVertexAttribPointer</td>
<td>16 \times Z</td>
<td>TRUE</td>
<td>Edge flag array enable</td>
<td>2.8</td>
</tr>
<tr>
<td>GetVertexAttribPointer</td>
<td>GetVertexAttribPointer</td>
<td>16 \times Z</td>
<td>0</td>
<td>Stride between edge flags</td>
<td>2.8</td>
</tr>
<tr>
<td>GetVertexAttribPointer</td>
<td>GetVertexAttribPointer</td>
<td>16 \times Z</td>
<td>0</td>
<td>Pointer to the edge flag array</td>
<td>2.8</td>
</tr>
</tbody>
</table>
### Table 6.9. 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>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>VERTEX_ARRAY_BUFFER_BINDING</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Vertex array buffer binding</td>
<td>2.9</td>
<td>vertex-array</td>
</tr>
<tr>
<td>NORMAL_ARRAY_BUFFER_BINDING</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Normal array buffer binding</td>
<td>2.9</td>
<td>vertex-array</td>
</tr>
<tr>
<td>COLOR_ARRAY_BUFFER_BINDING</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Color array buffer binding</td>
<td>2.9</td>
<td>vertex-array</td>
</tr>
<tr>
<td>INDEX_ARRAY_BUFFER_BINDING</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Index array buffer binding</td>
<td>2.9</td>
<td>vertex-array</td>
</tr>
<tr>
<td>TEXTURE_COORD_ARRAY_BUFFER_BINDING</td>
<td>$2 \times Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Texcoord array buffer binding</td>
<td>2.9</td>
<td>vertex-array</td>
</tr>
<tr>
<td>EDGE_FLAG_ARRAY_BUFFER_BINDING</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Edge flag array buffer binding</td>
<td>2.9</td>
<td>vertex-array</td>
</tr>
<tr>
<td>SECONDARY_COLOR_ARRAY_BUFFER_BINDING</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Secondary color array buffer binding</td>
<td>2.9</td>
<td>vertex-array</td>
</tr>
<tr>
<td>FOG_COORD_ARRAY_BUFFER_BINDING</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Fog coordinate array buffer binding</td>
<td>2.9</td>
<td>vertex-array</td>
</tr>
<tr>
<td>ELEMENT_ARRAY_BUFFER_BINDING</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Element array buffer binding</td>
<td>2.9</td>
<td>vertex-array</td>
</tr>
<tr>
<td>VERTEX_ATTRIB_ARRAY_BUFFER_BINDING</td>
<td>$16 \times Z^+$</td>
<td>GetVertexAttribiv</td>
<td>0</td>
<td>Attribute array buffer binding</td>
<td>2.9</td>
<td>vertex-array</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>
<td>vertex-array</td>
</tr>
<tr>
<td>Get command</td>
<td>Type</td>
<td>Initial value</td>
<td>Description</td>
<td>Sec.</td>
<td>Attribute</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>------</td>
<td>---------------</td>
<td>-------------</td>
<td>------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>client active texture unit selector</td>
<td>TEXTURE</td>
<td>2</td>
<td>Current buffer binding</td>
<td>2.7</td>
<td>vertex array</td>
<td></td>
</tr>
<tr>
<td>Current buffer binding</td>
<td>ARRAY BUFFER BINDING</td>
<td>0</td>
<td></td>
<td>2.9</td>
<td>vertex array</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.10. Vertex Array Data (not in Vertex Array objects)
<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
<th>Attribute</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>
<td>-</td>
</tr>
<tr>
<td>BUFFER_SIZE</td>
<td>$n \times Z^+$</td>
<td>GetBufferParameteriv</td>
<td>0</td>
<td>Buffer data size</td>
<td>2.9</td>
<td>-</td>
</tr>
<tr>
<td>BUFFER_USAGE</td>
<td>$n \times Z^9$</td>
<td>GetBufferParameteriv</td>
<td>STATIC_DRAW</td>
<td>Buffer usage pattern</td>
<td>2.9</td>
<td>-</td>
</tr>
<tr>
<td>BUFFER_ACCESS</td>
<td>$n \times Z^3$</td>
<td>GetBufferParameteriv</td>
<td>READ_WRITE</td>
<td>Buffer access flag</td>
<td>2.9</td>
<td>-</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>
<td>-</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>
<td>-</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Initial Value</td>
<td>Description</td>
<td>Sec.</td>
<td>Attribute</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>-------------</td>
<td>---------------</td>
<td>-------------</td>
<td>------</td>
<td>-----------</td>
</tr>
<tr>
<td>COLOR_MATRIX (TRANSPOSE_COLOR_MATRIX)</td>
<td>(2 \times M^4)</td>
<td>GetFloatv</td>
<td>Identity</td>
<td>Color matrix stack</td>
<td>3.7.3</td>
<td>−</td>
</tr>
<tr>
<td>MODELVIEW_MATRIX (TRANSPOSE_MODELVIEW_MATRIX)</td>
<td>(32 \times M^4)</td>
<td>GetFloatv</td>
<td>Identity</td>
<td>Model-view matrix stack</td>
<td>2.12.2</td>
<td>−</td>
</tr>
<tr>
<td>PROJECTION_MATRIX (TRANSPOSE_PROJECTION_MATRIX)</td>
<td>(2 \times M^4)</td>
<td>GetFloatv</td>
<td>Identity</td>
<td>Projection matrix stack</td>
<td>2.12.2</td>
<td>−</td>
</tr>
<tr>
<td>TEXTURE_MATRIX (TRANSPOSE_TEXTURE_MATRIX)</td>
<td>(2 \times 2 \times M^4)</td>
<td>GetFloatv</td>
<td>Identity</td>
<td>Texture matrix stack</td>
<td>2.12.2</td>
<td>−</td>
</tr>
<tr>
<td>VIEWPORT</td>
<td>(4 \times Z)</td>
<td>GetIntegerv</td>
<td>see 2.12.1</td>
<td>Viewport origin &amp; extent</td>
<td>2.12.1</td>
<td>viewport</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>2.12.1</td>
<td>viewport</td>
</tr>
<tr>
<td>COLOR_MATRIX_STACK_DEPTH</td>
<td>(Z^+)</td>
<td>GetIntegerv</td>
<td>1</td>
<td>Color matrix stack pointer</td>
<td>3.7.3</td>
<td>−</td>
</tr>
<tr>
<td>MODELVIEW_STACK_DEPTH</td>
<td>(Z^+)</td>
<td>GetIntegerv</td>
<td>1</td>
<td>Model-view matrix stack pointer</td>
<td>2.12.2</td>
<td>−</td>
</tr>
<tr>
<td>PROJECTION_STACK_DEPTH</td>
<td>(Z^+)</td>
<td>GetIntegerv</td>
<td>1</td>
<td>Projection matrix stack pointer</td>
<td>2.12.2</td>
<td>−</td>
</tr>
<tr>
<td>TEXTURE_STACK_DEPTH</td>
<td>(2 \times Z^+)</td>
<td>GetIntegerv</td>
<td>1</td>
<td>Texture matrix stack pointer</td>
<td>2.12.2</td>
<td>−</td>
</tr>
<tr>
<td>MATRIX_MODE</td>
<td>(Z_4)</td>
<td>GetIntegerv</td>
<td>MODELVIEW</td>
<td>Current matrix mode</td>
<td>2.12.2</td>
<td>transform</td>
</tr>
<tr>
<td>NORMALIZE</td>
<td>(B)</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Current normal normalization on/off</td>
<td>2.12.3</td>
<td>transform/enable</td>
</tr>
<tr>
<td>RESCALE_NORMAL</td>
<td>(B)</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Current normal rescaling on/off</td>
<td>2.12.3</td>
<td>transform/enable</td>
</tr>
<tr>
<td>CLIP_PLANE:</td>
<td>(6 \times R^4)</td>
<td>GetClipPlane</td>
<td>0,0,0,0</td>
<td>User clipping plane coefficients</td>
<td>2.17</td>
<td>transform</td>
</tr>
<tr>
<td>CLIP_PLANE:</td>
<td>(6 \times B)</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>ith user clipping plane enabled</td>
<td>2.17</td>
<td>transform/enable</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Initial Value</td>
<td>Description</td>
<td>Sec.</td>
<td>Attribute</td>
</tr>
<tr>
<td>------------------</td>
<td>------</td>
<td>-------------</td>
<td>---------------</td>
<td>--------------------------------</td>
<td>------</td>
<td>---------------</td>
</tr>
<tr>
<td>FOG_COLOR</td>
<td>C</td>
<td>GetFloatv</td>
<td>0,0,0,0</td>
<td>Fog color</td>
<td>3.11</td>
<td>fog</td>
</tr>
<tr>
<td>FOG_INDEX</td>
<td>CI</td>
<td>GetFloatv</td>
<td>0</td>
<td>Fog index</td>
<td>3.11</td>
<td>fog</td>
</tr>
<tr>
<td>FOG_DENSITY</td>
<td>R</td>
<td>GetFloatv</td>
<td>1.0</td>
<td>Exponential fog density</td>
<td>3.11</td>
<td>fog</td>
</tr>
<tr>
<td>FOG_START</td>
<td>R</td>
<td>GetFloatv</td>
<td>0.0</td>
<td>Linear fog start</td>
<td>3.11</td>
<td>fog</td>
</tr>
<tr>
<td>FOG_END</td>
<td>R</td>
<td>GetFloatv</td>
<td>1.0</td>
<td>Linear fog end</td>
<td>3.11</td>
<td>fog</td>
</tr>
<tr>
<td>FOG_MODE</td>
<td>Z_3</td>
<td>GetIntegerv</td>
<td>EXP</td>
<td>Fog mode</td>
<td>3.11</td>
<td>fog</td>
</tr>
<tr>
<td>FOG</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>True if fog enabled</td>
<td>3.11</td>
<td>fog/enable</td>
</tr>
<tr>
<td>FOG_COORD_SRC</td>
<td>Z_2</td>
<td>GetIntegerv</td>
<td>FRAGMENT_DEPTH</td>
<td>Source of coordinate for fog calculation</td>
<td>3.11</td>
<td>fog</td>
</tr>
<tr>
<td>COLOR_SUM</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>True if color sum enabled</td>
<td>3.10</td>
<td>fog/enable</td>
</tr>
<tr>
<td>SHADE_MODEL</td>
<td>Z</td>
<td>GetIntegerv</td>
<td>SMOOTH</td>
<td>ShadeModel setting</td>
<td>2.19.7</td>
<td>lighting</td>
</tr>
<tr>
<td>CLAMP_VERTEX_COLOR</td>
<td>Z_3</td>
<td>GetIntegerv</td>
<td>TRUE</td>
<td>Vertex color clamping</td>
<td>2.19.6</td>
<td>lighting/enable</td>
</tr>
<tr>
<td>CLAMP_FRAGMENT_COLOR</td>
<td>Z_3</td>
<td>GetIntegerv</td>
<td>FIXED.ONLY</td>
<td>Fragment color clamping</td>
<td>3.8</td>
<td>color-buffer/enable</td>
</tr>
<tr>
<td>CLAMP_READ_COLOR</td>
<td>Z_3</td>
<td>GetIntegerv</td>
<td>FIXED.ONLY</td>
<td>Read color clamping</td>
<td>4.2</td>
<td>color-buffer/enable</td>
</tr>
<tr>
<td>Section</td>
<td>Command</td>
<td>Initial Value</td>
<td>Description</td>
<td>Type</td>
<td>Get value</td>
<td>Get value Parameter</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>---------------</td>
<td>-------------</td>
<td>------</td>
<td>-----------</td>
<td>---------------------</td>
</tr>
<tr>
<td>2.19.1</td>
<td>lighting/enable</td>
<td>FALSE</td>
<td>True if lighting is enabled</td>
<td>GetEnabled</td>
<td>LIGHTING</td>
<td>B</td>
</tr>
<tr>
<td>2.19.3</td>
<td>lighting/enable</td>
<td>FALSE</td>
<td>True if color tracking is enabled</td>
<td>GetEnabled</td>
<td>LIGHTING</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Material properties</td>
<td>GetMaterial</td>
<td>LIGHTING</td>
<td>ambient</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Tracking current color</td>
<td>GetMaterial</td>
<td>LIGHTING</td>
<td>diffuse</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Face(s) affected by color tracking</td>
<td>GetMaterial</td>
<td>LIGHTING</td>
<td>specular</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Emissive material color</td>
<td>GetMaterial</td>
<td>LIGHTING</td>
<td>emissive</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Specular material color</td>
<td>GetMaterial</td>
<td>LIGHTING</td>
<td>specular</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ambient scene color</td>
<td>GetMaterial</td>
<td>LIGHTING</td>
<td>ambient</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Viewer is local</td>
<td>GetBool</td>
<td>LIGHTING</td>
<td>boolean</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Use two-sided lighting</td>
<td>GetBool</td>
<td>LIGHTING</td>
<td>boolean</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Color control</td>
<td>GetColor</td>
<td>LIGHTING</td>
<td>color</td>
</tr>
</tbody>
</table>

Table 6.14. Lighting (see also table 2.11 for defaults)

Version 3.0 (with changes marked - August 11, 2008)
<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMBIENT</td>
<td>8 × C</td>
<td>GetLightfv</td>
<td>(0.0,0.0,0.0,1.0)</td>
<td>Ambient intensity of light i</td>
<td>2.19.1</td>
<td>lighting</td>
</tr>
<tr>
<td>DIFFUSE</td>
<td>8 × C</td>
<td>GetLightfv</td>
<td>see table 2.11</td>
<td>Diffuse intensity of light i</td>
<td>2.19.1</td>
<td>lighting</td>
</tr>
<tr>
<td>SPECULAR</td>
<td>8 × C</td>
<td>GetLightfv</td>
<td>see table 2.11</td>
<td>Specular intensity of light i</td>
<td>2.19.1</td>
<td>lighting</td>
</tr>
<tr>
<td>POSITION</td>
<td>8 × P</td>
<td>GetLightfv</td>
<td>(0.0,0.0,1.0,0.0)</td>
<td>Position of light i</td>
<td>2.19.1</td>
<td>lighting</td>
</tr>
<tr>
<td>CONSTANT_ATTENUATION</td>
<td>8 × R+</td>
<td>GetLightfv</td>
<td>1.0</td>
<td>Constant atten. factor</td>
<td>2.19.1</td>
<td>lighting</td>
</tr>
<tr>
<td>LINEAR_ATTENUATION</td>
<td>8 × R+</td>
<td>GetLightfv</td>
<td>0.0</td>
<td>Linear atten. factor</td>
<td>2.19.1</td>
<td>lighting</td>
</tr>
<tr>
<td>QUADRATIC_ATTENUATION</td>
<td>8 × R+</td>
<td>GetLightfv</td>
<td>0.0</td>
<td>Quadratic atten. factor</td>
<td>2.19.1</td>
<td>lighting</td>
</tr>
<tr>
<td>SPOT_DIRECTION</td>
<td>8 × D</td>
<td>GetLightfv</td>
<td>(0.0,0.0,-1.0)</td>
<td>Spotlight direction of light i</td>
<td>2.19.1</td>
<td>lighting</td>
</tr>
<tr>
<td>SPOT_EXPONENT</td>
<td>8 × R+</td>
<td>GetLightfv</td>
<td>0.0</td>
<td>Spotlight exponent of light i</td>
<td>2.19.1</td>
<td>lighting</td>
</tr>
<tr>
<td>SPOT_CUTOFF</td>
<td>8 × R+</td>
<td>GetLightfv</td>
<td>180.0</td>
<td>Spot. angle of light i</td>
<td>2.19.1</td>
<td>lighting</td>
</tr>
<tr>
<td>LIGHT</td>
<td>8 × B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>True if light i enabled</td>
<td>2.19.1</td>
<td>lighting/enable</td>
</tr>
<tr>
<td>COLOR_INDEXES</td>
<td>2 × 3 × R</td>
<td>GetMaterialfv</td>
<td>0,1</td>
<td>a_m, d_m, and s_m for color index</td>
<td>2.19.1</td>
<td>lighting</td>
</tr>
</tbody>
</table>
### Table 6.16. Rasterization

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>POINT_SIZE</td>
<td>$R^+$</td>
<td>GetFloatv</td>
<td>1.0</td>
<td>Point size</td>
<td>3.4</td>
<td>point</td>
</tr>
<tr>
<td>POINT_SMOOTH</td>
<td>$B$</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Point antialiasing on</td>
<td>3.4</td>
<td>point/enable</td>
</tr>
<tr>
<td>POINT_SPRITE</td>
<td>$B$</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Point sprite enable</td>
<td>3.4</td>
<td>point/enable</td>
</tr>
<tr>
<td>POINT_SIZE_MIN</td>
<td>$R^+$</td>
<td>GetFloatv</td>
<td>0.0</td>
<td>Attenuated minimum point size</td>
<td>3.4</td>
<td>point</td>
</tr>
<tr>
<td>POINT_SIZE_MAX</td>
<td>$R^+$</td>
<td>GetFloatv</td>
<td>1</td>
<td>Attenuated maximum point size. Max. of the impl. dependent max. aliased and smooth point sizes.</td>
<td>3.4</td>
<td>point</td>
</tr>
<tr>
<td>POINT_FADE_THRESHOLD_SIZE</td>
<td>$R^+$</td>
<td>GetFloatv</td>
<td>1.0</td>
<td>Threshold for alpha attenuation</td>
<td>3.4</td>
<td>point</td>
</tr>
<tr>
<td>POINT_DISTANCE_ATTENUATION</td>
<td>$3 \times R^+$</td>
<td>GetFloatv</td>
<td>1,0,0</td>
<td>Attenuation coefficients</td>
<td>3.4</td>
<td>point</td>
</tr>
<tr>
<td>POINT_SPRITE_COORD_ORIGIN</td>
<td>$Z_2$</td>
<td>GetIntegerv</td>
<td>UPPER_LEFT</td>
<td>Origin orientation for point sprites</td>
<td>3.4</td>
<td>point</td>
</tr>
<tr>
<td>LINE_WIDTH</td>
<td>$R^+$</td>
<td>GetFloatv</td>
<td>1.0</td>
<td>Line width</td>
<td>3.5</td>
<td>line</td>
</tr>
<tr>
<td>LINE_SMOOTH</td>
<td>$B$</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Line antialiasing on</td>
<td>3.5</td>
<td>line/enable</td>
</tr>
<tr>
<td>LINE_STIPPLE_PATTERN</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>1's</td>
<td>Line stipple</td>
<td>3.5.2</td>
<td>line</td>
</tr>
<tr>
<td>LINE_STIPPLE_REPEAT</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>1</td>
<td>Line stipple repeat</td>
<td>3.5.2</td>
<td>line</td>
</tr>
<tr>
<td>LINE_STIPPLE</td>
<td>$B$</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Line stipple enable</td>
<td>3.5.2</td>
<td>line/enable</td>
</tr>
</tbody>
</table>
### Table 6.17. Rasterization (cont.)

<table>
<thead>
<tr>
<th>Command</th>
<th>Type</th>
<th>Description</th>
<th>Initial Value</th>
<th>Get Command</th>
<th>Get value</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>CULLFACE</td>
<td>Z₁</td>
<td>Polygon culling enabled</td>
<td>FALSE</td>
<td>GetIntegerv</td>
<td>Z₁</td>
<td>polygon/enable</td>
</tr>
<tr>
<td>POLYGONMODE</td>
<td>Z₂</td>
<td>Polygon frontface CW/CCW indicator</td>
<td>FALSE</td>
<td>GetIntegerv</td>
<td>Z₂</td>
<td>polygon/enable</td>
</tr>
<tr>
<td>POLYGONSMOOTH</td>
<td>B</td>
<td>Polygon anti-aliasing on</td>
<td>FALSE</td>
<td>GetFloatv</td>
<td>B</td>
<td>polygon/enable</td>
</tr>
<tr>
<td>POLYGONORIENTLINE</td>
<td>R</td>
<td>Polygon offset factor</td>
<td>0</td>
<td>GetFloatv</td>
<td>R</td>
<td>polygon/enable</td>
</tr>
<tr>
<td>POLYGONORIENTOFFSCREEN</td>
<td>B</td>
<td>Polygon offset units</td>
<td>FALSE</td>
<td>GetFloatv</td>
<td>B</td>
<td>polygon/enable</td>
</tr>
<tr>
<td>POLYGONORIENTOFFSCREENPOINT</td>
<td>B</td>
<td>Polygon offset enable for POINT mode rasterization</td>
<td>FALSE</td>
<td>GetFloatv</td>
<td>B</td>
<td>polygon/enable</td>
</tr>
<tr>
<td>POLYGONORIENTOFFSCREENLINE</td>
<td>B</td>
<td>Polygon offset enable for LINE mode rasterization</td>
<td>FALSE</td>
<td>GetFloatv</td>
<td>B</td>
<td>polygon/enable</td>
</tr>
<tr>
<td>POLYGONORIENTOFFSCREENFILL</td>
<td>B</td>
<td>Polygon offset enable for FILL mode rasterization</td>
<td>FALSE</td>
<td>GetFloatv</td>
<td>B</td>
<td>polygon/enable</td>
</tr>
<tr>
<td>POLYGONSTIPPLE</td>
<td>I</td>
<td>Polygon stipple</td>
<td>1's</td>
<td>GetPolygonStipple</td>
<td>I</td>
<td>polygon/enable</td>
</tr>
<tr>
<td>POLYGONSTIPPLE</td>
<td>B</td>
<td>Polygon stipple enable</td>
<td>FALSE</td>
<td>GetPolygonStipple</td>
<td>B</td>
<td>polygon/enable</td>
</tr>
</tbody>
</table>

Version 3.0 (with changes marked - August 11, 2008)
### Table 6.18. Multisampling

<table>
<thead>
<tr>
<th>Get value</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Attribute</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTISAMPLE</td>
<td>IsEnabled</td>
<td>TRUE</td>
<td>Multisample rasterization</td>
<td>ENABLED</td>
<td>6.2.3</td>
</tr>
<tr>
<td>SAMPLE_ALPHA_TO_COVERAGE</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Modify coverage from alpha to one</td>
<td>ENABLED</td>
<td>4.1.3</td>
</tr>
<tr>
<td>SAMPLE_COVERAGE</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Set coverage mask value</td>
<td>ENABLED</td>
<td>4.1.3</td>
</tr>
<tr>
<td>SAMPLE_COVERAGE_VALUE</td>
<td>GetFloatv</td>
<td>1</td>
<td>Coverage mask value</td>
<td>ENABLED</td>
<td>4.1.3</td>
</tr>
<tr>
<td>SAMPLE_COVERAGE_INVERT</td>
<td>GetBoolean</td>
<td>FALSE</td>
<td>Invert coverage mask value</td>
<td>ENABLED</td>
<td>4.1.3</td>
</tr>
</tbody>
</table>

Version 3.0 (with changes marked - August 11, 2008)
## 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>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXTURE_x:D</td>
<td>2 × 3 × B</td>
<td>IsActive</td>
<td>FALSE</td>
<td>True if xD texturing is enabled; x is 1, 2, or 3</td>
<td>3.9.17</td>
<td>texture/enable</td>
</tr>
<tr>
<td>TEXTURE_CUBE_MAP</td>
<td>2 × B</td>
<td>IsActive</td>
<td>FALSE</td>
<td>True if cube map texturing is enabled</td>
<td>3.9.13</td>
<td>texture/enable</td>
</tr>
<tr>
<td>TEXTURE_BINDING_x:D</td>
<td>2 × 3 × Z⁺</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Texture object bound to TEXTURE_x:D</td>
<td>3.9.12</td>
<td>texture</td>
</tr>
<tr>
<td>TEXTURE_BINDING_1D_ARRAY</td>
<td>2 × 3 × Z⁺</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Texture object bound to TEXTURE_1D_ARRAY</td>
<td>3.9.12</td>
<td>texture</td>
</tr>
<tr>
<td>TEXTURE_BINDING_2D_ARRAY</td>
<td>2 × 3 × Z⁺</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Texture object bound to TEXTURE_2D_ARRAY</td>
<td>3.9.12</td>
<td>texture</td>
</tr>
<tr>
<td>TEXTURE_BINDING_CUBE_MAP</td>
<td>2 × Z⁺</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Texture object bound to TEXTURE_CUBE_MAP</td>
<td>3.9.11</td>
<td>texture</td>
</tr>
<tr>
<td>TEXTURE_cube_map</td>
<td>n × I</td>
<td>GetTexImage</td>
<td>see 3.9</td>
<td>xD texture image at l.o.d.</td>
<td>3.9</td>
<td>–</td>
</tr>
<tr>
<td>TEXTURE_CUBE_MAP_POSITIVE_X</td>
<td>n × I</td>
<td>GetTexImage</td>
<td>see 3.9.1</td>
<td>+x face cube map texture image at l.o.d. i</td>
<td>3.9.1</td>
<td>–</td>
</tr>
<tr>
<td>TEXTURE_CUBE_MAP_NEGATIVE_X</td>
<td>n × I</td>
<td>GetTexImage</td>
<td>see 3.9.1</td>
<td>−x face cube map texture image at l.o.d. i</td>
<td>3.9.1</td>
<td>–</td>
</tr>
<tr>
<td>TEXTURE_CUBE_MAP_POSITIVE_Y</td>
<td>n × I</td>
<td>GetTexImage</td>
<td>see 3.9.1</td>
<td>+y face cube map texture image at l.o.d. i</td>
<td>3.9.1</td>
<td>–</td>
</tr>
<tr>
<td>TEXTURE_CUBE_MAP_NEGATIVE_Y</td>
<td>n × I</td>
<td>GetTexImage</td>
<td>see 3.9.1</td>
<td>−y face cube map texture image at l.o.d. i</td>
<td>3.9.1</td>
<td>–</td>
</tr>
<tr>
<td>TEXTURE_CUBE_MAP_POSITIVE_Z</td>
<td>n × I</td>
<td>GetTexImage</td>
<td>see 3.9.1</td>
<td>+z face cube map texture image at l.o.d. i</td>
<td>3.9.1</td>
<td>–</td>
</tr>
<tr>
<td>TEXTURE_CUBE_MAP_NEGATIVE_Z</td>
<td>n × I</td>
<td>GetTexImage</td>
<td>see 3.9.1</td>
<td>−z face cube map texture image at l.o.d. i</td>
<td>3.9.1</td>
<td>–</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Initial Value</td>
<td>Description</td>
<td>Sec.</td>
<td>Attribute</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------</td>
<td>------------------</td>
<td>---------------</td>
<td>------------------------------------------</td>
<td>------</td>
<td>-----------</td>
</tr>
<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>3.9</td>
<td>texture</td>
</tr>
<tr>
<td>TEXTURE_MIN_FILTER</td>
<td>$n \times Z_0$</td>
<td>GetTexParameter</td>
<td>see 3.9</td>
<td>Minification function</td>
<td>3.9</td>
<td>texture</td>
</tr>
<tr>
<td>TEXTURE_MAG_FILTER</td>
<td>$n \times Z_2$</td>
<td>GetTexParameter</td>
<td>see 3.9</td>
<td>Magnification function</td>
<td>3.9</td>
<td>texture</td>
</tr>
<tr>
<td>TEXTURE_WRAP_S</td>
<td>$n \times Z_5$</td>
<td>GetTexParameter</td>
<td>REPEAT</td>
<td>Texcoord s wrap mode</td>
<td>3.9</td>
<td>texture</td>
</tr>
<tr>
<td>TEXTURE_WRAP_T</td>
<td>$n \times Z_5$</td>
<td>GetTexParameter</td>
<td>REPEAT</td>
<td>Texcoord t wrap mode (2D, 3D, cube map textures only)</td>
<td>3.9</td>
<td>texture</td>
</tr>
<tr>
<td>TEXTURE_WRAP_R</td>
<td>$n \times Z_5$</td>
<td>GetTexParameter</td>
<td>REPEAT</td>
<td>Texcoord r wrap mode (3D textures only)</td>
<td>3.9</td>
<td>texture</td>
</tr>
<tr>
<td>TEXTURE_PRIORITY</td>
<td>$n \times R^{[0,1]}$</td>
<td>GetTexParameterfv</td>
<td>1</td>
<td>Texture object priority</td>
<td>3.9</td>
<td>texture</td>
</tr>
<tr>
<td>TEXTURE_RESIDENT</td>
<td>$n \times B$</td>
<td>GetTexParameteriv</td>
<td>see 3.9.12</td>
<td>Texture residency</td>
<td>3.9</td>
<td>texture</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>3.9</td>
<td>texture</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>3.9</td>
<td>texture</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>3.9</td>
<td>texture</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>3.9</td>
<td>texture</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\textsubscript{texobj})</td>
<td>3.9</td>
<td>texture</td>
</tr>
<tr>
<td>DEPTH_TEXTURE_MODE</td>
<td>$n \times Z_3$</td>
<td>GetTexParameteriv</td>
<td>LUMINANCE</td>
<td>Depth texture mode</td>
<td>3.9</td>
<td>texture</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>3.9</td>
<td>texture</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>3.9</td>
<td>texture</td>
</tr>
<tr>
<td>GENERATE_MIPMAP</td>
<td>$n \times B$</td>
<td>GetTexParameter</td>
<td>FALSE</td>
<td>Automatic mipmap generation enabled</td>
<td>3.9</td>
<td>texture</td>
</tr>
</tbody>
</table>

Table 6.20. 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>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXTURE, WIDTH</td>
<td>( n \times Z^+ )</td>
<td>GetTexLevelParameter</td>
<td>0</td>
<td>Specified width</td>
<td>3.9</td>
<td>–</td>
</tr>
<tr>
<td>TEXTURE, HEIGHT</td>
<td>( n \times Z^+ )</td>
<td>GetTexLevelParameter</td>
<td>0</td>
<td>Specified height (2D/3D)</td>
<td>3.9</td>
<td>–</td>
</tr>
<tr>
<td>TEXTURE, DEPTH</td>
<td>( n \times Z^+ )</td>
<td>GetTexLevelParameter</td>
<td>0</td>
<td>Specified depth (3D)</td>
<td>3.9</td>
<td>–</td>
</tr>
<tr>
<td>TEXTURE, BORDER</td>
<td>( n \times Z^+ )</td>
<td>GetTexLevelParameter</td>
<td>0</td>
<td>Specified border width</td>
<td>3.9</td>
<td>–</td>
</tr>
<tr>
<td>TEXTURE, INTERNAL_FORMAT (TEXTURE, COMPONENTS)</td>
<td>( n \times Z_{\text{grs}} )</td>
<td>GetTexLevelParameter</td>
<td>1</td>
<td>Internal format</td>
<td>3.9</td>
<td>–</td>
</tr>
<tr>
<td>TEXTURE, ( x ), SIZE</td>
<td>( n \times 8 \times Z^+ )</td>
<td>GetTexLevelParameter</td>
<td>0</td>
<td>Component resolution (( x ) is RED, GREEN, BLUE, ALPHA, LUMINANCE, INTENSITY, DEPTH, or STENCIL)</td>
<td>3.9</td>
<td>–</td>
</tr>
<tr>
<td>TEXTURE, ( x ), TYPE</td>
<td>( n \times Z_5 )</td>
<td>GetTexLevelParameter</td>
<td>NONE</td>
<td>Component type (( x ) is RED, GREEN, BLUE, ALPHA, LUMINANCE, INTENSITY, or DEPTH)</td>
<td>6.1.3</td>
<td>–</td>
</tr>
<tr>
<td>TEXTURE, COMPRESSED</td>
<td>( n \times B )</td>
<td>GetTexLevelParameter</td>
<td>FALSE</td>
<td>True if image has a compressed internal format</td>
<td>3.9.3</td>
<td>-</td>
</tr>
<tr>
<td>TEXTURE, COMPRESSED IMAGE, SIZE</td>
<td>( n \times Z^+ )</td>
<td>GetTexLevelParameter</td>
<td>0</td>
<td>Size (in bytes) of compressed image</td>
<td>3.9.3</td>
<td>-</td>
</tr>
</tbody>
</table>
### Table 6.22. Texture Environment and Generation

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>COORD_REPLACE</td>
<td>$2 \times B$</td>
<td>GetTexEnviv</td>
<td>FALSE</td>
<td>Coordinate replacement enable</td>
<td>3.4</td>
<td>point</td>
</tr>
<tr>
<td>ACTIVE_TEXTURE</td>
<td>$Z_2^*$</td>
<td>GetIntegerv</td>
<td>TEXTURE0</td>
<td>Active texture unit selector</td>
<td>2.7</td>
<td>texture</td>
</tr>
<tr>
<td>TEXTURE_ENV_MODE</td>
<td>$2 \times Z_6$</td>
<td>GetTexEnviv</td>
<td>MODULATE</td>
<td>Texture application function</td>
<td>3.9.13</td>
<td>texture</td>
</tr>
<tr>
<td>TEXTURE_ENV_COLOR</td>
<td>$2 \times C$</td>
<td>GetTexEnvfv</td>
<td>0.0,0.0</td>
<td>Texture environment color</td>
<td>3.9.13</td>
<td>texture</td>
</tr>
<tr>
<td>TEXTURE_LOD_BIAS</td>
<td>$2 \times R$</td>
<td>GetTexEnvfv</td>
<td>0.0</td>
<td>Texture level of detail bias</td>
<td>3.9.7</td>
<td>texture</td>
</tr>
<tr>
<td>TEXTURE_GEN x</td>
<td>$2 \times 4 \times B$</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Texgen enabled ($x$ is S, T, R, or Q)</td>
<td>2.12.4</td>
<td>texture/enable</td>
</tr>
<tr>
<td>EYE_PLANE</td>
<td>$2 \times 4 \times R^4$</td>
<td>GetTexGenfv</td>
<td>see 2.12.4</td>
<td>Texgen plane equation coefficients (for S, T, R, and Q)</td>
<td>2.12.4</td>
<td>texture</td>
</tr>
<tr>
<td>OBJECT_PLANE</td>
<td>$2 \times 4 \times R^4$</td>
<td>GetTexGenfv</td>
<td>see 2.12.4</td>
<td>Texgen object linear coefficients (for S, T, R, and Q)</td>
<td>2.12.4</td>
<td>texture</td>
</tr>
<tr>
<td>TEXTURE_GEN_MODE</td>
<td>$2 \times 4 \times Z_5$</td>
<td>GetTexGeniv</td>
<td>EYE_LINEAR</td>
<td>Function used for texgen (for S, T, R, and Q)</td>
<td>2.12.4</td>
<td>texture</td>
</tr>
<tr>
<td>COMBINE_RGB</td>
<td>$2 \times Z_8$</td>
<td>GetTexEnviv</td>
<td>MODULATE</td>
<td>RGB combiner function</td>
<td>3.9.13</td>
<td>texture</td>
</tr>
<tr>
<td>COMBINE_ALPHA</td>
<td>$2 \times Z_6$</td>
<td>GetTexEnviv</td>
<td>MODULATE</td>
<td>Alpha combiner function</td>
<td>3.9.13</td>
<td>texture</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Initial Value</td>
<td>Description</td>
<td>Sec.</td>
<td>Attribute</td>
</tr>
<tr>
<td>-------------</td>
<td>------------</td>
<td>-----------------</td>
<td>---------------</td>
<td>-------------</td>
<td>------</td>
<td>-----------</td>
</tr>
<tr>
<td>SRC0, RGB</td>
<td>2 * Z3</td>
<td>GetTexEnvv</td>
<td>TEXTURE</td>
<td>RGB source 0</td>
<td>3.9.13</td>
<td>texture</td>
</tr>
<tr>
<td>SRC1, RGB</td>
<td>2 * Z3</td>
<td>GetTexEnvv</td>
<td>PREVIOUS</td>
<td>RGB source 1</td>
<td>3.9.13</td>
<td>texture</td>
</tr>
<tr>
<td>SRC2, RGB</td>
<td>2 * Z3</td>
<td>GetTexEnvv</td>
<td>CONSTANT</td>
<td>RGB source 2</td>
<td>3.9.13</td>
<td>texture</td>
</tr>
<tr>
<td>SRC0, ALPHA</td>
<td>2 * Z3</td>
<td>GetTexEnvv</td>
<td>TEXTURE</td>
<td>Alpha source 0</td>
<td>3.9.13</td>
<td>texture</td>
</tr>
<tr>
<td>SRC1, ALPHA</td>
<td>2 * Z3</td>
<td>GetTexEnvv</td>
<td>PREVIOUS</td>
<td>Alpha source 1</td>
<td>3.9.13</td>
<td>texture</td>
</tr>
<tr>
<td>SRC2, ALPHA</td>
<td>2 * Z3</td>
<td>GetTexEnvv</td>
<td>CONSTANT</td>
<td>Alpha source 2</td>
<td>3.9.13</td>
<td>texture</td>
</tr>
<tr>
<td>OPERAND0, RGB</td>
<td>2 * Z4</td>
<td>GetTexEnvv</td>
<td>SRC_COLOR</td>
<td>RGB operand 0</td>
<td>3.9.13</td>
<td>texture</td>
</tr>
<tr>
<td>OPERAND1, RGB</td>
<td>2 * Z4</td>
<td>GetTexEnvv</td>
<td>SRC_COLOR</td>
<td>RGB operand 1</td>
<td>3.9.13</td>
<td>texture</td>
</tr>
<tr>
<td>OPERAND2, RGB</td>
<td>2 * Z4</td>
<td>GetTexEnvv</td>
<td>SRC_ALPHA</td>
<td>RGB operand 2</td>
<td>3.9.13</td>
<td>texture</td>
</tr>
<tr>
<td>OPERAND0, ALPHA</td>
<td>2 * Z2</td>
<td>GetTexEnvv</td>
<td>SRC_ALPHA</td>
<td>Alpha operand 0</td>
<td>3.9.13</td>
<td>texture</td>
</tr>
<tr>
<td>OPERAND1, ALPHA</td>
<td>2 * Z2</td>
<td>GetTexEnvv</td>
<td>SRC_ALPHA</td>
<td>Alpha operand 1</td>
<td>3.9.13</td>
<td>texture</td>
</tr>
<tr>
<td>OPERAND2, ALPHA</td>
<td>2 * Z2</td>
<td>GetTexEnvv</td>
<td>SRC_ALPHA</td>
<td>Alpha operand 2</td>
<td>3.9.13</td>
<td>texture</td>
</tr>
<tr>
<td>RGB_SCALE</td>
<td>2 * R^4</td>
<td>GetTexEnvfv</td>
<td>1.0</td>
<td>RGB post-combiner scaling</td>
<td>3.9.13</td>
<td>texture</td>
</tr>
<tr>
<td>ALPHA_SCALE</td>
<td>2 * R^4</td>
<td>GetTexEnvfv</td>
<td>1.0</td>
<td>Alpha post-combiner scaling</td>
<td>3.9.13</td>
<td>texture</td>
</tr>
</tbody>
</table>
### Table 6.2.4. Pixel Operations

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Command</th>
<th>Description</th>
<th>Initial Value</th>
<th>Get Command</th>
<th>Get Value</th>
<th>Type</th>
<th>B</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>scissor/enable</td>
<td>scissor</td>
<td>Scissor enabled</td>
<td>FALSE</td>
<td>FALSE</td>
<td>B</td>
<td>4 × 2</td>
<td>B</td>
<td>Z</td>
</tr>
<tr>
<td>alpha-test/enable</td>
<td>alpha</td>
<td>Alpha test enabled</td>
<td>ALWAYS</td>
<td>ALWAYS</td>
<td>B</td>
<td>4 × 2</td>
<td>B</td>
<td>Z</td>
</tr>
<tr>
<td>stencil-buffer/enable</td>
<td>stencil</td>
<td>Stencil enabled</td>
<td>FALSE</td>
<td>FALSE</td>
<td>B</td>
<td>4 × 2</td>
<td>B</td>
<td>Z</td>
</tr>
<tr>
<td>stencil-buffer</td>
<td>stencil</td>
<td>Stencil enabled</td>
<td>FALSE</td>
<td>FALSE</td>
<td>B</td>
<td>4 × 2</td>
<td>B</td>
<td>Z</td>
</tr>
<tr>
<td>stencil-buffer</td>
<td>stencil</td>
<td>Stencil enabled</td>
<td>FALSE</td>
<td>FALSE</td>
<td>B</td>
<td>4 × 2</td>
<td>B</td>
<td>Z</td>
</tr>
<tr>
<td>stencil-buffer</td>
<td>stencil</td>
<td>Stencil enabled</td>
<td>FALSE</td>
<td>FALSE</td>
<td>B</td>
<td>4 × 2</td>
<td>B</td>
<td>Z</td>
</tr>
<tr>
<td>stencil-buffer</td>
<td>stencil</td>
<td>Stencil enabled</td>
<td>FALSE</td>
<td>FALSE</td>
<td>B</td>
<td>4 × 2</td>
<td>B</td>
<td>Z</td>
</tr>
<tr>
<td>stencil-buffer</td>
<td>stencil</td>
<td>Stencil enabled</td>
<td>FALSE</td>
<td>FALSE</td>
<td>B</td>
<td>4 × 2</td>
<td>B</td>
<td>Z</td>
</tr>
<tr>
<td>stencil-buffer</td>
<td>stencil</td>
<td>Stencil enabled</td>
<td>FALSE</td>
<td>FALSE</td>
<td>B</td>
<td>4 × 2</td>
<td>B</td>
<td>Z</td>
</tr>
<tr>
<td>stencil-buffer</td>
<td>stencil</td>
<td>Stencil enabled</td>
<td>FALSE</td>
<td>FALSE</td>
<td>B</td>
<td>4 × 2</td>
<td>B</td>
<td>Z</td>
</tr>
<tr>
<td>stencil-buffer</td>
<td>stencil</td>
<td>Stencil enabled</td>
<td>FALSE</td>
<td>FALSE</td>
<td>B</td>
<td>4 × 2</td>
<td>B</td>
<td>Z</td>
</tr>
<tr>
<td>stencil-buffer</td>
<td>stencil</td>
<td>Stencil enabled</td>
<td>FALSE</td>
<td>FALSE</td>
<td>B</td>
<td>4 × 2</td>
<td>B</td>
<td>Z</td>
</tr>
<tr>
<td>stencil-buffer</td>
<td>stencil</td>
<td>Stencil enabled</td>
<td>FALSE</td>
<td>FALSE</td>
<td>B</td>
<td>4 × 2</td>
<td>B</td>
<td>Z</td>
</tr>
<tr>
<td>stencil-buffer</td>
<td>stencil</td>
<td>Stencil enabled</td>
<td>FALSE</td>
<td>FALSE</td>
<td>B</td>
<td>4 × 2</td>
<td>B</td>
<td>Z</td>
</tr>
</tbody>
</table>

**Table 6.24. Pixel Operations**

Version 3.0 (with changes marked - August 11, 2008)
<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPTHTEST</td>
<td>$B$</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Depth buffer enabled</td>
<td>4.1.6</td>
<td>depth-buffer/enable</td>
</tr>
<tr>
<td>DEPTH, FUNC</td>
<td>$Z_8$</td>
<td>GetIntegerv</td>
<td>LESS</td>
<td>Depth buffer test function</td>
<td>4.1.6</td>
<td>depth-buffer</td>
</tr>
<tr>
<td>BLENDFUNC</td>
<td>$Z_8$</td>
<td>GetIntegerv</td>
<td>LESS</td>
<td>Depth buffer test function</td>
<td>4.1.6</td>
<td>depth-buffer</td>
</tr>
<tr>
<td>BLEND</td>
<td>$1 * \times B$</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Blending enabled for draw buffer $i$</td>
<td>4.1.8</td>
<td>color-buffer/enable</td>
</tr>
<tr>
<td>BLEND_SRC, RGB (v1.3 BLEND_SRC)</td>
<td>$Z_{15}$</td>
<td>GetIntegerv</td>
<td>ONE</td>
<td>Blending source RGB function</td>
<td>4.1.8</td>
<td>color-buffer</td>
</tr>
<tr>
<td>BLEND_SRC, ALPHA</td>
<td>$Z_{15}$</td>
<td>GetIntegerv</td>
<td>ONE</td>
<td>Blending source A function</td>
<td>4.1.8</td>
<td>color-buffer</td>
</tr>
<tr>
<td>BLEND_DST, RGB (v1.3 BLEND_DST)</td>
<td>$Z_{14}$</td>
<td>GetIntegerv</td>
<td>ZERO</td>
<td>Blending dest. RGB function</td>
<td>4.1.8</td>
<td>color-buffer</td>
</tr>
<tr>
<td>BLEND_DST, ALPHA</td>
<td>$Z_{14}$</td>
<td>GetIntegerv</td>
<td>ZERO</td>
<td>Blending dest. A function</td>
<td>4.1.8</td>
<td>color-buffer</td>
</tr>
<tr>
<td>BLEND_EQUATION, RGB (v1.5 BLEND_EQUATION)</td>
<td>$Z_{5}$</td>
<td>GetIntegerv</td>
<td>FUNC, ADD</td>
<td>RGB blending equation</td>
<td>4.1.8</td>
<td>color-buffer</td>
</tr>
<tr>
<td>BLEND_EQUATION, ALPHA</td>
<td>$Z_{5}$</td>
<td>GetIntegerv</td>
<td>FUNC, ADD</td>
<td>Alpha blending equation</td>
<td>4.1.8</td>
<td>color-buffer</td>
</tr>
<tr>
<td>BLEND_COLOR</td>
<td>$C$</td>
<td>GetFloatv</td>
<td>0,0,0,0</td>
<td>Constant blend color</td>
<td>4.1.8</td>
<td>color-buffer</td>
</tr>
<tr>
<td>FRAMEBUFFER_SRGB</td>
<td>$B$</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>sRGB update and blending enable</td>
<td>4.1.8</td>
<td>color-buffer/enable</td>
</tr>
<tr>
<td>DITHER</td>
<td>$B$</td>
<td>IsEnabled</td>
<td>TRUE</td>
<td>Dithering enabled</td>
<td>4.1.10</td>
<td>color-buffer/enable</td>
</tr>
<tr>
<td>INDEX, LOGIC, OP (v1.0 LOGIC, OP)</td>
<td>$B$</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Index logic op enabled</td>
<td>4.1.11</td>
<td>color-buffer/enable</td>
</tr>
<tr>
<td>COLOR, LOGIC, OP</td>
<td>$B$</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Color logic op enabled</td>
<td>4.1.11</td>
<td>color-buffer/enable</td>
</tr>
<tr>
<td>LOGIC, OP, MODE</td>
<td>$Z_{16}$</td>
<td>GetIntegerv</td>
<td>COPY</td>
<td>Logic op function</td>
<td>4.1.11</td>
<td>color-buffer</td>
</tr>
</tbody>
</table>

Table 6.25. Pixel Operations (cont.)
<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX_WRITEMASK</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>1's</td>
<td>Color index writemask</td>
<td>4.2.2</td>
<td>color-buffer</td>
</tr>
<tr>
<td>COLOR_WRITEMASK</td>
<td>$B \ast \times 4 \times B$</td>
<td>GetBooleani_v</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>
<td>color-buffer</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>
<td>depth-buffer</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>
<td>stencil-buffer</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>
<td>stencil-buffer</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 (RGBA mode)</td>
<td>4.2.3</td>
<td>color-buffer</td>
</tr>
<tr>
<td>INDEX_CLEAR_VALUE</td>
<td>$CI$</td>
<td>GetFloatv</td>
<td>0</td>
<td>Color buffer clear value (color index mode)</td>
<td>4.2.3</td>
<td>color-buffer</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>
<td>depth-buffer</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>
<td>stencil-buffer</td>
</tr>
<tr>
<td>ACCUM_CLEAR_VALUE</td>
<td>$4 \times R^+$</td>
<td>GetFloatv</td>
<td>0</td>
<td>Accumulation buffer clear value</td>
<td>4.2.3</td>
<td>accum-buffer</td>
</tr>
</tbody>
</table>

Table 6.26. Framebuffer Control
### Table 6.27. Framebuffer (state per target binding point)

<table>
<thead>
<tr>
<th>Get command</th>
<th>Type</th>
<th>Description</th>
<th>Initial value</th>
<th>Get value</th>
<th>Attribute</th>
<th>Sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetIntegerv</td>
<td>DRAW_FRAMEBUFFER_BINDING</td>
<td>framebuffer object bound to DRAW_FRAMEBUFFER</td>
<td>0</td>
<td>DRAW_FRAMEBUFFER_BINDING</td>
<td>4.4.1</td>
<td></td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>READ_FRAMEBUFFER_BINDING</td>
<td>framebuffer object bound to READ_FRAMEBUFFER</td>
<td>0</td>
<td>READ_FRAMEBUFFER_BINDING</td>
<td>4.4.1</td>
<td></td>
</tr>
</tbody>
</table>
Table 6.28. Framebuffer (state per framebuffer object)
<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE</td>
<td>Z</td>
<td>GetFramebuffer-Attachment-Parameteriv</td>
<td>NONE</td>
<td>Type of image attached to framebuffer attachment point</td>
<td>4.4.2</td>
<td>–</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_OBJECT_NAME</td>
<td>Z</td>
<td>GetFramebuffer-Attachment-Parameteriv</td>
<td>0</td>
<td>Name of object attached to framebuffer attachment point</td>
<td>4.4.2</td>
<td>–</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL</td>
<td>Z</td>
<td>GetFramebuffer-Attachment-Parameteriv</td>
<td>0</td>
<td>Mipmap level of texture image attached, if object attached is texture</td>
<td>4.4.2</td>
<td>–</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE</td>
<td>Z^+</td>
<td>GetFramebuffer-Attachment-Parameteriv</td>
<td>TEXTURE_CUBE_MAP_POSITIVE_X</td>
<td>Cubemap face of texture image attached, if object attached is cubemap texture</td>
<td>4.4.2</td>
<td>–</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER</td>
<td>Z</td>
<td>GetFramebuffer-Attachment-Parameteriv</td>
<td>0</td>
<td>Layer of texture image attached, if object attached is 3D texture</td>
<td>4.4.2</td>
<td>–</td>
</tr>
<tr>
<td>BUFFER_ATTACHMENT_COLOR_ENCODING</td>
<td>Z_2</td>
<td>GetFramebuffer-Attachment-Parameteriv</td>
<td>-</td>
<td>Encoding of components in the attached image</td>
<td>6.1.3</td>
<td>–</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE</td>
<td>Z_4</td>
<td>GetFramebuffer-Attachment-Parameteriv</td>
<td>-</td>
<td>Data type of components in the attached image</td>
<td>6.1.3</td>
<td>–</td>
</tr>
<tr>
<td>FRAMEBUFFER_ATTACHMENT_?,X_SIZE</td>
<td>Z^+</td>
<td>GetFramebuffer-Attachment-Parameteriv</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>
<td>–</td>
</tr>
</tbody>
</table>
### Table 6.30. Renderbuffer (state per target and binding point)

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Initial Value</th>
<th>Get Command</th>
<th>Description</th>
<th>Sec. Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>RENDERBUFFER_BINDING</td>
<td>0</td>
<td>GetIntegerv</td>
<td>renderbuffer object bound to RENDERBUFFER</td>
<td>44.2</td>
</tr>
</tbody>
</table>

This table lists the renderbuffer state for different targets and binding points. The `GetIntegerv` command is used to get the initial value of the renderbuffer object bound to the `RENDERBUFFER` target.
### 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>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>RENDERBUFFER_WIDTH</td>
<td>Z+</td>
<td>GetRenderbufferParameteriv</td>
<td>0</td>
<td>Width of renderbuffer</td>
<td>4.4.2</td>
<td></td>
</tr>
<tr>
<td>RENDERBUFFER_HEIGHT</td>
<td>Z+</td>
<td>GetRenderbufferParameteriv</td>
<td>0</td>
<td>Height of renderbuffer</td>
<td>4.4.2</td>
<td></td>
</tr>
<tr>
<td>RENDERBUFFER_INTERNAL_FORMAT</td>
<td>Z+</td>
<td>GetRenderbufferParameteriv</td>
<td>RGBA</td>
<td>Internal format of renderbuffer</td>
<td>4.4.2</td>
<td></td>
</tr>
<tr>
<td>RENDERBUFFER_RED_SIZE</td>
<td>Z+</td>
<td>GetRenderbufferParameteriv</td>
<td>0</td>
<td>Size in bits of renderbuffer image’s red component</td>
<td>4.4.2</td>
<td></td>
</tr>
<tr>
<td>RENDERBUFFER_GREEN_SIZE</td>
<td>Z+</td>
<td>GetRenderbufferParameteriv</td>
<td>0</td>
<td>Size in bits of renderbuffer image’s green component</td>
<td>4.4.2</td>
<td></td>
</tr>
<tr>
<td>RENDERBUFFER_BLUE_SIZE</td>
<td>Z+</td>
<td>GetRenderbufferParameteriv</td>
<td>0</td>
<td>Size in bits of renderbuffer image’s blue component</td>
<td>4.4.2</td>
<td></td>
</tr>
<tr>
<td>RENDERBUFFER_ALPHA_SIZE</td>
<td>Z+</td>
<td>GetRenderbufferParameteriv</td>
<td>0</td>
<td>Size in bits of renderbuffer image’s alpha component</td>
<td>4.4.2</td>
<td></td>
</tr>
<tr>
<td>RENDERBUFFER_DEPTH_SIZE</td>
<td>Z+</td>
<td>GetRenderbufferParameteriv</td>
<td>0</td>
<td>Size in bits of renderbuffer image’s depth component</td>
<td>4.4.2</td>
<td></td>
</tr>
<tr>
<td>RENDERBUFFER_STENCIL_SIZE</td>
<td>Z+</td>
<td>GetRenderbufferParameteriv</td>
<td>0</td>
<td>Size in bits of renderbuffer image’s stencil component</td>
<td>4.4.2</td>
<td></td>
</tr>
<tr>
<td>RENDERBUFFER_SAMPLES</td>
<td>Z+</td>
<td>GetRenderbufferParameteriv</td>
<td>0</td>
<td>Number of samples</td>
<td>4.4.2</td>
<td></td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Initial Value</td>
<td>Description</td>
<td>Sec.</td>
<td>Attribute</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------</td>
<td>---------------</td>
<td>---------------</td>
<td>------------------------------------</td>
<td>-------</td>
<td>-------------</td>
</tr>
<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>
<td>pixel-store</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>
<td>pixel-store</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>
<td>pixel-store</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>
<td>pixel-store</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>
<td>pixel-store</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>
<td>pixel-store</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>
<td>pixel-store</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>
<td>pixel-store</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>4.3.2</td>
<td>pixel-store</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>4.3.2</td>
<td>pixel-store</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>4.3.2</td>
<td>pixel-store</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>4.3.2</td>
<td>pixel-store</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>4.3.2</td>
<td>pixel-store</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>4.3.2</td>
<td>pixel-store</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>4.3.2</td>
<td>pixel-store</td>
</tr>
<tr>
<td>PACK_ALIGNMENT</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>4</td>
<td>Value of PACK_ALIGNMENT</td>
<td>4.3.2</td>
<td>pixel-store</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>4.3.2</td>
<td>pixel-store</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>6.1.13</td>
<td>pixel-store</td>
</tr>
</tbody>
</table>
### 6.2. STATE TABLES

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Initial Value</th>
<th>Get Command</th>
<th>Description</th>
<th>Sec.</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map-Color</td>
<td>FALSE</td>
<td>FALSE</td>
<td>GetBooleanv</td>
<td>True if colors are mapped</td>
<td>3.7.3</td>
<td>pixel</td>
</tr>
<tr>
<td>Map-Stencil</td>
<td>FALSE</td>
<td>FALSE</td>
<td>GetBooleanv</td>
<td>True if stencil values are mapped</td>
<td>3.7.3</td>
<td>pixel</td>
</tr>
<tr>
<td>Index-Shift</td>
<td>0</td>
<td>GetIntegerv</td>
<td></td>
<td>Value of INDEX-SHIFT</td>
<td>3.7.3</td>
<td>pixel</td>
</tr>
<tr>
<td>Index-Offset</td>
<td>0</td>
<td>GetIntegerv</td>
<td></td>
<td>Value of INDEX-OFFSET</td>
<td>3.7.3</td>
<td>pixel</td>
</tr>
<tr>
<td>x-SCALE</td>
<td>R</td>
<td>GetFloatv</td>
<td></td>
<td>Value of x-SCALE; x is RED, GREEN, BLUE, ALPHA, or DEPTH</td>
<td>3.7.3</td>
<td>pixel</td>
</tr>
<tr>
<td>x-BIAS</td>
<td>0</td>
<td>GetFloatv</td>
<td></td>
<td>Value of x-BIAS</td>
<td>3.7.3</td>
<td>pixel</td>
</tr>
</tbody>
</table>
### Table 6.34. Pixels (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>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLOR_TABLE</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>True if color table lookup is done</td>
<td>3.7.3</td>
<td>pixel/enable</td>
</tr>
<tr>
<td>POST_CONVOLUTION_COLOR_TABLE</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>True if post convolution color table lookup is done</td>
<td>3.7.3</td>
<td>pixel/enable</td>
</tr>
<tr>
<td>POST_COLOR_MATRIX_COLOR_TABLE</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>True if post color matrix color table lookup is done</td>
<td>3.7.3</td>
<td>pixel/enable</td>
</tr>
<tr>
<td>COLOR_TABLE</td>
<td>I</td>
<td>GetColorTable</td>
<td>empty</td>
<td>Color table</td>
<td>3.7.3</td>
<td>–</td>
</tr>
<tr>
<td>POST_CONVOLUTION_COLOR_TABLE</td>
<td>I</td>
<td>GetColorTable</td>
<td>empty</td>
<td>Post convolution color table</td>
<td>3.7.3</td>
<td>–</td>
</tr>
<tr>
<td>POST_COLOR_MATRIX_COLOR_TABLE</td>
<td>I</td>
<td>GetColorTable</td>
<td>empty</td>
<td>Post color matrix color table</td>
<td>3.7.3</td>
<td>–</td>
</tr>
<tr>
<td>COLOR_TABLE_FORMAT</td>
<td>2 × 3 × Z&lt;sub&gt;42&lt;/sub&gt;</td>
<td>GetColorTable-Parameteriv</td>
<td>RGBA</td>
<td>Color tables’ internal image format</td>
<td>3.7.3</td>
<td>–</td>
</tr>
<tr>
<td>COLOR_TABLE_WIDTH</td>
<td>2 × 3 × Z&lt;sup&gt;+&lt;/sup&gt;</td>
<td>GetColorTable-Parameteriv</td>
<td>0</td>
<td>Color tables’ specified width</td>
<td>3.7.3</td>
<td>–</td>
</tr>
<tr>
<td>COLOR_TABLE_&lt;sub&gt;x&lt;/sub&gt;_&lt;sub&gt;SIZE&lt;/sub&gt;</td>
<td>6 × 2 × 3 × Z&lt;sup&gt;+&lt;/sup&gt;</td>
<td>GetColorTable-Parameteriv</td>
<td>0</td>
<td>Color table component resolution; x is RED, GREEN, BLUE, ALPHA, LUMINANCE, or INTENSITY</td>
<td>3.7.3</td>
<td>–</td>
</tr>
<tr>
<td>COLOR_TABLE_SCALE</td>
<td>3 × R&lt;sup&gt;4&lt;/sup&gt;</td>
<td>GetColorTable-Parameteriv</td>
<td>1,1,1,1</td>
<td>Scale factors applied to color table entries</td>
<td>3.7.3</td>
<td>pixel</td>
</tr>
<tr>
<td>COLOR_TABLE_BIAS</td>
<td>3 × R&lt;sup&gt;4&lt;/sup&gt;</td>
<td>GetColorTable-Parameteriv</td>
<td>0,0,0,0</td>
<td>Bias factors applied to color table entries</td>
<td>3.7.3</td>
<td>pixel</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Initial Value</td>
<td>Description</td>
<td>Sec.</td>
<td>Attribute</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------</td>
<td>---------------------</td>
<td>---------------</td>
<td>--------------------------------------------------</td>
<td>------</td>
<td>----------------------</td>
</tr>
<tr>
<td>CONVOLUTION 1D</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>True if 1D convolution is done</td>
<td>3.7.3</td>
<td>pixel/enable</td>
</tr>
<tr>
<td>CONVOLUTION 2D</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>True if 2D convolution is done</td>
<td>3.7.3</td>
<td>pixel/enable</td>
</tr>
<tr>
<td>SEPARABLE 2D</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>True if separable 2D convolution is done</td>
<td>3.7.3</td>
<td>pixel/enable</td>
</tr>
<tr>
<td>CONVOLUTION xD</td>
<td>$2 \times I$</td>
<td>GetConvolution-Filter</td>
<td>empty</td>
<td>Convolution filters; $x$ is 1 or 2</td>
<td>3.7.3</td>
<td>–</td>
</tr>
<tr>
<td>SEPARABLE 3D</td>
<td>$2 \times I$</td>
<td>GetSeparable-Filter</td>
<td>empty</td>
<td>Separable convolution filter</td>
<td>3.7.3</td>
<td>–</td>
</tr>
<tr>
<td>CONVOLUTION BORDER COLOR</td>
<td>$3 \times C$</td>
<td>GetConvolution-Parameterfv</td>
<td>0,0,0,0</td>
<td>Convolution border color</td>
<td>3.7.5</td>
<td>pixel</td>
</tr>
<tr>
<td>CONVOLUTION BORDER MODE</td>
<td>$3 \times Z_4$</td>
<td>GetConvolution-Parameterfv</td>
<td>REDUCE</td>
<td>Convolution border mode</td>
<td>3.7.5</td>
<td>pixel</td>
</tr>
<tr>
<td>CONVOLUTION FILTER SCALE</td>
<td>$3 \times R^4$</td>
<td>GetConvolution-Parameterfv</td>
<td>1,1,1</td>
<td>Scale factors applied to convolution filter entries</td>
<td>3.7.3</td>
<td>pixel</td>
</tr>
<tr>
<td>CONVOLUTION FILTER BIAS</td>
<td>$3 \times R^4$</td>
<td>GetConvolution-Parameterfv</td>
<td>0,0,0,0</td>
<td>Bias factors applied to convolution filter entries</td>
<td>3.7.3</td>
<td>pixel</td>
</tr>
<tr>
<td>CONVOLUTION FORMAT</td>
<td>$3 \times Z_{12}$</td>
<td>GetConvolution-Parameterfv</td>
<td>RGBA</td>
<td>Convolution filter internal format</td>
<td>3.7.5</td>
<td>–</td>
</tr>
<tr>
<td>CONVOLUTION WIDTH</td>
<td>$3 \times Z^+$</td>
<td>GetConvolution-Parameterfv</td>
<td>0</td>
<td>Convolution filter width</td>
<td>3.7.5</td>
<td>–</td>
</tr>
<tr>
<td>CONVOLUTION HEIGHT</td>
<td>$2 \times Z^+$</td>
<td>GetConvolution-Parameterfv</td>
<td>0</td>
<td>Convolution filter height</td>
<td>3.7.5</td>
<td>–</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Initial Value</td>
<td>Description</td>
<td>Sec.</td>
<td>Attribute</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>------</td>
<td>-------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
<td>-----------</td>
</tr>
<tr>
<td>POST_CONVOLUTION_x_SCALE</td>
<td>R</td>
<td>GetFloatv</td>
<td>1</td>
<td>Component scale factors after convolution; ( x ) is RED, GREEN, BLUE, or ALPHA</td>
<td>3.7.3</td>
<td>pixel</td>
</tr>
<tr>
<td>POST_CONVOLUTION_x_BIAS</td>
<td>R</td>
<td>GetFloatv</td>
<td>0</td>
<td>Component bias factors after convolution</td>
<td>3.7.3</td>
<td>pixel</td>
</tr>
<tr>
<td>POST_COLOR_MATRIX_x_SCALE</td>
<td>R</td>
<td>GetFloatv</td>
<td>1</td>
<td>Component scale factors after color matrix</td>
<td>3.7.3</td>
<td>pixel</td>
</tr>
<tr>
<td>POST_COLOR_MATRIX_x_BIAS</td>
<td>R</td>
<td>GetFloatv</td>
<td>0</td>
<td>Component bias factors after color matrix</td>
<td>3.7.3</td>
<td>pixel</td>
</tr>
<tr>
<td>HISTOGRAM</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>True if histogramming is enabled</td>
<td>3.7.3</td>
<td>pixel/enable</td>
</tr>
<tr>
<td>HISTOGRAMWIDTH</td>
<td>I</td>
<td>GetHistogram</td>
<td>empty</td>
<td>Histogram table</td>
<td>3.7.3</td>
<td>–</td>
</tr>
<tr>
<td>HISTOGRAM_FORMAT</td>
<td>2 \times Z^+</td>
<td>GetHistogram-Parameteriv</td>
<td>RGBA</td>
<td>Histogram table internal format</td>
<td>3.7.3</td>
<td>–</td>
</tr>
<tr>
<td>HISTOGRAM_x_SIZE</td>
<td>5 \times 2 \times Z^+</td>
<td>GetHistogram-Parameteriv</td>
<td>0</td>
<td>Histogram table component resolution; ( x ) is RED, GREEN, BLUE, ALPHA, or LUMINANCE</td>
<td>3.7.3</td>
<td>–</td>
</tr>
<tr>
<td>HISTOGRAM_SINK</td>
<td>B</td>
<td>GetHistogram-Parameteriv</td>
<td>FALSE</td>
<td>True if histogramming consumes pixel groups</td>
<td>3.7.3</td>
<td>–</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Initial Value</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
<td>-------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MINMAX</td>
<td>B</td>
<td>GetIsEnabled</td>
<td>FALSE</td>
<td>True if minmax is enabled</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GetMinmax</td>
<td>(M,M,M,M),(m,m,m,m)</td>
<td>Minmax table</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GetMinmaxFormat</td>
<td>RGB</td>
<td>Minmax table internal format</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GetMinmaxParameter1</td>
<td>TRUE</td>
<td>True if minmax consumes pixel groups</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GetMinmaxParameter2</td>
<td>TRUE</td>
<td>TRUE if minmax consumes pixel groups</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GetMinmaxParameter3</td>
<td>TRUE</td>
<td>TRUE if minmax consumes pixel groups</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GetMinmaxParameter4</td>
<td>TRUE</td>
<td>TRUE if minmax consumes pixel groups</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>GetMinmaxParameter5</td>
<td>TRUE</td>
<td>TRUE if minmax consumes pixel groups</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIZE</td>
<td>Z</td>
<td>GetIntegerv</td>
<td>1</td>
<td>Size of table</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.37. Pixels (cont.)
### Table 6.38. Evaluators (Get\(\text{Map}\) takes a map name)

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORDER</td>
<td>(9 \times Z_{8^*})</td>
<td>GetMapiv</td>
<td>1</td>
<td>1d map order</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>ORDER</td>
<td>(9 \times 2 \times Z_{8^*})</td>
<td>GetMapiv</td>
<td>1,1</td>
<td>2d map orders</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>COEFF</td>
<td>(9 \times 8 \times R^n)</td>
<td>GetMapfv</td>
<td>see 5.1</td>
<td>1d control points</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>COEFF</td>
<td>(9 \times 8 \times 8 \times R^n)</td>
<td>GetMapfv</td>
<td>see 5.1</td>
<td>2d control points</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>DOMAIN</td>
<td>(9 \times 2 \times R)</td>
<td>GetMapfv</td>
<td>see 5.1</td>
<td>1d domain endpoints</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>DOMAIN</td>
<td>(9 \times 4 \times R)</td>
<td>GetMapfv</td>
<td>see 5.1</td>
<td>2d domain endpoints</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>(\text{MAP}_1.x)</td>
<td>(9 \times B)</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>1d map enables: (x) is map type</td>
<td>5.1</td>
<td>eval/enable</td>
</tr>
<tr>
<td>(\text{MAP}_2.x)</td>
<td>(9 \times B)</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>2d map enables: (x) is map type</td>
<td>5.1</td>
<td>eval/enable</td>
</tr>
<tr>
<td>(\text{MAP}_1,\text{GRID},\text{DOMAIN})</td>
<td>(2 \times R)</td>
<td>GetFloatv</td>
<td>0,1</td>
<td>1d grid endpoints</td>
<td>5.1</td>
<td>eval</td>
</tr>
<tr>
<td>(\text{MAP}_2,\text{GRID},\text{DOMAIN})</td>
<td>(4 \times R)</td>
<td>GetFloatv</td>
<td>0,1;0,1</td>
<td>2d grid endpoints</td>
<td>5.1</td>
<td>eval</td>
</tr>
<tr>
<td>(\text{MAP}_1,\text{GRID},\text{SEGMENTS})</td>
<td>(Z^+)</td>
<td>GetFloatv</td>
<td>1</td>
<td>1d grid divisions</td>
<td>5.1</td>
<td>eval</td>
</tr>
<tr>
<td>(\text{MAP}_2,\text{GRID},\text{SEGMENTS})</td>
<td>(2 \times Z^+)</td>
<td>GetFloatv</td>
<td>1,1</td>
<td>2d grid divisions</td>
<td>5.1</td>
<td>eval</td>
</tr>
<tr>
<td>(\text{AUTO},\text{NORMAL})</td>
<td>(B)</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>True if automatic normal generation enabled</td>
<td>5.1</td>
<td>eval/enable</td>
</tr>
</tbody>
</table>
### Table 6.39. Shader Object State

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>SEC.</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetShaderiv</td>
<td>FALSE</td>
<td>Type of shader (vertex or fragment)</td>
<td>2.20.1</td>
<td>Z</td>
</tr>
<tr>
<td>GetDelete</td>
<td>FALSE</td>
<td>Shader flagged for deletion</td>
<td>2.20.1</td>
<td>-</td>
</tr>
<tr>
<td>GetStatus</td>
<td>FALSE</td>
<td>Last compile succeeded</td>
<td>6.1.15</td>
<td>-</td>
</tr>
<tr>
<td>GetLog</td>
<td>empty string</td>
<td>Info log for shader objects</td>
<td>6.1.15</td>
<td>-</td>
</tr>
<tr>
<td>GetSource</td>
<td>empty string</td>
<td>Source code for a shader</td>
<td>6.1.15</td>
<td>-</td>
</tr>
<tr>
<td>GetLength</td>
<td>0</td>
<td>Length of info log</td>
<td>6.1.15</td>
<td>-</td>
</tr>
<tr>
<td>GetSourceLength</td>
<td>0</td>
<td>Length of source code</td>
<td>6.1.15</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>SEC.</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetShaderiv</td>
<td></td>
<td>Shader type</td>
<td>6.1.15</td>
<td>S</td>
</tr>
<tr>
<td>GetDelete</td>
<td></td>
<td>Shader deleted</td>
<td>6.1.15</td>
<td>Z</td>
</tr>
<tr>
<td>GetStatus</td>
<td></td>
<td>Shader status</td>
<td>6.1.15</td>
<td>Z</td>
</tr>
<tr>
<td>GetLog</td>
<td></td>
<td>Shader log</td>
<td>6.1.15</td>
<td>Z</td>
</tr>
<tr>
<td>GetSource</td>
<td></td>
<td>Shader source</td>
<td>6.1.15</td>
<td>Z</td>
</tr>
<tr>
<td>GetLength</td>
<td></td>
<td>Shader length</td>
<td>6.1.15</td>
<td>Z</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Initial Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------</td>
<td>-----------------</td>
<td>---------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>CURRENT_PROGRAM</td>
<td>$Z^+$</td>
<td>GetInteger</td>
<td>0</td>
<td>Name of current program object</td>
</tr>
<tr>
<td>DELETE_STATUS</td>
<td>$B$</td>
<td>GetProgramiv</td>
<td>FALSE</td>
<td>Program object deleted</td>
</tr>
<tr>
<td>LINK_STATUS</td>
<td>$B$</td>
<td>GetProgramiv</td>
<td>FALSE</td>
<td>Last link attempt succeeded</td>
</tr>
<tr>
<td>VALIDATE_STATUS</td>
<td>$B$</td>
<td>GetProgramiv</td>
<td>FALSE</td>
<td>Last validate attempt succeeded</td>
</tr>
<tr>
<td>ATTACHED_SHADER</td>
<td>$Z^+$</td>
<td>GetProgramiv</td>
<td>0</td>
<td>Number of attached shader objects</td>
</tr>
<tr>
<td>-</td>
<td>0 + $Z^+$</td>
<td>GetAttachedShaders</td>
<td>empty</td>
<td>Shader objects attached</td>
</tr>
<tr>
<td>-</td>
<td>$S$</td>
<td>GetProgramInfoLog</td>
<td>empty</td>
<td>Info log for program object</td>
</tr>
<tr>
<td>INFOLOG_LENGTH</td>
<td>$Z^+$</td>
<td>GetProgramiv</td>
<td>0</td>
<td>Length of info log</td>
</tr>
<tr>
<td>ACTIVE_UNIFORMS</td>
<td>$Z^+$</td>
<td>GetProgramiv</td>
<td>0</td>
<td>Number of active uniforms</td>
</tr>
<tr>
<td>-</td>
<td>0 + $Z$</td>
<td>GetUniformLocation</td>
<td>–</td>
<td>Location of active uniforms</td>
</tr>
<tr>
<td>-</td>
<td>0 + $Z^+$</td>
<td>GetActiveUniform</td>
<td>–</td>
<td>Size of active uniform</td>
</tr>
<tr>
<td>-</td>
<td>0 + $Z^+$</td>
<td>GetActiveUniform</td>
<td>–</td>
<td>Type of active uniform</td>
</tr>
<tr>
<td>-</td>
<td>0 + char</td>
<td>GetActiveUniform</td>
<td>empty</td>
<td>Name of active uniform</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>
</tr>
<tr>
<td>-</td>
<td>512 + $R$</td>
<td>GetUniform</td>
<td>0</td>
<td>Uniform value</td>
</tr>
<tr>
<td>ACTIVE_ATTRIBUTES</td>
<td>$Z^+$</td>
<td>GetProgramiv</td>
<td>0</td>
<td>Number of active attributes</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Initial Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>---------------------------</td>
<td>---------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>-</td>
<td>0 × Z</td>
<td>GetAttribLocation</td>
<td>Z</td>
<td>Location of active generic attribute</td>
</tr>
<tr>
<td>-</td>
<td>0 × Z^+</td>
<td>GetActiveAttrib</td>
<td>Z</td>
<td>Size of active attribute</td>
</tr>
<tr>
<td>-</td>
<td>0 × Z^+</td>
<td>GetActiveAttrib</td>
<td>Z</td>
<td>Type of active attribute</td>
</tr>
<tr>
<td>-</td>
<td>0 × char</td>
<td>GetActiveAttrib</td>
<td>empty</td>
<td>Name of active attribute</td>
</tr>
<tr>
<td>ACTIVE,ATTRIBUTE,MAX,LENGTH</td>
<td>Z^+</td>
<td>GetProgramiv</td>
<td>0</td>
<td>Maximum active attribute name length</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>
</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>
</tr>
<tr>
<td>TRANSFORM,FEEDBACK,VARYING,MAX,LENGTH</td>
<td>Z^+</td>
<td>GetProgramiv</td>
<td>0</td>
<td>Maximum transform varying name length</td>
</tr>
<tr>
<td>-</td>
<td>Z^+</td>
<td>GetTransformFeedbackVarying</td>
<td>-</td>
<td>Size of each transform feedback varying variable</td>
</tr>
<tr>
<td>-</td>
<td>Z^+</td>
<td>GetTransformFeedbackVarying</td>
<td>-</td>
<td>Type of each transform feedback varying variable</td>
</tr>
<tr>
<td>-</td>
<td>0^+ × char</td>
<td>GetTransformFeedbackVarying</td>
<td>-</td>
<td>Name of each transform feedback varying variable</td>
</tr>
<tr>
<td>Get value</td>
<td>Type</td>
<td>Get Command</td>
<td>Initial Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>-------------</td>
<td>---------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>VERTEX, PROGRAM, TWO-SIDE</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Two-sided color mode</td>
</tr>
<tr>
<td>CURRENT, VERTEX, ATTRIB</td>
<td>16 * × R4</td>
<td>GetVertexAttrib</td>
<td>0,0,0,1</td>
<td>Generic vertex attribute</td>
</tr>
<tr>
<td>VERTEX, PROGRAM, POINT-SIZE</td>
<td>B</td>
<td>IsEnabled</td>
<td>FALSE</td>
<td>Point size mode</td>
</tr>
</tbody>
</table>
Table 6.43. Query Object State
<table>
<thead>
<tr>
<th>Sec.</th>
<th>Attribute Description</th>
<th>Initial Value</th>
<th>Get Command</th>
<th>Type</th>
<th>Get Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1.13</td>
<td>Buffer object bound to generic bind point for transform feedback</td>
<td>0</td>
<td>GetIntegerv</td>
<td>$\mathbb{Z}^+$</td>
<td>$\mathbb{Z}^+$</td>
</tr>
<tr>
<td>6.1.13</td>
<td>Buffer object bound to each transform feedback attribute stream</td>
<td>0</td>
<td>GetIntegerv</td>
<td>$\mathbb{Z}^+$</td>
<td>$\mathbb{Z}^+$</td>
</tr>
<tr>
<td>6.1.13</td>
<td>Start offset of binding range for each transform feedback attribute stream</td>
<td>0</td>
<td>GetIntegerv</td>
<td>$\mathbb{Z}^+$</td>
<td>$\mathbb{Z}^+$</td>
</tr>
<tr>
<td>6.1.13</td>
<td>Size of binding range for each transform feedback attribute stream</td>
<td>0</td>
<td>GetIntegerv</td>
<td>$\mathbb{Z}^+$</td>
<td>$\mathbb{Z}^+$</td>
</tr>
<tr>
<td>2.15</td>
<td>Max number of components to write to a single buffer in interleaved mode</td>
<td>64</td>
<td>GetIntegerv</td>
<td>$\mathbb{Z}^+$</td>
<td>$\mathbb{Z}^+$</td>
</tr>
<tr>
<td>2.15</td>
<td>Max number of separate attributes or varings that can be captured in transform feedback</td>
<td>4</td>
<td>GetIntegerv</td>
<td>$\mathbb{Z}^+$</td>
<td>$\mathbb{Z}^+$</td>
</tr>
<tr>
<td>2.15</td>
<td>Max number of components per attribute or varying in separate mode</td>
<td>4</td>
<td>GetIntegerv</td>
<td>$\mathbb{Z}^+$</td>
<td>$\mathbb{Z}^+$</td>
</tr>
</tbody>
</table>

Table 6.44. Transform Feedback State

Version 3.0 (with changes marked - August 11, 2008)
### Table 6.45. Hints

<table>
<thead>
<tr>
<th>Command</th>
<th>Value</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PERSPECTIVE-CORRECTION-HINT</td>
<td>GETINTEGER</td>
<td>(Z_3)</td>
<td>Perspective correction hint</td>
</tr>
<tr>
<td>POINT-SMOOTH-HINT</td>
<td>GETINTEGER</td>
<td>(Z_3)</td>
<td>Point smooth hint</td>
</tr>
<tr>
<td>LINE-SMOOTH-HINT</td>
<td>GETINTEGER</td>
<td>(Z_3)</td>
<td>Line smooth hint</td>
</tr>
<tr>
<td>POLYGON-SMOOTH-HINT</td>
<td>GETINTEGER</td>
<td>(Z_3)</td>
<td>Polygon smooth hint</td>
</tr>
<tr>
<td>FOG-HINT</td>
<td>GETINTEGER</td>
<td>(Z_3)</td>
<td>Fog hint</td>
</tr>
<tr>
<td>MIPMAP-QUALITY-HINT</td>
<td>GETINTEGER</td>
<td>(Z_3)</td>
<td>Mipmap generation hint</td>
</tr>
<tr>
<td>TEXTURE-COMPRESSION-HINT</td>
<td>GETINTEGER</td>
<td>(Z_3)</td>
<td>Texture compression hint</td>
</tr>
<tr>
<td>FRAGMENT-SHADER-DERIVATIVE-HINT</td>
<td>GETINTEGER</td>
<td>(Z_3)</td>
<td>Fragment shader derivative hint</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sec.</th>
<th>Attribute</th>
<th>Initial Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6</td>
<td>hint</td>
<td>DONT-CARE</td>
<td>Perspective correction hint</td>
</tr>
<tr>
<td>3.6</td>
<td>hint</td>
<td>DONT-CARE</td>
<td>Point smooth hint</td>
</tr>
<tr>
<td>3.6</td>
<td>hint</td>
<td>DONT-CARE</td>
<td>Line smooth hint</td>
</tr>
<tr>
<td>3.6</td>
<td>hint</td>
<td>DONT-CARE</td>
<td>Polygon smooth hint</td>
</tr>
<tr>
<td>3.6</td>
<td>hint</td>
<td>DONT-CARE</td>
<td>Fog hint</td>
</tr>
<tr>
<td>3.6</td>
<td>hint</td>
<td>DONT-CARE</td>
<td>Mipmap generation hint</td>
</tr>
<tr>
<td>3.6</td>
<td>hint</td>
<td>DONT-CARE</td>
<td>Texture compression hint</td>
</tr>
<tr>
<td>3.6</td>
<td>hint</td>
<td>DONT-CARE</td>
<td>Fragment shader derivative hint</td>
</tr>
</tbody>
</table>

Version 3.0 (with changes marked - August 11, 2008)
### 6.2. STATE TABLES

<table>
<thead>
<tr>
<th>Get Command</th>
<th>Minimum Value</th>
<th>Description</th>
<th>Type</th>
<th>Get value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetIntegerv</td>
<td>8</td>
<td>Maximum number of lights</td>
<td>Z+</td>
<td>MAXLIGHTS</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>6</td>
<td>Maximum number of user clipping planes</td>
<td>Z+</td>
<td>MAXCLIPPLANE</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>32</td>
<td>Maximum number of color matrix stack depth</td>
<td>Z+</td>
<td>MAXCOLORSTACKDEPTH</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>2</td>
<td>Maximum number of model-view stack depth</td>
<td>Z+</td>
<td>MAXMODELVIEWSTACKDEPTH</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>2</td>
<td>Maximum number of projection matrix stack depth</td>
<td>Z+</td>
<td>MAXPROJECTIONSTACKDEPTH</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>4</td>
<td>Maximum number of texture matrix stack depth</td>
<td>Z+</td>
<td>MAXTEXTURESTACKDEPTH</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>256</td>
<td>Maximum number of subpixel precision in screen</td>
<td>Z+</td>
<td>MAXSUBPIXELBITS</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>1024</td>
<td>Maximum number of 2D/1D texture layers</td>
<td>Z+</td>
<td>MAXTEXTURELAYER</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>256</td>
<td>Maximum number of 3D texture levels</td>
<td>Z+</td>
<td>MAXTEXTURELAYER.texture_level</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>3.91</td>
<td>Maximum number of layers for texture arrays</td>
<td>Z+</td>
<td>MAXTEXTURELAYER.texture_array</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>3.91</td>
<td>Maximum absolute texture level of detail bias</td>
<td>Z+</td>
<td>MAXTEXTURELAYER.texture_level.bias</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>3.97</td>
<td>Maximum absolute texture level of detail bias</td>
<td>Z+</td>
<td>MAXTEXTURELAYER.texture_level.bias.2</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>1024</td>
<td>Maximum absolute texture level of detail bias</td>
<td>Z+</td>
<td>MAXTEXTURELAYER.texture_level.bias.3</td>
</tr>
<tr>
<td>GetFloatv</td>
<td>2.0</td>
<td>Maximum number of details</td>
<td>R+</td>
<td>MAXDETAILS</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>256</td>
<td>Maximum number of layers for texture arrays</td>
<td>Z+</td>
<td>MAXTEXTURELAYER.texture_array.2</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>1024</td>
<td>Maximum number of layers for texture arrays</td>
<td>Z+</td>
<td>MAXTEXTURELAYER.texture_array.3</td>
</tr>
<tr>
<td>GetIntegerv</td>
<td>1024</td>
<td>Maximum number of layers for texture arrays</td>
<td>Z+</td>
<td>MAXTEXTURELAYER.texture_array.4</td>
</tr>
</tbody>
</table>

Table 6.46. Implementation Dependent Values

Version 3.0 (with changes marked - August 11, 2008)
<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Minimum Value</th>
<th>Description</th>
<th>Sec.</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXPIXELMAPTABLE</td>
<td>Z⁺</td>
<td>GetIntegerv</td>
<td>32</td>
<td>Maximum size of a PixelMap translation table</td>
<td>3.7.3</td>
<td>–</td>
</tr>
<tr>
<td>MAXNAMESTACKDEPTH</td>
<td>Z⁺</td>
<td>GetIntegerv</td>
<td>64</td>
<td>Maximum selection name stack depth</td>
<td>5.2</td>
<td>–</td>
</tr>
<tr>
<td>MAXLISTNESTING</td>
<td>Z⁺</td>
<td>GetIntegerv</td>
<td>64</td>
<td>Maximum display list call nesting</td>
<td>5.4</td>
<td>–</td>
</tr>
<tr>
<td>MAXEVALORDER</td>
<td>Z⁺</td>
<td>GetIntegerv</td>
<td>8</td>
<td>Maximum evaluator polynomial order</td>
<td>5.1</td>
<td>–</td>
</tr>
<tr>
<td>MAXVIEWPORTDIMS</td>
<td>2×Z⁺</td>
<td>GetIntegerv</td>
<td>see 2.12.1</td>
<td>Maximum viewport dimensions</td>
<td>2.12.1</td>
<td>–</td>
</tr>
<tr>
<td>MAXATTRIBSTACKDEPTH</td>
<td>Z⁺</td>
<td>GetIntegerv</td>
<td>16</td>
<td>Maximum depth of the server attribute stack</td>
<td>6</td>
<td>–</td>
</tr>
<tr>
<td>MAXCLIENTATTRIBSTACKDEPTH</td>
<td>Z⁺</td>
<td>GetIntegerv</td>
<td>16</td>
<td>Maximum depth of the client attribute stack</td>
<td>6</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Z⁺</td>
<td></td>
<td>32</td>
<td>Max. size of a color table</td>
<td>3.7.3</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Z⁺</td>
<td></td>
<td>32</td>
<td>Max. size of the histogram table</td>
<td>3.7.3</td>
<td>–</td>
</tr>
<tr>
<td>ALIASEDPONTSIZERANGE</td>
<td>2×R⁺</td>
<td>GetFloatv</td>
<td>1,1</td>
<td>Range (lo to hi) of aliased point sizes</td>
<td>3.4</td>
<td>–</td>
</tr>
<tr>
<td>SMOOTHPOINTSIZEGRANULARITY</td>
<td>2×R⁺</td>
<td>GetFloatv</td>
<td>1,1</td>
<td>Range (lo to hi) of antialiased point sizes</td>
<td>3.4</td>
<td>–</td>
</tr>
<tr>
<td>SMOOTHPOINTSIZEGRANULARITY</td>
<td>R⁺</td>
<td>GetFloatv</td>
<td>–</td>
<td>Antialiased point size granularity</td>
<td>3.4</td>
<td>–</td>
</tr>
</tbody>
</table>
### 6.2. STATE TABLES

<table>
<thead>
<tr>
<th>Minimum Value</th>
<th>Command</th>
<th>Type</th>
<th>Description</th>
<th>Get</th>
<th>Get</th>
<th>Table 6.48. Implementation Dependent Values (cont.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIASED LINE WIDTH RANGE</td>
<td>GetFloatv</td>
<td>$2 \times R^+$</td>
<td>Range (lo to hi) of aliased line widths</td>
<td>GetFloatv</td>
<td>$2 \times R^+$</td>
<td>$R^+$</td>
</tr>
<tr>
<td>SMOOTH LINE WIDTH RANGE</td>
<td>GetFloatv</td>
<td>$3 \times Z^+$</td>
<td>Range (lo to hi) of anti-aliased line widths</td>
<td>GetFloatv</td>
<td>$3 \times Z^+$</td>
<td>$Z^+$</td>
</tr>
<tr>
<td>SMOOTH LINE WIDTH OR ANISOTROPY</td>
<td>GetFloatv</td>
<td>$2 \times R^+$</td>
<td>Anti-aliased line width granularity</td>
<td>GetFloatv</td>
<td>$2 \times Z^+$</td>
<td>$Z^+$</td>
</tr>
<tr>
<td>MAX CONVOLUTION WIDTH</td>
<td>GetConvolutionParameteriv</td>
<td>$3 \times Z^+$</td>
<td>Maximum width of convolution filter</td>
<td>GetConvolutionParameteriv</td>
<td>$2 \times Z^+$</td>
<td>$Z^+$</td>
</tr>
<tr>
<td>MAX CONVOLUTION HEIGHT</td>
<td>GetConvolutionParameteriv</td>
<td>$3 \times Z^+$</td>
<td>Maximum height of convolution filter</td>
<td>GetConvolutionParameteriv</td>
<td>$2 \times Z^+$</td>
<td>$Z^+$</td>
</tr>
<tr>
<td>MAX ELEMENTS INDICES</td>
<td>GetIntegerv</td>
<td>$Z$</td>
<td>Recommended max. number of DrawRangeElements indices</td>
<td>GetIntegerv</td>
<td>$0 \times Z$</td>
<td>$Z$</td>
</tr>
<tr>
<td>MAX ELEMENTS VERTICES</td>
<td>GetIntegerv</td>
<td>$Z$</td>
<td>Recommended max. number of DrawRangeElements vertices</td>
<td>GetIntegerv</td>
<td>$0 \times Z$</td>
<td>$Z$</td>
</tr>
<tr>
<td>COMPRESSED TEXTURE FORMATS</td>
<td>GetIntegerv</td>
<td>$Z$</td>
<td>Enumerated compressed texture formats</td>
<td>GetIntegerv</td>
<td>$0 \times Z$</td>
<td>$Z$</td>
</tr>
<tr>
<td>NON COMPRESSED TEXTURE FORMATS</td>
<td>GetIntegerv</td>
<td>$Z$</td>
<td>Number of compressed texture formats</td>
<td>GetIntegerv</td>
<td>$0 \times Z$</td>
<td>$Z$</td>
</tr>
</tbody>
</table>

Version 3.0 (with changes marked - August 11, 2008)
### 6.2. STATE TABLES

<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>( 2 \times Z )</td>
<td>GetQueryiv</td>
<td>see 6.1.12</td>
<td>Asynchronous counter bits</td>
<td>6.1.12</td>
</tr>
<tr>
<td>( 0 + \times Z )</td>
<td>GetString</td>
<td></td>
<td>Supported individual extension names</td>
<td>6.1.11</td>
</tr>
<tr>
<td>( Z^+ )</td>
<td>GetIntegerv</td>
<td></td>
<td>Major version number supported</td>
<td>6.1.11</td>
</tr>
<tr>
<td>( S^+ )</td>
<td>GetString</td>
<td></td>
<td>Context full/forward compatible flag</td>
<td>6.1.11</td>
</tr>
<tr>
<td>( Z^+ )</td>
<td>GetIntegerv</td>
<td></td>
<td>Supported extension names</td>
<td>6.1.11</td>
</tr>
<tr>
<td>( S^+ )</td>
<td>GetString</td>
<td></td>
<td>Render string</td>
<td>6.1.11</td>
</tr>
<tr>
<td>( Z^+ )</td>
<td>GetIntegerv</td>
<td></td>
<td>Shading Language version supported</td>
<td>6.1.11</td>
</tr>
<tr>
<td>( S^+ )</td>
<td>GetString</td>
<td></td>
<td>Vendor string</td>
<td>6.1.11</td>
</tr>
<tr>
<td>( S^+ )</td>
<td>GetString</td>
<td></td>
<td>OpenGL version supported</td>
<td>6.1.11</td>
</tr>
</tbody>
</table>

**Table 6.49. Implementation Dependent Values (cont.)**

Version 3.0 (with changes marked - August 11, 2008)
<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Minimum Value</th>
<th>Description</th>
<th>Sec.</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_TEXTURE_UNITS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>2</td>
<td>Number of fixed-function texture units</td>
<td>2.6</td>
<td>–</td>
</tr>
<tr>
<td>MAX_VERTEX_ATTRIBS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>16</td>
<td>Number of active vertex attributes</td>
<td>2.7</td>
<td>–</td>
</tr>
<tr>
<td>MAX_VERTEX_UNIFORM_COMPONENTS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>1024</td>
<td>Number of components for vertex shader uniform variables</td>
<td>2.20.3</td>
<td>–</td>
</tr>
<tr>
<td>MAX_VARYING_COMPONENTS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>64</td>
<td>Number of components for varying variables</td>
<td>2.20.3</td>
<td>–</td>
</tr>
<tr>
<td>MAX_COMBINED_TEXTURE_IMAGE_UNITS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>16</td>
<td>Total number of texture units accessible by the GL</td>
<td>2.20.4</td>
<td>–</td>
</tr>
<tr>
<td>MAX_VERTEX_TEXTURE_IMAGE_UNITS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>16</td>
<td>Number of texture image units accessible by a vertex shader</td>
<td>2.20.4</td>
<td>–</td>
</tr>
<tr>
<td>MAX_TEXTURE_IMAGE_UNITS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>16</td>
<td>Number of texture image units accessible by fragment processing</td>
<td>2.20.4</td>
<td>–</td>
</tr>
<tr>
<td>MAX_TEXTURE_COORDS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>8</td>
<td>Number of texture coordinate sets</td>
<td>2.7</td>
<td>–</td>
</tr>
<tr>
<td>MAX_FRAGMENT_UNIFORM_COMPONENTS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>1024</td>
<td>Number of components for frag. shader uniform variables</td>
<td>3.12.1</td>
<td>–</td>
</tr>
<tr>
<td>MIN_PROGRAM_TEXEL_OFFSET</td>
<td>Z</td>
<td>GetIntegerv</td>
<td>-8</td>
<td>Minimum texel offset allowed in lookup</td>
<td>2.20.4</td>
<td>–</td>
</tr>
<tr>
<td>MAX_PROGRAM_TEXEL_OFFSET</td>
<td>Z</td>
<td>GetIntegerv</td>
<td>7</td>
<td>Maximum texel offset allowed in lookup</td>
<td>2.20.4</td>
<td>–</td>
</tr>
<tr>
<td>Get Command</td>
<td>Minimum Value</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>---------------</td>
<td>-------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AUX BUFFERS</td>
<td>0</td>
<td>Number of auxiliary buffers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAX DRAW BUFFERS</td>
<td>8*</td>
<td>Maximum number of active draw buffers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RGBA MODE</td>
<td>B</td>
<td>GetBooleanv</td>
<td>True if color buffers store RGBA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDEX MODE</td>
<td>B</td>
<td>GetBooleanv</td>
<td>True if color buffers store indexes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOUBLEBUFFER</td>
<td>B</td>
<td>GetBooleanv</td>
<td>True if front &amp; back buffers exist</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STEREO</td>
<td>B</td>
<td>GetBooleanv</td>
<td>True if left &amp; right buffers exist</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAMPLES</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>Number of multisample buffers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAX COLOR ATTACHMENTS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>Maximum number of FBO attachment points for color buffers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAX SAMPLES</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>Maximum number of samples supported for multisampling</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x BITS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>Number of bits in x color buffer component (x is one of RED, GREEN, BLUE, ALPHA, or INDEX)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEPTH</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>Number of depth buffer planes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STENCIL</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>Number of stencil planes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACCUM x BITS</td>
<td>Z+</td>
<td>GetIntegerv</td>
<td>Number of bits in x accumulation buffer component (x is RED, GREEN, BLUE, or ALPHA)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.51. Framebuffer Dependent Values
### Table 6.52. Miscellaneous

<table>
<thead>
<tr>
<th>Get value</th>
<th>Type</th>
<th>Get Command</th>
<th>Initial Value</th>
<th>Description</th>
<th>Sec.</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST.BASE</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Setting of <code>ListBase</code></td>
<td>5.4</td>
<td>list</td>
</tr>
<tr>
<td>LIST.INDEX</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Number of display list under construction; 0 if none</td>
<td>5.4</td>
<td>–</td>
</tr>
<tr>
<td>LIST.MODE</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Mode of display list under construction; undefined if none</td>
<td>5.4</td>
<td>–</td>
</tr>
<tr>
<td>–</td>
<td>$16 \times A$</td>
<td>–</td>
<td>empty</td>
<td>Server attribute stack</td>
<td>6</td>
<td>–</td>
</tr>
<tr>
<td>ATTRIB_STACK_DEPTH</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Server attribute stack pointer</td>
<td>6</td>
<td>–</td>
</tr>
<tr>
<td>–</td>
<td>$16 \times A$</td>
<td>–</td>
<td>empty</td>
<td>Client attribute stack</td>
<td>6</td>
<td>–</td>
</tr>
<tr>
<td>CLIENT_ATTRIB_STACK_DEPTH</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Client attribute stack pointer</td>
<td>6</td>
<td>–</td>
</tr>
<tr>
<td>NAME_STACK_DEPTH</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Name stack depth</td>
<td>5.2</td>
<td>–</td>
</tr>
<tr>
<td>RENDER_MODE</td>
<td>$Z_3$</td>
<td>GetIntegerv</td>
<td>RENDER</td>
<td><code>RenderMode</code> setting</td>
<td>5.2</td>
<td>–</td>
</tr>
<tr>
<td>SELECTION_BUFFER_POINTER</td>
<td>$Y$</td>
<td>GetPointer</td>
<td>0</td>
<td>Selection buffer pointer</td>
<td>5.2</td>
<td>select</td>
</tr>
<tr>
<td>SELECTION_BUFFER_SIZE</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Selection buffer size</td>
<td>5.2</td>
<td>select</td>
</tr>
<tr>
<td>FEEDBACK_BUFFER_POINTER</td>
<td>$Y$</td>
<td>GetPointer</td>
<td>0</td>
<td>Feedback buffer pointer</td>
<td>5.3</td>
<td>feedback</td>
</tr>
<tr>
<td>FEEDBACK_BUFFER_SIZE</td>
<td>$Z^+$</td>
<td>GetIntegerv</td>
<td>0</td>
<td>Feedback buffer pointer</td>
<td>5.3</td>
<td>feedback</td>
</tr>
<tr>
<td>FEEDBACK_BUFFER_TYPE</td>
<td>$Z_5$</td>
<td>GetIntegerv</td>
<td>2D</td>
<td>Feedback type</td>
<td>5.3</td>
<td>feedback</td>
</tr>
<tr>
<td>–</td>
<td>$n \times Z_8$</td>
<td>GetError</td>
<td>0</td>
<td>Current error code(s)</td>
<td>2.5</td>
<td>–</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>
<td>–</td>
</tr>
<tr>
<td>–</td>
<td>$B$</td>
<td>–</td>
<td>FALSE</td>
<td>Occlusion query active</td>
<td>4.17</td>
<td>–</td>
</tr>
<tr>
<td>CURRENT_QUERY</td>
<td>$3 \times Z^+$</td>
<td>GetQueryiv</td>
<td>0</td>
<td>Active query object names</td>
<td>6.1.12</td>
<td>–</td>
</tr>
</tbody>
</table>
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
- The values of matrices other than the top-of-stack matrices
A.3. INVARIANCE RULES

- Scissor parameters (other than enable)
- Writemasks (color, index, depth, stencil)
- Clear values (color, index, depth, stencil, accumulation)
  - Current values (color, index, normal, texture coords, edgeflag)
  - Current raster color, index and texture coordinates.
  - Material properties (ambient, diffuse, specular, emission, shininess)

Strongly suggested:

- Matrix mode
- Matrix stack depths
- Alpha test parameters (other than enable)
- Stencil parameters (other than enable)
- Depth test parameters (other than enable)
- Blend parameters (other than enable)
- Logical operation parameters (other than enable)
- Pixel storage and transfer state
- Evaluator state (except as it affects the vertex data generated by the evaluators)
- Polygon offset parameters (other than enables, and except as they affect the depth values of fragments)

**Corollary 1** Fragment generation is invariant with respect to the state values marked with • in Rule 2.

**Corollary 2** The window coordinates (x, y, and z) of generated fragments are also invariant with respect to

**Required:**

- Current values (color, color index, normal, texture coords, edgeflag)
- Current raster color, color index, and texture coordinates
- Material properties (ambient, diffuse, specular, emission, shininess)

**Rule 3** The arithmetic of each per-fragment operation is invariant except with respect to parameters that directly control it (the parameters that control the alpha test, for instance, are the alpha test enable, the alpha test function, and the alpha test reference value).
A.4. WHAT ALL THIS MEANS

Corollary 3 Images rendered into different color buffers sharing the same framebuffer, either simultaneously or separately using the same command sequence, are pixel identical.

Rule 4 The same vertex or fragment shader will produce the same result when run multiple times with the same input. The wording 'the same shader' means a program object that is populated with the same source strings, which are compiled and then linked, possibly multiple times, and which program object is then executed using the same GL state vector.

Rule 5 All fragment shaders that either conditionally or unconditionally assign \texttt{gl\_FragCoord.z} to \texttt{gl\_FragDepth} are depth-invariant with respect to each other, for those fragments where the assignment to \texttt{gl\_FragDepth} actually is done.

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 CURRENT_RASTER_TEXTURE_COORDS must be maintained correctly at all times, including periods while texture mapping is not enabled, and when the GL is in color index mode.

2. When requested, texture coordinates returned in feedback mode are always valid, including periods while texture mapping is not enabled, and when the GL is in color index mode.

3. The error semantics of upward compatible OpenGL revisions may change. Otherwise, only additions can be made to upward compatible revisions.

4. GL query commands are not required to satisfy the semantics of the Flush or the Finish commands. All that is required is that the queried state be consistent with complete execution of all previously executed GL commands.

5. 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.


7. The mask specified as the third argument to StencilFune 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.
8. Polygon shading is completed before the polygon mode is interpreted. If the shade model is FLAT, all of the points or lines generated by a single polygon will have the same color.

9. A display list is just a group of commands and arguments, so errors generated by commands in a display list must be generated when the list is executed. If the list is created in COMPILE mode, errors should not be generated while the list is being created.

10. RasterPos does not change the current raster index from its default value in an RGBA mode GL context. Likewise, RasterPos does not change the current raster color from its default value in a color index GL context. Both the current raster index and the current raster color can be queried, however, regardless of the color mode of the GL context.

11. A material property that is attached to the current color via ColorMaterial always takes the value of the current color. Attempts to change that material property via Material calls have no effect.

12. Material and ColorMaterial can be used to modify the RGBA material properties, even in a color index context. Likewise, Material can be used to modify the color index material properties, even in an RGBA context.

13. There is no atomicity requirement for OpenGL rendering commands, even at the fragment level.

14. Because rasterization of non-antialiased polygons is point sampled, polygons that have no area generate no fragments when they are rasterized in FILL mode, and the fragments generated by the rasterization of “narrow” polygons may not form a continuous array.

15. OpenGL does not force left- or right-handedness on any of its coordinates systems. Consider, however, the following conditions: (1) the object coordinate system is right-handed; (2) the only commands used to manipulate the model-view matrix are Scale (with positive scaling values only), Rotate, and Translate; (3) exactly one of either Frustum or Ortho is used to set the projection matrix; (4) the near value is less than the far value for DepthRange. If these conditions are all satisfied, then the eye coordinate system is right-handed and the clip, normalized device, and window coordinate systems are left-handed.

16. ColorMaterial has no effect on color index lighting.
17. (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, and the coordinates of vertex A (resp. B) are identical to those of vertex C (resp. D), and the state of the the coordinate transformation is identical when A, B, C, and D are specified). 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.

18. OpenGL state continues to be modified in FEEDBACK mode and in SELECT mode. The contents of the framebuffer are not modified.

19. The current raster position, the user defined clip planes, the spot directions and the light positions for LIGHTi, and the eye planes for texgen are transformed when they are specified. They are not transformed during a PopAttrib, or when copying a context.

20. 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.

21. For any GL and framebuffer state, and for any group of GL commands and arguments, the resulting GL and framebuffer state is identical whether the GL commands and arguments are executed normally or from a display list.
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 less than four, the data corresponding to texels outside the image are irrelevant and undefined.

When an RGTC image with a width of $w$, height of $h$, and block size of block-size (8 or 16 bytes) is decoded, the corresponding image size (in bytes) is:

$$\lceil \frac{w}{4} \rceil \times \lceil \frac{h}{4} \rceil \times \text{blocksize}.$$  

When decoding an RGTC image, the block containing the texel at offset $(x, y)$ begins at an offset (in bytes) relative to the base of the image of:

$$\text{blocksize} \times \left( \lceil \frac{w}{4} \rceil \times \lceil \frac{y}{4} \rceil + \lceil \frac{x}{4} \rceil \right).$$

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. RGTC COMPRESSED TEXTURE IMAGE FORMATS

C.1.1 Format COMPRESSED_RED_RGTC1

Each $4 \times 4$ 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 $bits_x$ 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 $RED_1$ as though they were pixels with a format of LUMINANCE and a type of UNSIGNED_BYTE.
$bits$ 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:
C.1. RGTC COMPRESSED TEXTURE IMAGE FORMATS

\[ R = \begin{cases}
  \text{RED}_0, & \text{red}_0 > \text{red}_1, \text{code}(x, y) = 0 \\
  \text{RED}_1, & \text{red}_0 > \text{red}_1, \text{code}(x, y) = 1 \\
  \frac{6\text{RED}_0 + \text{RED}_1}{7}, & \text{red}_0 > \text{red}_1, \text{code}(x, y) = 2 \\
  \frac{5\text{RED}_0 + 2\text{RED}_1}{7}, & \text{red}_0 > \text{red}_1, \text{code}(x, y) = 3 \\
  \frac{4\text{RED}_0 + 3\text{RED}_1}{7}, & \text{red}_0 > \text{red}_1, \text{code}(x, y) = 4 \\
  \frac{3\text{RED}_0 + 4\text{RED}_1}{7}, & \text{red}_0 > \text{red}_1, \text{code}(x, y) = 5 \\
  \frac{2\text{RED}_0 + 5\text{RED}_1}{7}, & \text{red}_0 > \text{red}_1, \text{code}(x, y) = 6 \\
  \frac{\text{RED}_0 + 6\text{RED}_1}{7}, & \text{red}_0 > \text{red}_1, \text{code}(x, y) = 7 \\
  \text{RED}_0, & \text{red}_0 \leq \text{red}_1, \text{code}(x, y) = 0 \\
  \text{RED}_1, & \text{red}_0 \leq \text{red}_1, \text{code}(x, y) = 1 \\
  \frac{4\text{RED}_0 + \text{RED}_1}{5}, & \text{red}_0 \leq \text{red}_1, \text{code}(x, y) = 2 \\
  \frac{3\text{RED}_0 + 2\text{RED}_1}{5}, & \text{red}_0 \leq \text{red}_1, \text{code}(x, y) = 3 \\
  \frac{2\text{RED}_0 + 3\text{RED}_1}{5}, & \text{red}_0 \leq \text{red}_1, \text{code}(x, y) = 4 \\
  \frac{\text{RED}_0 + 4\text{RED}_1}{5}, & \text{red}_0 \leq \text{red}_1, \text{code}(x, y) = 5 \\
  \text{RED}_{\text{min}}, & \text{red}_0 \leq \text{red}_1, \text{code}(x, y) = 6 \\
  \text{RED}_{\text{max}}, & \text{red}_0 \leq \text{red}_1, \text{code}(x, y) = 7
\end{cases} \]

\( \text{RED}_{\text{min}} \) and \( \text{RED}_{\text{max}} \) 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 COMPRESSED SIGNED RED RGTC1

Each 4 × 4 block of texels consists of 64 bits of signed red image data. The red
values of a texel are extracted in the same way as COMPRESSED RED RGTC1 except
\( \text{red}_0, \text{red}_1, \text{RED}_0, \text{RED}_1, \text{RED}_{\text{min}}, \) and \( \text{RED}_{\text{max}} \) are signed values defined as
follows:

\( \text{red}_0 \) and \( \text{red}_1 \) are 8-bit signed (two’s complement) integers.

\[
\text{RED}_0 = \begin{cases}
  \frac{\text{red}_0}{127.0}, & \text{red}_0 > -128 \\
  -1.0, & \text{red}_0 = -128
\end{cases}
\]

\[
\text{RED}_1 = \begin{cases}
  \frac{\text{red}_1}{127.0}, & \text{red}_1 > -128 \\
  -1.0, & \text{red}_1 = -128
\end{cases}
\]

\( \text{RED}_{\text{min}} = -1.0 \)
C.1. RGTC COMPRESSED TEXTURE IMAGE FORMATS

\[ \text{RED}_{\text{max}} = 1.0 \]

CAVEAT for signed red_0 and red_1 values: the expressions red_0 > red_1 and red_0 \leq red_1 above are considered undefined (read: may vary by implementation) when red_0 = -127 and red_1 = -128. This is because if red_0 were remapped to -127 prior to the comparison to reduce the latency of a hardware decompressor, the expressions would reverse their logic. Encoders for the signed red-green formats should avoid encoding blocks where red_0 = -127 and red_1 = -128.

C.1.3 Format COMPRESSED_RG_RGTC2

Each 4 \times 4 block of texels consists of 64 bits of compressed unsigned red image data followed by 64 bits of compressed unsigned green image data.

The first 64 bits of compressed red are decoded exactly like COMPRESSED_RED_RGTC1 above.

The second 64 bits of compressed green are decoded exactly like COMPRESSED_RED_RGTC1 above except the decoded value \( R \) for this second block is considered the resulting green value \( G \).

Since the decoded texel has a red-green format, the resulting RGBA value for the texel is \( (R, G, 0, 1) \).

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

State that can be shared between contexts includes display lists, pixel and vertex buffer objects, program and shader objects, and texture objects (except for the texture objects named zero).

Framebuffer and vertex array objects are not shared.

D.1 Object Deletion Behavior

After an object is deleted, its name is immediately marked unused. Caution should be taken when deleting a shared object bound in multiple contexts. Following its deletion, the object’s name can be used by any context to create a new object, or be returned by Gen* commands, even though the underlying object state and data may still be in use by contexts other than the one in which the object was deleted. Such another context may continue using the object, and may still contain state identifying its name as being currently bound, until such time as another attempt to bind the name is made in that context. Since the name is marked unused, rather than rebinding the underlying object, a new object with the same name will be created.
Appendix E

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.1.

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 allow defining subsets of OpenGL functionality targeted to specific application domains. While OpenGL 3.0 only defines a single profile, future versions may introduce profiles addressing domains such as workstation, gaming, and embedded. Implementations are not required to support all defined profiles, but must support at least one profile.

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 Profiles and Deprecated Features of OpenGL 3.0

OpenGL 3.0 defines a single profile, and all OpenGL 3.0 implementations must support that profile.
E.1. PROFILES AND DEPRECATED FEATURES OF OPENGL 3.0

The features deprecated in OpenGL 3.0 are summarized below, together with the sections of the specification in which they are defined. Functions which are completely deprecated will generate an `INVALID_OPERATION` error if called in a forward-compatible context. Functions which are partially deprecated (e.g. no longer accept some parameter values) will generate the errors appropriate for any other unrecognized value of that parameter when a deprecated value is passed in a forward-compatible context.

- 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.

- 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*` (section 2.6.1); `Color*`, `FogCoord*`, `Index*`, `Normal3*`, `SecondaryColor3*`, `TexCoord*`, `Vertex*` (section 2.7); and all associated state in tables 6.4 and 6.5. 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.

- Edge flags and fixed-function vertex processing - `ColorPointer`, `EdgeFlagPointer`, `FogCoordPointer`, `IndexPointer`, `NormalPointer`, `SecondaryColorPointer`, `TexCoordPointer`, `VertexPointer`, `EnableClientState`, `DisableClientState`, and `ClientActiveTexture` (section 2.8); `Frustum`, `LoadIdentity`, `LoadMatrix`, `LoadTransposeMatrix`, `MatrixMode`, `MultMatrix`, `MultTransposeMatrix`, `Ortho`, `PopMatrix`, `PushMatrix`, `Rotate`, `Scale`, and `Translate` (section 2.12.2); `Enable/Disable` targets `RESCALE_NORMAL` and `NORMALIZE` (section 2.12.3); `TexGen*` and `Enable/Disable` targets `TEXTURE_GEN_*` (section 2.12.4, `Material*`
Light*, LightModel*, and ColorMaterial, and Enable/Disable targets LIGHTING, VERTEX_PROGRAM_TWO_SIDE, LIGHTi, and COLOR_MATERIAL (sections 2.19.2 and 2.19.3; and all associated fixed-function vertex array, multitexture, matrix and matrix stack, normal and texture coordinate, and lighting 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, ClampColor, and ShadeModel commands in section 2.19 are not deprecated, as they still affect other non-deprecated functionality;

• Client vertex arrays - all vertex array attribute pointers must refer to buffer objects (section 2.9.1). 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* (section 2.11).

• Current raster position - RasterPos* and WindowPos* (section 2.18), and all associated state.

• Two-sided color selection - to be done

• Non-sprite points (section 3.4) - 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 (section 3.5.2, and all associated state.

• Quadrilateral and polygon primitives - vertex array drawing modes POLYGON, QUADS, and QUAD_STRIP (section 2.6.1, related descriptions of rasterization of non-triangle polygons in section 3.6, and all associated state. Bug 3562

• 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_STENCIL (section 3.6.2, 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 (sections 3.7.2, 3.7.3, and 3.7.5), and all associated state and commands defining that state.

• Pixel drawing - DrawPixels and PixelZoom (section 3.7.4). However, the language describing pixel rectangles in section 3.7 is retained as it is required for TexImage* and ReadPixels.

• Bitmaps - Bitmap (section 3.8).

• 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.

• Legacy pixel formats - all ALPHA LUMINANCE, LUMINANCE_ALPHA, and INTENSITY external and internal formats, including compressed, floating-point, and integer variants (see tables 3.6, 3.15, 3.17, 3.19, 3.23, and 6.1); all references to luminance and intensity formats elsewhere in the specification, including conversion to and from those formats; and all associated state, including state describing the allocation or format of luminance and intensity texture or framebuffer components.

• Depth texture mode - DEPTH_TEXTURE_MODE. Section 3.9.14 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 3.9.1); all language in section 3.9 referring to nonzero border widths during texture image specification and texture sampling; TexParameter* target TEXTURE_BORDER_COLOR and texture wrap mode CLAMP_TO_BORDER (section 3.9.4); and all associated state.

• Automatic mipmap generation - TexParameter* target GENERATE_MIPMAP (section 3.9.7), and all associated state
E.1. Profiles and Deprecated Features of OpenGL 3.0

- Fixed-function fragment processing - `AreTexturesResident`, `PrioritizeTextures`, and `TexParameter` target `TEXTURE_PRIORITY`; `TexEnv` target `TEXTURE_ENV` and all associated parameters; `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; and all associated state.

- Alpha test - `AlphaFunc` and `Enable/Disable` target `ALPHA_TEST` (section 4.1.4), 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` (section 4.2.4); the `ACCUM_*_BITS` framebuffer state describing the size of accumulation buffer components (table 6.51); 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.

- 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 in table 5.1 (section 5.1, and all associated state.

- Selection and feedback modes - `RenderMode`, `InitNames`, `PopName`, `PushName`, `LoadName`, and `SelectBuffer` (section 5.2); `FeedbackBuffer` and `PassThrough` (section 5.3); and all associated state.

- Display lists - `NewList`, `EndList`, `CallList`, `CallLists`, `ListBase`, `GenLists`, `IsList`, and `DeleteLists` (section 5.4); all references to display lists and behavior when compiling commands into display lists elsewhere in the specification; and all associated state.

- Hints - the `PERSPECTIVE_CORRECTION_HINT`, `FOG_HINT`, and `GENERATE_MIPMAP_HINT` targets to `Hint` (section 5.6).

- Attribute stacks - `PushAttrib`, `PushClientAttrib`, `PopAttrib`, `PopClientAttrib`, `MAX_ATTRIB_STACK_DEPTH`, `MAX_CLIENT_ATTRIB_STACK_DEPTH` state, and the values `ALL_ATTRIB_BITS` and `CLIENT_ALL_ATTRIB_BITS` (section 6.1.18).
• Unified extension string - EXTENSIONS target to **GetString** (section 6.1.11).

• 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 1.1

OpenGL version 1.1 is the first revision since the original version 1.0 was released on 1 July 1992. Version 1.1 is upward compatible with version 1.0, meaning that any program that runs with a 1.0 GL implementation will also run unchanged with a 1.1 GL implementation. Several additions were made to the GL, especially to the texture mapping capabilities, but also to the geometry and fragment operations. Following are brief descriptions of each addition.

F.1 Vertex Array

Arrays of vertex data may be transferred to the GL with many fewer commands than were previously necessary. Six arrays are defined, one each storing vertex positions, normal coordinates, colors, color indices, texture coordinates, and edge flags. The arrays may be specified and enabled independently, or one of the pre-defined configurations may be selected with a single command.

The primary goal was to decrease the number of subroutine calls required to transfer non-display listed geometry data to the GL. A secondary goal was to improve the efficiency of the transfer; especially to allow direct memory access (DMA) hardware to be used to effect the transfer. The additions match those of the GL_EXT_vertex_array extension, except that static array data are not supported (because they complicated the interface, and were not being used), and the pre-defined configurations are added (both to reduce subroutine count even further, and to allow for efficient transfer of array data).
F.2 Polygon Offset

Depth values of fragments generated by the rasterization of a polygon may be shifted toward or away from the origin, as an affine function of the window coordinate depth slope of the polygon. Shifted depth values allow coplanar geometry, especially facet outlines, to be rendered without depth buffer artifacts. They may also be used by future shadow generation algorithms.

The additions match those of the GL_EXT_polygon_offset extension, with two exceptions. First, the offset is enabled separately for POINT, LINE, and FILL rasterization modes, all sharing a single affine function definition. (Shifting the depth values of the outline fragments, instead of the fill fragments, allows the contents of the depth buffer to be maintained correctly.) Second, the offset bias is specified in units of depth buffer resolution, rather than in the [0,1] depth range.

F.3 Logical Operation

Fragments generated by RGBA rendering may be merged into the framebuffer using a logical operation, just as color index fragments are in GL version 1.0. Blending is disabled during such operation because it is rarely desired, because many systems could not support it, and to match the semantics of the GL_EXT_blend_logic_op extension, on which this addition is loosely based.

F.4 Texture Image Formats

Stored texture arrays have a format, known as the internal format, rather than a simple count of components. The internal format is represented as a single enumerated value, indicating both the organization of the image data (LUMINANCE, RGB, etc.) and the number of bits of storage for each image component. Clients can use the internal format specification to suggest the desired storage precision of texture images. New base internal formats, ALPHA and INTENSITY, provide new texture environment operations. These additions match those of a subset of the GL_EXT_texture extension.

F.5 Texture Replace Environment

A common use of texture mapping is to replace the color values of generated fragments with texture color data. This could be specified only indirectly in GL version 1.0, which required that client specified “white” geometry be modulated
by a texture. GL version 1.1 allows such replacement to be specified explicitly, possibly improving performance. These additions match those of a subset of the GL\_EXT\_texture extension.

F.6 Texture Proxies

Texture proxies allow a GL implementation to advertise different maximum texture image sizes as a function of some other texture parameters, especially of the internal image format. Clients may use the proxy query mechanism to tailor their use of texture resources at run time. The proxy interface is designed to allow such queries without adding new routines to the GL interface. These additions match those of a subset of the GL\_EXT\_texture extension, except that implementations return allocation information consistent with support for complete mipmap arrays.

F.7 Copy Texture and Subtexture

Texture array data can be specified from framebuffer memory, as well as from client memory, and rectangular subregions of texture arrays can be redefined either from client or framebuffer memory. These additions match those defined by the GL\_EXT\_copy\_texture and GL\_EXT\_subtexture extensions.

F.8 Texture Objects

A set of texture arrays and their related texture state can be treated as a single object. Such treatment allows for greater implementation efficiency when multiple arrays are used. In conjunction with the subtexture capability, it also allows clients to make gradual changes to existing texture arrays, rather than completely redefining them. These additions match those of the GL\_EXT\_texture\_object extension, with slight additions to the texture residency semantics.

F.9 Other Changes

1. Color indices may now be specified as unsigned bytes.

2. Texture coordinates \(s, t,\) and \(r\) are divided by \(q\) during the rasterization of points, pixel rectangles, and bitmaps. This division was documented only for lines and polygons in the 1.0 version.
3. The line rasterization algorithm was changed so that vertical lines on pixel borders rasterize correctly.

4. Separate pixel transfer discussions in chapter 3 and chapter 4 were combined into a single discussion in chapter 3.

5. Texture alpha values are returned as 1.0 if there is no alpha channel in the texture array. This behavior was unspecified in the 1.0 version, and was incorrectly documented in the reference manual.

6. Fog start and end values may now be negative.

7. Evaluated color values direct the evaluation of the lighting equation if ColorMaterial is enabled.

F.10 Acknowledgements

OpenGL 1.1 is the result of the contributions of many people, representing a cross section of the computer industry. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:

Kurt Akeley, Silicon Graphics
Bill Armstrong, Evans & Sutherland
Andy Bigos, 3Dlabs
Pat Brown, IBM
Jim Cobb, Evans & Sutherland
Dick Coulter, Digital Equipment
Bruce D’Amora, GE Medical Systems
John Dennis, Digital Equipment
Fred Fisher, Accel Graphics
Chris Frazier, Silicon Graphics
Todd Frazier, Evans & Sutherland
Tim Freese, NCD
Ken Garnett, NCD
Mike Heck, Template Graphics Software
Dave Higgins, IBM
Phil Huxley, 3Dlabs
Dale Kirkland, Intergraph
Hock San Lee, Microsoft
Kevin LeFebvre, Hewlett Packard
Jim Miller, IBM
Tim Misner, SunSoft
F.10. ACKNOWLEDGEMENTS

Jeremy Morris, 3Dlabs
Israel Pinkas, Intel
Bimal Poddar, IBM
Lyle Ramshaw, Digital Equipment
Randi Rost, Hewlett Packard
John Schimpf, Silicon Graphics
Mark Segal, Silicon Graphics
Igor Sinyak, Intel
Jeff Stevenson, Hewlett Packard
Bill Sweeney, SunSoft
Kelvin Thompson, Portable Graphics
Neil Trevett, 3Dlabs
Linas Vepstas, IBM
Andy Vesper, Digital Equipment
Henri Warren, Megatek
Paula Womack, Silicon Graphics
Mason Woo, Silicon Graphics
Steve Wright, Microsoft

Version 3.0 (with changes marked - August 11, 2008)
Appendix G

Version 1.2

OpenGL version 1.2, released on March 16, 1998, is the second revision since the original version 1.0. Version 1.2 is upward compatible with version 1.1, meaning that any program that runs with a 1.1 GL implementation will also run unchanged with a 1.2 GL implementation.

Several additions were made to the GL, especially to texture mapping capabilities and the pixel processing pipeline. Following are brief descriptions of each addition.

G.1 Three-Dimensional Texturing

Three-dimensional textures can be defined and used. In-memory formats for three-dimensional images, and pixel storage modes to support them, are also defined. The additions match those of the GL\_EXT\_texture3D extension.

One important application of three-dimensional textures is rendering volumes of image data.

G.2 BGRA Pixel Formats

BGRA extends the list of client memory color formats. Specifically, it provides a component order matching file and framebuffer formats common on Windows platforms. The additions match those of the GL\_EXT\_bgra extension.
G.3  Packed Pixel Formats

Packed pixels in client memory are represented entirely by one unsigned byte, one
unsigned short, or one unsigned integer. The fields with the packed pixel are not
proper machine types, but the pixel as a whole is. Thus the pixel storage modes
and their unpacking counterparts all work correctly with packed pixels.

The additions match those of the GL\_EXT\_packed\_pixels extension, with the
further addition of reversed component order packed formats.

G.4  Normal Rescaling

Normals may be rescaled by a constant factor derived from the model-view matrix.
Rescaling can operate faster than renormalization in many cases, while resulting in
the same unit normals.

The additions are based on the GL\_EXT\_rescale\_normal extension.

G.5  Separate Specular Color

Lighting calculations are modified to produce a primary color consisting of emis-
sive, ambient and diffuse terms of the usual GL lighting equation, and a secondary
color consisting of the specular term. Only the primary color is modified by the
texture environment; the secondary color is added to the result of texturing to pro-
duce a single post-texturing color. This allows highlights whose color is based on
the light source creating them, rather than surface properties.

The additions match those of the GL\_EXT\_separate\_specular\_color extension.

G.6  Texture Coordinate Edge Clamping

GL normally clamps such that the texture coordinates are limited to exactly the
range \([0, 1]\). When a texture coordinate is clamped using this algorithm, the texture
sampling filter straddles the edge of the texture image, taking half its sample values
from within the texture image, and the other half from the texture border. It is
sometimes desirable to clamp a texture without requiring a border, and without
using the constant border color.

A new texture clamping algorithm, CLAMP\_TO\_EDGE, clamps texture coordi-
nates at all mipmap levels such that the texture filter never samples a border texel.
The color returned when clamping is derived only from texels at the edge of the
texture image.
The additions match those of the GL_SGIS_texture_edge_clamp extension.

G.7 Texture Level of Detail Control

Two constraints related to the texture level of detail parameter $\lambda$ are added. One constraint clamps $\lambda$ to a specified floating point range. The other limits the selection of mipmap image arrays to a subset of the arrays that would otherwise be considered.

Together these constraints allow a large texture to be loaded and used initially at low resolution, and to have its resolution raised gradually as more resolution is desired or available. Image array specification is necessarily integral, rather than continuous. By providing separate, continuous clamping of the $\lambda$ parameter, it is possible to avoid "popping" artifacts when higher resolution images are provided.

The additions match those of the GL_SGIS_texture_lod extension.

G.8 Vertex Array Draw Element Range

A new form of DrawElements that provides explicit information on the range of vertices referred to by the index set is added. Implementations can take advantage of this additional information to process vertex data without having to scan the index data to determine which vertices are referenced.

The additions match those of the GL_EXT_draw_range_elements extension.

G.9 Imaging Subset

The remaining new features are primarily intended for advanced image processing applications, and may not be present in all GL implementations. They are collectively referred to as the imaging subset.

G.9.1 Color Tables

A new RGBA-format color lookup mechanism is defined in the pixel transfer process, providing additional lookup capabilities beyond the existing lookup. The key difference is that the new lookup tables are treated as one-dimensional images with internal formats, like texture images and convolution filter images. Thus the new tables can operate on a subset of the components of passing pixel groups. For example, a table with internal format ALPHA modifies only the A component of each pixel group, leaving the R, G, and B components unmodified.
Three independent lookups may be performed: prior to convolution; after convolution and prior to color matrix transformation; after color matrix transformation and prior to gathering pipeline statistics.

Methods to initialize the color lookup tables from the framebuffer, in addition to the standard memory source mechanisms, are provided.

Portions of a color lookup table may be redefined without reinitializing the entire table. The affected portions may be specified either from client memory or from the framebuffer.

The additions match those of the GL\textunderscore EXT\textunderscore color\textunderscore table and GL\textunderscore EXT\textunderscore color\textunderscore subtable extensions.

G.9.2 Convolution

One- or two-dimensional convolution operations are executed following the first color table lookup in the pixel transfer process. The convolution kernels are themselves treated as one- and two-dimensional images, which can be loaded from application memory or from the framebuffer.

The convolution framework is designed to accommodate three-dimensional convolution, but that API is left for a future extension.

The additions match those of the GL\textunderscore EXT\textunderscore convolution and GL\textunderscore HP\textunderscore convolution\textunderscore border\textunderscore modes extensions.

G.9.3 Color Matrix

A 4x4 matrix transformation and associated matrix stack are added to the pixel transfer path. The matrix operates on RGBA pixel groups, using the equation

\[ C' = MC, \]

where

\[ C = \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix} \]

and \( M \) is the 4 \times 4 matrix on the top of the color matrix stack. After the matrix multiplication, each resulting color component is scaled and biased by a programmed amount. Color matrix multiplication follows convolution.

The color matrix can be used to reassign and duplicate color components. It can also be used to implement simple color space conversions.

The additions match those of the GL\textunderscore SGI\textunderscore color\textunderscore matrix extension.
G.9.4 Pixel Pipeline Statistics

Pixel operations that count occurrences of specific color component values (histogram) and that track the minimum and maximum color component values (minmax) are performed at the end of the pixel transfer pipeline. An optional mode allows pixel data to be discarded after the histogram and/or minmax operations are completed. Otherwise the pixel data continues on to the next operation unaffected.

The additions match those of the GL_EXT_histogram extension.

G.9.5 Constant Blend Color

A constant color that can be used to define blend weighting factors may be defined. A typical usage is blending two RGB images. Without the constant blend factor, one image must have an alpha channel with each pixel set to the desired blend factor.

The additions match those of the GL_EXT_blend_color extension.

G.9.6 New Blending Equations

Blending equations other than the normal weighted sum of source and destination components may be used.

Two of the new equations produce the minimum (or maximum) color components of the source and destination colors. Taking the maximum is useful for applications such as maximum projection in medical imaging.

The other two equations are similar to the default blending equation, but produce the difference of its left and right hand sides, rather than the sum. Image differences are useful in many image processing applications.

The additions match those of the GL_EXT_blend_minmax and GL_EXT_blend_subtract extensions.

G.10 Acknowledgements

OpenGL 1.2 is the result of the contributions of many people, representing a cross section of the computer industry. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:

- Kurt Akeley, Silicon Graphics
- Bill Armstrong, Evans & Sutherland
- Otto Berkes, Microsoft
- Pierre-Luc Bisaillon, Matrox Graphics
- Drew Bliss, Microsoft

Version 3.0 (with changes marked - August 11, 2008)
G.10. ACKNOWLEDGEMENTS

David Blythe, Silicon Graphics
Jon Brewster, Hewlett Packard
Dan Brokenshire, IBM
Pat Brown, IBM
Newton Cheung, S3
Bill Clifford, Digital
Jim Cobb, Parametric Technology
Bruce D’Amora, IBM
Kevin Dallas, Microsoft
Mahesh Dandapani, Rendition
Daniel Daum, AccelGraphics
Suzy Deffeyes, IBM
Peter Doyle, Intel
Jay Duluk, Raycer
Craig Dunwoody, Silicon Graphics
Dave Erb, IBM
Fred Fisher, AccelGraphics / Dynamic Pictures
Celeste Fowler, Silicon Graphics
Allen Gallotta, ATI
Ken Garnett, NCD
Michael Gold, Nvidia / Silicon Graphics
Craig Groeschel, Metro Link
Jan Hardenbergh, Mitsubishi Electric
Mike Heck, Template Graphics Software
Dick Hessel, Raycer Graphics
Paul Ho, Silicon Graphics
Shawn Hopwood, Silicon Graphics
Jim Hurley, Intel
Phil Huxley, 3Dlabs
Dick Jay, Template Graphics Software
Paul Jensen, 3Dfx
Brett Johnson, Hewlett Packard
Michael Jones, Silicon Graphics
Tim Kelley, Real3D
Jon Khazam, Intel
Louis Khouw, Sun
Dale Kirkland, Intergraph
Chris Kitrick, Raycer
Don Kuo, S3
Herb Kuta, Quantum 3D
G.10. ACKNOWLEDGEMENTS

Phil Lacroute, Silicon Graphics
Prakash Ladia, S3
Jon Leech, Silicon Graphics
Kevin Lefebvre, Hewlett Packard
David Ligon, Raycer Graphics
Kent Lin, S3
Dan McCabe, S3
Jack Middleton, Sun
Tim Misner, Intel
Bill Mitchell, National Institute of Standards
Jeremy Morris, 3Dlabs
Gene Munce, Intel
William Newhall, Real3D
Matthew Papakipos, Nvidia / Raycer
Garry Paxinos, Metro Link
Hanspeter Pfister, Mitsubishi Electric
Richard Pimentel, Parametric Technology
Bimal Poddar, IBM / Intel
Rob Putney, IBM
Mike Quinlan, Real3D
Nate Robins, University of Utah
Detlef Roettger, Elsa
Randi Rost, Hewlett Packard
Kevin Rushforth, Sun
Richard S. Wright, Real3D
Hock San Lee, Microsoft
John Schimpf, Silicon Graphics
Stefan Seeboth, ELSA
Mark Segal, Silicon Graphics
Bob Seitsinger, S3
Min-Zhi Shao, S3
Colin Sharp, Rendition
Igor Sinyak, Intel
Bill Sweeney, Sun
William Sweeney, Sun
Nathan Tuck, Raycer
Doug Twillenger, Sun
John Tynefeld, 3dfx
Kartik Venkataraman, Intel
Andy Vesper, Digital Equipment

Version 3.0 (with changes marked - August 11, 2008)
Henri Warren, Digital Equipment / Megatek
Paula Womack, Silicon Graphics
Steve Wright, Microsoft
David Yu, Silicon Graphics
Randy Zhao, S3
Appendix H

Version 1.2.1

OpenGL version 1.2.1, released on October 14, 1998, introduced ARB extensions (see Appendix O). The only ARB extension defined in this version is multitexture, allowing application of multiple textures to a fragment in one rendering pass. Multitexture is based on the GL_SGIS_multitexture extension, simplified by removing the ability to route texture coordinate sets to arbitrary texture units.

A new corollary discussing display list and immediate mode invariance was added to Appendix B on April 1, 1999.
Appendix I

Version 1.3

OpenGL version 1.3, released on August 14, 2001, is the third revision since the original version 1.0. Version 1.3 is upward compatible with earlier versions, meaning that any program that runs with a 1.2, 1.1, or 1.0 GL implementation will also run unchanged with a 1.3 GL implementation.

Several additions were made to the GL, especially texture mapping capabilities previously defined by ARB extensions. Following are brief descriptions of each addition.

I.1 Compressed Textures

Compressing texture images can reduce texture memory utilization and improve performance when rendering textured primitives. The GL provides a framework upon which extensions providing specific compressed image formats can be built, and a set of generic compressed internal formats that allow applications to specify that texture images should be stored in compressed form without needing to code for specific compression formats (specific compressed formats, such as S3TC or FXT1, are supported by extensions).

Texture compression was promoted from the GL_ARB_texture_compression extension.

I.2 Cube Map Textures

Cube map textures provide a new texture generation scheme for looking up textures from a set of six two-dimensional images representing the faces of a cube. The \((str)\) texture coordinates are treated as a direction vector emanating from the center of a cube. At texture generation time, the interpolated per-fragment \((str)\) selects
one cube face two-dimensional image based on the largest magnitude coordinate (the major axis). A new \((st)\) is calculated by dividing the two other coordinates (the minor axes values) by the major axis value, and the new \((st)\) is used to lookup into the selected two-dimensional texture image face of the cube map.

Two new texture coordinate generation modes are provided for use in conjunction with cube map texturing. The \texttt{REFLECTION\_MAP} mode generates texture coordinates \((str)\) matching the vertex’s eye-space reflection vector, useful for environment mapping without the singularity inherent in \texttt{SPHERE\_MAP} mapping. The \texttt{NORMAL\_MAP} mode generates texture coordinates matching the vertex’s transformed eye-space normal, useful for texture-based diffuse lighting models.

Cube mapping was promoted from the \texttt{GL\_ARB\_texture\_cube\_map} extension.

### I.3 Multisample

Multisampling provides an antialiasing mechanism which samples 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 antialiasing appears to be automatic at the application level. Because each sample includes depth and stencil information, the depth and stencil functions perform equivalently to the single-sample mode.

When multisampling is supported, 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.

Multisampling is usually an expensive operation, so it is usually not supported on all contexts. Applications must obtain a multisample-capable context using the new interfaces provided by GLX 1.4 or by the \texttt{WGL\_ARB\_multisample} extension.

Multisampling was promoted from the \texttt{GL\_ARB\_multisample} extension; The definition of the extension was changed slightly to support both multisampling and supersampling implementations.

### I.4 Multitexture

Multitexture adds support for multiple texture units. The capabilities of the multiple texture units are identical, except that evaluation and feedback are supported only for texture unit 0. Each texture unit has its own state vector which includes texture vertex array specification, texture image and filtering parameters, and texture environment application.

The texture environments of the texture units are applied in a pipelined fashion whereby the output of one texture environment is used as the input fragment color.
for the next texture environment. Changes to texture client state and texture server state are each routed through one of two selectors which control which instance of texture state is affected.

Multitexture was promoted from the GL_ARB_multitexture extension.

I.5 Texture Add Environment Mode

The TEXTURE_ENV_MODE texture environment function ADD provides a texture function to add incoming fragment and texture source colors.

Texture add mode was promoted from the GL_ARB_texture_add extension.

I.6 Texture Combine Environment Mode

The TEXTURE_ENV_MODE texture environment function COMBINE provides a wide range of programmable combiner functions using the incoming fragment color, texture source color, texture constant color, and the result of the previous texture environment stage as possible parameters.

Combiner operations include passthrough, multiplication, addition and biased addition, subtraction, and linear interpolation of specified parameters. Different combiner operations may be selected for RGB and A components, and the final result may be scaled by 1, 2, or 4.

Texture combine was promoted from the GL_ARB_texture_env_combine extension.

I.7 Texture Dot3 Environment Mode

The TEXTURE_ENV_MODE COMBINE operations also provide three-component dot products of specified parameters, with the resulting scalar value replicated into the RGB or RGBA components of the output color. The dot product is performed using pseudo-signed arithmetic to enable per-pixel lighting computations.

Texture DOT3 mode was promoted from the GL_ARB_texture_env_dot3 extension.

I.8 Texture Border Clamp

The texture wrap parameter CLAMP_TO_BORDER mode clamps texture coordinates at all mipmap levels such that when the texture filter straddles an edge of the texture...
image, the color returned is derived only from border texels. This behavior mirrors
the behavior of the texture edge clamp mode introduced by OpenGL 1.2.

Texture border clamp was promoted from the
GL\_ARB\_texture\_border\_clamp extension.

I.9 Transpose Matrix

New functions and tokens are added allowing application matrices stored in row
major order rather than column major order to be transferred to the implementa-
tion. This allows an application to use standard C-language 2-dimensional arrays
and have the array indices match the expected matrix row and column indexes.
These arrays are referred to as transpose matrices since they are the transpose of
the standard matrices passed to OpenGL.

Transpose matrix adds an interface for transferring data to and from the OpenGL
pipeline. It does not change any OpenGL processing or imply any changes in state
representation.

Transpose matrix was promoted from the GL\_ARB\_transpose\_matrix exten-
sion.

I.10 Acknowledgements

OpenGL 1.3 is the result of the contributions of many people. Following is a partial
list of the contributors, including the company that they represented at the time of
their contribution:

Adrian Muntianu, ATI
Al Reyes, 3dfx
Alain Bouchard, Matrox
Alan Commike, SGI
Alan Heirich, Compaq
Alex Herrera, SP3D
Allen Akin, VA Linux
Allen Gallotta, ATI
Alligator Descartes, Arcane
Andy Vesper, MERL
Andy Wolf, Diamond Multimedia
Axel Schildan, S3
Barthold Lichtenbelt, 3Dlabs
Benj Lipchak, Compaq
Bill Armstrong, Evans & Sutherland
Bill Clifford, Intel
Bill Mannel, SGI
Bimal Poddar, Intel
Bob Beretta, Apple
Brent Insko, NVIDIA
Brian Goldiez, UCF
Brian Greenstone, Apple
Brian Paul, VA Linux
Brian Sharp, GLSetup
Bruce D’Amora, IBM
Bruce Stockwell, Compaq
Chris Brady, Alt.software
Chris Frazier, Raycer
Chris Hall, 3dlabs
Chris Hecker, GLSetup
Chris Lane, Intel
Chris Thornborrow, PixelFusion
Christopher Fraser, IMG
Chuck Smith, Intelligraphics
Craig Dunwoody, SGI
Dairsie Latimer, PixelFusion
Dale Kirkland, 3Dlabs / Intergraph
Dan Brokenshire, IBM
Dan Ginsburg, ATI
Dan McCabe, S3
Dave Aronson, Microsoft
Dave Gosselin, ATI
Dave Shreiner, SGI
Dave Zenz, Dell
David Aronson, Microsoft
David Blythe, SGI
David Kirk, NVIDIA
David Story, SGI
David Yu, SGI
Deanna Hohn, 3dfx
Dick Coulter, Silicon Magic
Don Mullis, 3dfx
Eamon O Dea, PixelFusion
Edward (Chip) Hill, Pixelfusion
Eiji Obata, NEC
I.10. ACKNOWLEDGEMENTS

Elio Del Giudice, Matrox
Eric Young, S3
Evan Hart, ATI
Fred Fisher, 3dLabs
Garry Paxinos, Metro Link
Gary Tarolli, 3dfx
George Kyriazis, NVIDIA
Graham Connor, IMG
Herb Kuta, Quantum3D
Howard Miller, Apple
Igor Sinyak, Intel
Jack Middleton, Sun
James Bowman, 3dfx
Jan C. Hardenbergh, MERL
Jason Mitchell, ATI
Jeff Weyman, ATI
Jeffrey Newquist, 3dfx
Jens Owen, Precision Insight
Jeremy Morris, 3Dlabs
Jim Bushnell, Pyramid Peak
John Dennis, Sharp Eye
John Metcalfe, IMG
John Stauffer, Apple
John Tynan, PixelFusion
John W. Polick, NEC
Jon Khazam, Intel
Jon Leech, SGI
Jon Paul Schelter, Matrox
Karl Hilleslad, NVIDIA
Kelvin Thompson
Ken Cameron, Pixelfusion
Ken Dyke, Apple
Ken Nicholson, SGI
Kent Lin, Intel
Kevin Lefebvre, HP
Kevin Martin, VA Linux
Kurt Akeley, SGI
Les Silvern, NEC
Mahesh Dandipani, Rendition
Mark Kilgard, NVIDIA

Version 3.0 (with changes marked - August 11, 2008)
Martin Amon, 3dfx
Martina Sourada, ATI
Matt Lavoie, Pixelfusion
Matt Russo, Matrox
Matthew Papakipos, NVIDIA
Michael Gold, NVIDIA
Miriam Geller, SGI
Morgan Von Essen, Metro Link
Naruki Aruga, PFU
Nathan Tuck, Raycer Graphics
Neil Trevett, 3Dlabs
Newton Cheung, S3
Nick Triantos, NVIDIA
Patrick Brown, Intel
Paul Jensen, 3dfx
Paul Keller, NVIDIA
Paul Martz, HP
Paula Womack, 3dfx
Peter Doenges, Evans & Sutherland
Peter Graffagnino, Apple
Phil Huxley, 3Dlabs
Ralf Biermann, Elsa AG
Randi Rost, 3Dlabs
Renee Rashid, Micron
Rich Johnson, HP
Richard Pimentel, PTC
Richard Schlein, Apple
Rick Hammerstone, ATI
Rik Faith, VA Linux
Rob Glidden, Sun
Rob Wheeler, 3dfx
Shari Petersen, Rendition
Shawn Hopwood, SGI
Steve Glickman, Silicon Magic
Steve McGuigan, SGI
Steve Wright, Microsoft
Stuart Anderson, Metro Link
T. C. Zhao, MERL
Teri Morrison, HP
Thomas Fox, IBM

Version 3.0 (with changes marked - August 11, 2008)
I.10. ACKNOWLEDGEMENTS

Tim Kelley, Real 3D
Tom Frisinger, ATI
Victor Vedovato, Micron
Vikram Simha, MERL
Yanjun Zhang, Sun
Zahid Hussain, TI
Appendix J

Version 1.4

OpenGL version 1.4, released on July 24, 2002, is the fourth revision since the original version 1.0. Version 1.4 is upward compatible with earlier versions, meaning that any program that runs with a 1.3, 1.2, 1.1, or 1.0 GL implementation will also run unchanged with a 1.4 GL implementation.

In addition to numerous additions to the classical fixed-function GL pipeline in OpenGL 1.4, the OpenGL ARB also approved the GL_ARB_vertex_program extension, which supports programmable vertex processing. Following are brief descriptions of each addition to OpenGL 1.4; see Chapter O for a description of GL_ARB_vertex_program.

J.1 Automatic Mipmap Generation

Setting the texture parameter \texttt{GENERATE\_MIPMAP} to \texttt{TRUE} introduces a side effect to any modification of the \texttt{level\_base} of a mipmap array, wherein all higher levels of the mipmap pyramid are recomputed automatically by successive filtering of the base level array.

Automatic mipmap generation was promoted from the \texttt{GL\_SGIS\_generate\_mipmap} extension.

J.2 Blend Squaring

Blend squaring extends the set of supported source and destination blend functions to permit squaring RGB and alpha values during blending. Functions \texttt{SRC\_COLOR} and \texttt{ONE\_MINUS\_SRC\_COLOR} are added to the allowed source blending functions,
and \texttt{DST\_COLOR} and \texttt{ONE\_MINUS\_DST\_COLOR} are added to the allowed destination blending functions.

Blend squaring was promoted from the \texttt{GL\_NV\_blend\_square} extension.

\section*{J.3 Changes to the Imaging Subset}

The subset of blending features described by \texttt{BlendEquation}, \texttt{BlendColor}, and the \texttt{BlendFunc} modes \texttt{CONSTANT\_COLOR}, \texttt{ONE\_MINUS\_CONSTANT\_COLOR}, \texttt{CONSTANT\_ALPHA}, and \texttt{ONE\_MINUS\_CONSTANT\_ALPHA} are now supported. These feature were available only in the optional imaging subset in versions 1.2 and 1.3 of the GL.

\section*{J.4 Depth Textures and Shadows}

Depth textures define a new texture internal format, \texttt{DEPTH}, normally used to represent depth values. Applications include image-based shadow casting, displacement mapping, and image-based rendering.

Image-based shadowing is enabled with a new texture application mode defined by the parameter \texttt{TEXTURE\_COMPARE\_MODE}. This mode enables comparing texture \( r \) coordinates to depth texture values to generate a boolean result.

Depth textures and shadows were promoted from the \texttt{GL\_ARB\_depth\_texture} and \texttt{GL\_ARB\_shadow} extensions.

\section*{J.5 Fog Coordinate}

A new associated vertex and fragment datum, the \textit{fog coordinate} may be used in computing fog for a fragment, instead of using eye distance to the fragment, by specifying the coordinate with the \texttt{FogCoord} commands and setting the \texttt{FOG\_COORDINATE\_SOURCE} fog parameter. Fog coordinates are particularly useful in computing more complex fog models.

Fog coordinate was promoted from the \texttt{GL\_EXT\_fog\_coord} extension.

\section*{J.6 Multiple Draw Arrays}

Multiple primitives may be drawn in a single call using the \texttt{MultiDrawArrays} and \texttt{MultiDrawElements} comments.

Multiple draw arrays was promoted from the \texttt{GL\_EXT\_multi\_draw\_arrays} extension.
J.7  Point Parameters

Point parameters defined by the PointParameter commands support additional geometric characteristics of points, allowing the size of a point to be affected by linear or quadratic distance attenuation, and increasing control of the mapping from point size to raster point area and point transparency. This effect may be used for distance attenuation in rendering particles or light points.

Point parameters was promoted from the GL_ARB_point_parameters extension.

J.8  Secondary Color

The secondary color may be varied even when lighting is disabled by specifying it as a vertex parameter with the SecondaryColor commands.

Secondary color was promoted from the GL_EXT_secondary_color extension.

J.9  Separate Blend Functions

Blending capability is extended with BlendFuncSeparate to allow independent setting of the RGB and alpha blend functions for blend operations that require source and destination blend factors.

Separate blend functions was promoted from the GL_EXT_blend_func_separate extension.

J.10  Stencil Wrap

New stencil operations INCR_WRAP and DECR_WRAP allow the stencil value to wrap around the range of stencil values instead of saturating to the minimum or maximum values on decrement or increment. Stencil wrapping is needed for algorithms that use the stencil buffer for per-fragment inside-outside primitive computations.

Stencil wrap was promoted from the GL_EXT_stencil_wrap extension.

J.11  Texture Crossbar Environment Mode

Texture crossbar extends the texture combine environment mode COMBINE by allowing use of the texture color from different texture units as sources to the texture combine function.
Texture environment crossbar was promoted from the GL_ARB_texture_env_crossbar extension.

J.12 Texture LOD Bias

The texture filter control parameter TEXTURE_LOD_BIAS may be set to bias the computed $\lambda$ parameter used in texturing for mipmap level of detail selection, providing a means to blur or sharpen textures. LOD bias may be used for depth of field and other special visual effects, as well as for some types of image processing.

Texture LOD bias was based on the GL_EXT_texture_lod_bias extension, with the addition of a second per-texture object bias term.

J.13 Texture Mirrored Repeat

Texture mirrored repeat extends the set of texture wrap modes with the mode MIRRORED_REPEAT. This effectively defines a texture map twice as large as the original texture image in which the additional half, for each mirrored texture coordinate, is a mirror image of the original texture. Mirrored repeat can be used seamless tiling of a surface.

Texture mirrored repeat was promoted from the GL_ARB_texture_mirrored_repeat extension.

J.14 Window Raster Position

The raster position may be set directly to specified window coordinates with the WindowPos commands, bypassing the transformation applied to RasterPos. Window raster position is particularly useful for imaging and other 2D operations.

Window raster position was promoted from the GL_ARB_window_pos extension.

J.15 Acknowledgements

OpenGL 1.4 is the result of the contributions of many people. Following is a partial list of the contributors, including the company that they represented at the time of their contribution. The editor especially thanks Bob Beretta and Pat Brown for their sustained efforts in leading the GL_ARB_vertex_program working group,
without which this critical extension could not have been defined and approved in conjunction with OpenGL 1.4.

Kurt Akeley, NVIDIA
Allen Akin
Bill Armstrong, Evans & Sutherland
Ben Ashbaugh, Intel
Chris Bentley, ATI
Bob Beretta, Apple
Daniel Brokenshire, IBM
Pat Brown, NVIDIA
Bill Clifford, Intel
Graham Connor, Videologic
Matt Craighead, NVIDIA
Suzy Deffeyes, IBM
Jean-Luc Dery, Discreet
Kenneth Dyke, Apple
Cass Everitt, NVIDIA
Allen Gallotta, ATI
Lee Gross, IBM
Evan Hart, ATI
Chris Hecker, Definition 6
Alan Heirich, Compaq / HP
Gareth Hughes, VA Linux
Michael I Gold, NVIDIA
Rich Johnson, HP
Mark Kilgard, NVIDIA
Dale Kirkland, 3Dlabs
David Kirk, NVIDIA
Christian Laforte, Alias—Wavefront
Luc Leblanc, Discreet
Jon Leech, SGI
Bill Licea-Kane, ATI
Barthold Lichtenbelt, 3Dlabs
Jack Middleton, Sun
Howard Miller, Apple
Jeremy Morris, 3Dlabs
Jon Paul Schelter, Matrox
Brian Paul, VA Linux / Tungsten Graphics
Bimal Poddar, Intel
Thomas Roell, Xi Graphics
J.15. ACKNOWLEDGEMENTS

Randi Rost, 3Dlabs
Jeremy Sandmel, ATI
John Stauffer, Apple
Nick Triantos, NVIDIA
Daniel Vogel, Epic Games
Mason Woo, World Wide Woo
Dave Zenz, Dell
Appendix K

Version 1.5

OpenGL version 1.5, released on July 29, 2003, is the fifth revision since the original version 1.0. Version 1.5 is upward compatible with earlier versions, meaning that any program that runs with a 1.4, 1.3, 1.2, 1.1, or 1.0 GL implementation will also run unchanged with a 1.5 GL implementation.

In addition to additions to the classical fixed-function GL pipeline in OpenGL 1.5, the OpenGL ARB also approved a related set of ARB extensions including the OpenGL Shading Language specification and the GL_ARB_shader_objects, GL_ARB_vertex_shader, and GL_ARB_fragment_shader extensions through which high-level shading language programs can be loaded and used in place of the fixed-function pipeline.

Following are brief descriptions of each addition to OpenGL 1.5. The low-level and high-level shading languages are important adjuncts to the OpenGL core. They are described in more detail in appendix O, and their corresponding ARB extension specifications are available online as described in that appendix.

K.1 Buffer Objects

Buffer objects allow various types of data (especially vertex array data) to be cached in high-performance graphics memory on the server, thereby increasing the rate of data transfers to the GL.

Buffer objects were promoted from the GL_ARB_vertex_buffer_object extension.
K.2 Occlusion Queries

An occlusion query is a mechanism whereby an application can query the number of pixels (or, more precisely, samples) drawn by a primitive or group of primitives. The primary purpose of occlusion queries is to determine the visibility of an object.

Occlusion query was promoted from the GL_ARB_occlusion_query extension.

K.3 Shadow Functions

Texture comparison functions are generalized to support all eight binary functions rather than just LEQUAL and GEQUAL.

Texture comparison functions were promoted from the GL_EXT_shadow_funcs extension.

K.4 Changed Tokens

To achieve consistency with the syntax guidelines for OpenGL function and token names, 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 K.1.

K.5 Acknowledgements

OpenGL 1.5 is the result of the contributions of many people. The editor especially thanks the following individuals for their sustained efforts in leading ARB working groups essential to the success of OpenGL 1.5 and of ARB extensions approved in conjunction with OpenGL 1.5:

Matt Craighead led the working group which created the GL_ARB_vertex_buffer_object extension and OpenGL 1.5 core feature. Kurt Akeley wrote the initial specification for the group.

Daniel Ginsburg and Matt Craighead led the working group which created the GL_ARB_occlusion_query extension and OpenGL 1.5 core feature.

Benjamin Lipchak led the fragment program working group which created the GL_ARB_fragment_program extension, completing the low-level programmable shading interface.

Bill Licea-Kane led the GL2 working group which created the high-level programmable shading interface, including the GL_ARB_fragment_shader,
### Table K.1: New token names and the old names they replace.

<table>
<thead>
<tr>
<th>New Token Name</th>
<th>Old Token Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOG_COORD_SRC</td>
<td>FOGCOORDINATE_SOURCE</td>
</tr>
<tr>
<td>FOG_COORD</td>
<td>FOGCOORDINATE</td>
</tr>
<tr>
<td>CURRENT_FOG_COORD</td>
<td>CURRENT_FOG_COORDINATE</td>
</tr>
<tr>
<td>FOGCOORD_ARRAY_TYPE</td>
<td>FOGCOORDINATE_ARRAY_TYPE</td>
</tr>
<tr>
<td>FOGCOORD_ARRAY_STRIDE</td>
<td>FOGCOORDINATE_ARRAY_STRIDE</td>
</tr>
<tr>
<td>FOGCOORD_ARRAY_POINTER</td>
<td>FOGCOORDINATE_ARRAY_POINTER</td>
</tr>
<tr>
<td>FOGCOORD_ARRAY</td>
<td>FOGCOORDINATEARRAY</td>
</tr>
<tr>
<td>FOGCOORD_ARRAY_BUFFER_BINDING</td>
<td>FOGCOORDINATE_ARRAY_BUFFER_BINDING</td>
</tr>
<tr>
<td>SRC0_RGB</td>
<td>SOURCE0_RGB</td>
</tr>
<tr>
<td>SRC1_RGB</td>
<td>SOURCE1_RGB</td>
</tr>
<tr>
<td>SRC2_RGB</td>
<td>SOURCE2_RGB</td>
</tr>
<tr>
<td>SRC0_ALPHA</td>
<td>SOURCE0_ALPHA</td>
</tr>
<tr>
<td>SRC1_ALPHA</td>
<td>SOURCE1_ALPHA</td>
</tr>
<tr>
<td>SRC2_ALPHA</td>
<td>SOURCE2_ALPHA</td>
</tr>
</tbody>
</table>

GL_ARB_shader_objects, and GL_ARB_vertex_shader extensions and the OpenGL Shading Language.

John Kessenich was the principal editor of the OpenGL Shading Language specification for the GL2 working group, starting from the initial glslang proposal written by John, Dave Baldwin, and Randi Rost.

A partial list of other contributors, including the company that they represented at the time of their contribution, follows:

- Kurt Akeley, NVIDIA
- Allen Akin
- Chad Anson, Dell Computer
- Bill Armstrong, Evans & Sutherland
- Ben Ashbaugh, Intel
- Dave Baldwin, 3Dlabs
- Chris Bentley, ATI
- Bob Beretta, Apple
- David Blythe
- Alain Bouchard, Matrox
- Daniel Brokenshire, IBM
- Pat Brown, NVIDIA
- John Carmack, Id Software
Paul Carmichael, NVIDIA
Bob Carwell, IBM
Paul Clarke, IBM
Bill Clifford, Intel
Roger Cloud, SGI
Graham Connor, Power VR
Matt Craighead, NVIDIA
Doug Crisman, SGI
Matt Cruikshank, Vital Images
Deron Dann Johnson, Sun
Suzy Deffeyes, IBM
Steve Demlow, Vital Images
Joe Deng, SiS
Jean-Luc Dery, Discreet
Kenneth Dyke, Apple
Brian Emberling, Sun
Cass Everitt, NVIDIA
Brandon Fliflet, Intel
Allen Gallotta, ATI
Daniel Ginsburg, ATI
Steve Glanville, NVIDIA
Peter Graffagnino, Apple
Lee Gross, IBM
Rick Hammerstone, ATI
Evan Hart, ATI
Chris Hecker, Definition 6
Alan Heirich, HP
Gareth Hughes, NVIDIA
Michael I Gold, NVIDIA
John Jarvis, Alt.software
Rich Johnson, HP
John Kessenich, 3Dlabs
Mark Kilgard, NVIDIA
Dale Kirkland, 3Dlabs
Raymond Klassen, Intel
Jason Knipe, Bioware
Jayant Kolhe, NVIDIA
Steve Koren, 3Dlabs
Bob Kuehne, SGI
Christian Laforte, Alias
K.5. ACKNOWLEDGEMENTS

Luc Leblanc, Discreet
Jon Leech, SGI
Kevin Lefebvre, HP
Bill Licea-Kane, ATI
Barthold Lichtenbelt, 3Dlabs
Kent Lin, Intel
Benjamin Lipchak, ATI
Rob Mace, ATI
Bill Mark, NVIDIA
Michael McCool, U. Waterloo
Jack Middleton, Sun
Howard Miller, Apple
Teri Morrison, HP / 3Dlabs
Marc Olano, SGI / U. Maryland
Jean-Francois Panisset, Discreet
Jon Paul Schelter, Matrox
Brian Paul, Tungsten Graphics
Scott Peterson, HP
Bimal Poddar, Intel
Thomas Roell, Xi Graphics
Phil Rogers, ATI
Ian Romanick, IBM
John Rosasco, Apple
Randi Rost, 3Dlabs
Matt Russo, Matrox
Jeremy Sandmel, ATI
Paul Sargent, 3Dlabs
Folker Schamel, Spinor GMBH
Michael Schulman, Sun
John Scott, Raven Software
Avinash Seetharamaiah, Intel
John Spitzer, NVIDIA
Vlad Stamate, Power VR
Michelle Stamnes, Intel
John Stauffer, Apple
Eskil Steenberg, Obsession
Bruce Stockwell, HP
Christopher Tan, IBM
Ray Tice, Avid
Pierre P. Tremblay, Discreet
Neil Trevett, 3Dlabs
Nick Triantos, NVIDIA
Douglas Twilleager, Sun
Shawn Underwood, SGI
Steve Urquhart, Intelligraphics
Victor Vedovato, ATI
Daniel Vogel, Epic Games
Mik Wells, Softimage
Helene Workman, Apple
Dave Zenz, Dell
Karel Zuiderveld, Vital Images
Appendix L

Version 2.0

OpenGL version 2.0, released on September 7, 2004, is the sixth revision since the original version 1.0. Despite incrementing the major version number (to indicate support for high-level programmable shaders), version 2.0 is upward compatible with earlier versions, meaning that any program that runs with a 1.5, 1.4, 1.3, 1.2, 1.1, or 1.0 GL implementation will also run unchanged with a 2.0 GL implementation.

Following are brief descriptions of each addition to OpenGL 2.0.

L.1 Programmable Shading

The OpenGL Shading Language, and the related APIs to create, manage, and use programmable shaders written in the Shading Language, were promoted to core features in OpenGL 2.0. The complete list of features related to programmable shading includes:

L.1.1 Shader Objects

Shader objects provides mechanisms necessary to manage shader and program objects. Shader objects were promoted from the GL_ARB_shader_objects extension.

L.1.2 Shader Programs

Vertex and fragment shader programs may be written in the high-level OpenGL Shading Language, replacing fixed-functionality vertex and fragment processing respectively. Vertex and fragment shader programs were promoted from the GL_ARB_vertex_shader and GL_ARB_fragment_shader extensions.
L.1.3 OpenGL Shading Language

The OpenGL Shading Language is a high-level, C-like language used to program the vertex and fragment pipelines. The Shading Language Specification defines the language proper, while OpenGL API features control how vertex and fragment programs interact with the fixed-function OpenGL pipeline and how applications manage those programs.

OpenGL 2.0 implementations must support at least revision 1.10 of the OpenGL Shading Language. Implementations may query the SHADING_LANGUAGE_VERSION string to determine the exact version of the language supported. The OpenGL Shading Language was promoted from the GL_ARB_shading_language_100 extension (the shading language itself is specified in a companion document; due to the way it’s written, that document did not need to be changed as a consequence of promoting programmable shading to the OpenGL core).

L.1.4 Changes To Shader APIs

Small changes to the APIs for managing shader and program objects were made in the process of promoting the shader extensions to the OpenGL 2.0 core. These changes do not affect the functionality of the shader APIs, but include use of the existing uint core GL type rather than the new handleARB type introduced by the extensions, and changes in some function names, for example mapping the extension function CreateShaderObjectARB into the core function CreateShader.

L.2 Multiple Render Targets

Programmable shaders may write different colors to multiple output color buffers in a single pass. Multiple render targets was promoted from the GL_ARB_draw_buffers extension.

L.3 Non-Power-Of-Two Textures

The restriction of textures to power-of-two dimensions has been relaxed for all texture targets, so that non-power-of-two textures may be specified without generating errors. Non-power-of-two textures was promoted from the GL_ARB_texture_non_power_of_two extension.
L.4. POINT SPRITES

Point sprites replace point texture coordinates with texture coordinates interpolated across the point. This allows drawing points as customized textures, useful for particle systems.

Point sprites were promoted from the GL_ARB_point_sprite extension, with the further addition of the POINT_SPRITE_COORD_ORIGIN parameter controlling the direction in which the \( t \) texture coordinate increases.

L.5. Separate Blend Equation

Blending capability is extended with \texttt{BlendEquationSeparate} to allow independent setting of the RGB and alpha blend equations for blend operations.

Separate blend functions was promoted from the \texttt{GL_EXT_blend_equation_separate} extension. Note that blend equation \texttt{LOGIC_OP} is not supported unless the \texttt{GL_EXT_blend_logic_op} extension is supported; \texttt{LOGIC_OP} was inadvertently included in the initial release of the OpenGL 2.0 Specification.

L.6. Separate Stencil

Separate stencil functionality may be defined for the front and back faces of primitives, improving performance of shadow volume and Constructive Solid Geometry rendering algorithms.

Separate stencil was based on the the API of the \texttt{GL_ATI_separateStencil} extension, with additional state defined by the similar \texttt{GL_EXT_stencil_two_side} extension.

L.7. Other Changes

Several minor revisions and corrections to the OpenGL 1.5 specification were made:

- In section 2.7, \texttt{SecondaryColor3} was changed to set A to 1.0 (previously 0.0), so the initial GL state can be restored.

- In section 2.18, transformation was added to the list of steps not performed by \texttt{WindowPos}.
• Section 3.9.1 was clarified to mandate that selection of texture internal format must allocate a non-zero number of bits for all components named by the internal format, and zero bits for all other components.

• Tables 3.24 and 3.25 were generalized to multiple textures by replacing $C_f$ with $C_p$.

• In section 6.1.9, GetHistogram was clarified to note that the Final Conversion pixel storage mode is not applied when storing histogram counts.

• The FOGCOORDARRAYBUFFER_BINDING enumerant alias was added to table K.1.

After the initial version of the OpenGL 2.0 was released, several more minor corrections were made in the specification revision approved on October 22, 2004:

• Corrected name of the fog source from FOG_COORD_SRC to FOG_COORD in section 2.18.

• Corrected last parameter type in the declaration of the UniformMatrix* commands to const float *value, in section 2.20.3.

• Changed the end of the second paragraph of the Conversion to Fragments subsection of section 3.7.4, to more clearly describe the set of generated fragments.

• Changed from the older FOG_COORDINATE to the newer FOG_COORD notation in section 3.11.

• Added POINT_SPRITECOORD_ORIGIN state to table 6.16.

• Changed the description of MAX_TEXTURE_UNITS in table 6.48 to reflect its legacy status (referring to the number of fixed-function texture units), and moved it into table 6.49.

• Removed duplicated table entries for MAX_TEXTURE_IMAGE_UNITS and MAX_TEXTURECOORDS from table 6.49.

• Added Victor Vedovato to the OpenGL 2.0 Acknowledgements section.

• Miscellaneous typographical corrections.

Additional minor corrections were made in the specification revision approved on February 9, 2005:
• Restored missing language from the depth texture extension in section 6.1.4, allowing \texttt{DEPTH\_COMPONENT} as a format for texture readbacks.

• Added separate blend equation to the feature list in appendix L. The feature has been in the actual OpenGL 2.0 specification all along, but was omitted from the feature list in the initial specification release.

• Removed \texttt{LOGIC\_OP} from the allowed blend equations in section 4.1.8 and table 4.1, and adjusted the type of the blend equation state in table 6.25 accordingly.

• Restored missing \texttt{VERTEX\_ATTRIB\_ARRAY\_BUFFER\_BINDING} state from table 6.8.

• Miscellaneous typographical corrections.

L.8 Acknowledgements

OpenGL 2.0 is the result of the contributions of many people. The editors especially thank the ongoing work of the ARB GL2 working group, led by Bill Licea-Kane and with specifications edited by John Kessenich and Barthold Lichtenbelt, in performing work necessary to promote the OpenGL Shading Language to a core OpenGL feature.

A partial list of other contributors, including the company that they represented at the time of their contribution, follows:

Kurt Akeley, NVIDIA
Allen Akin
Dave Baldwin, 3Dlabs
Bob Beretta, Apple
Pat Brown, NVIDIA
Matt Craighead, NVIDIA
Suzy Deffeyes, IBM
Ken Dyke, Apple
Cass Everitt, NVIDIA
Steve Glanville, NVIDIA
Michael I. Gold, NVIDIA
Evan Hart, ATI
Phil Huxley, 3Dlabs
Deron Dann Johnson, Sun
John Kessenich, 3Dlabs
Mark Kilgard, NVIDIA

Version 3.0 (with changes marked - August 11, 2008)
L.8. ACKNOWLEDGEMENTS

Dale Kirkland, 3Dlabs
Steve Koren, 3Dlabs
Jon Leech, SGI
Bill Licea-Kane, ATI
Barthold Lichtenbelt, 3Dlabs
Kent Lin, Intel
Benjamin Lipchak, ATI
Rob Mace, ATI
Michael McCool, U. Waterloo
Jack Middleton, Sun
Jeremy Morris, 3Dlabs
Teri Morrison, 3Dlabs
Marc Olano, SGI / U. Maryland
Glenn Ortner, ATI
Brian Paul, Tungsten Graphics
Bimal Poddar, Intel
Phil Rogers, ATI
Ian Romanick, IBM
Randi Rost, 3Dlabs
Jeremy Sandmel, ATI
Folker Schamel, Spinor GMBH
Geoff Stahl, Apple
Eskil Steenberg, Obsession
Neil Trevett, 3Dlabs
Victor Vedovato, ATI
Mik Wells, Softimage
Esen Yilmaz, Intel
Dave Zenz, Dell

Version 3.0 (with changes marked - August 11, 2008)
Appendix M

Version 2.1

OpenGL version 2.1, released on August 2, 2006, is the seventh revision since the original version 1.0. Despite incrementing the major version number (to indicate support for high-level programmable shaders), version 2.1 is upward compatible with earlier versions, meaning that any program that runs with a 2.0, 1.5, 1.4, 1.3, 1.2, 1.1, or 1.0 GL implementation will also run unchanged with a 2.0 GL implementation.

Following are brief descriptions of each addition to OpenGL 2.1.

M.1 OpenGL Shading Language

OpenGL 2.1 implementations must support at least revision 1.20 of the OpenGL Shading Language. Implementations may query the SHADING_LANGUAGE_VERSION string to determine the exact version of the language supported. Refer to the OpenGL Shading Language Specification for details of the changes between revision 1.10 and 1.20.

M.2 Non-Square Matrices

Added the UniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}fv commands in section 2.20.3, allowing specification of non-square uniform matrices.

M.3 Pixel Buffer Objects

Pixel buffer objects expand on the interface provided by the vertex buffer objects, allowing buffer objects to be used with both vertex array and pixel data. This allows
more acceleration opportunities for OpenGL pixel commands.

When a buffer object is bound to the `PIXEL_PACK_BUFFER` target, commands such as `ReadPixels` write their data into a buffer object. When a buffer object is bound to the `PIXEL_UNPACK_BUFFER` target, commands such as `DrawPixels` and `TexImage2D` read their data from a buffer object.

Pixel buffer objects was promoted from the `GL_ARB_pixel_buffer_object` extension. The specification was tightened to consistently require error be generated when read or write operations to a pixel buffer object would run past the end of the buffer, or would be misaligned with respect to the data type being read or written.

### M.4 sRGB Textures

New uncompressed and compressed color texture formats with sRGB color components are defined. The sRGB color space is based on typical (non-linear) monitor characteristics expected in a dimly lit office. It has been standardized by the International Electrotechnical Commission (IEC) as IEC 61966-2-1. The sRGB color space roughly corresponds to 2.2 gamma correction.

sRGB textures was promoted from the `GL_EXT_texture_sRGB` extension. Specific compressed sRGB internal formats defined by the extension were not included in OpenGL 2.1, while the generic uncompressed and compressed formats were retained.

### M.5 Other Changes

Several minor revisions and corrections to the OpenGL 2.0 specification were made:

- Note that the information log for program objects can include both link and validation information, in section 2.20.2.

- Noted in section 3.7.4 that there is a current raster secondary color, and added the `CURRENT_RASTER_SECONDARY_COLOR` query.

- Required perspective-correct interpolation for all fragment attributes except depth in sections 3.5.1 and 3.6.1, effectively making `GL_PERSPECTIVE_CORRECT_HINT` a no-op.

- Merged specific and generic compressed internal texture format tables into the single table 3.19.
• Changed the type of texture wrap mode and min/mag filter parameters from integer to enum in table 3.20.

• Removed mention of compressed texture depth components from section 3.9.1, since no compressed depth formats are currently defined.

• Added forward reference from section 3.9.5 to section 3.9.14, which defines how depth textures are actually used.

• Remove \( c \) notation in table 4.1, and fixed blend equations for FUNC.REVERSE_SUBTRACT.

• Noted in section 6.1.18 that all texture object parameters are pushed and popped by PushAttrib and PopAttrib when TEXTURE_BIT is set in the attribute mask.

• Miscellaneous typographical corrections.

Additional minor corrections were made in a specification revision approved on August 10, 2006, with a few additional fixes and omissions corrected on December 1, 2006:

• Noted in section 2.5 that INVALID_VALUE is generated for negative sizeiptr values.

• Noted in section 2.7 that VertexAttrib* can be used to load attributes for any supported matrix type, not just square matrices.

• Removed the description of generating multiple connected components when clipping vertices with \( w_c \) values of different signs, at the end of section 2.17. Implementations should not render in the \( w_c < 0 \) region.

• Added FLOAT_MAT2x3, FLOAT_MAT2x4, FLOAT_MAT3x2, FLOAT_MAT3x4, FLOAT_MAT4x2, and FLOAT_MAT4x3 tokens for non-square matrix types to GetActiveAttrib and GetActiveUniform in section 2.20.3, and expanded the description of how attribute components are mapped to matrix elements to match.

• Clarified in section 3.9.12 that the texture object passed to BindTexture must match the specified target, not just the dimensionality of target.

• Added missing TexEnv targets RGB_SCALE, ALPHA_SCALE, SRC\( n \_ \)RGB, SRC\( n \_ \)ALPHA, OPERAND\( n \_ \)RGB, and OPERAND\( n \_ \)ALPHA in section 3.9.13.
• Noted that `POINT_SPRITE` is a possible `env` parameter to `GetTexEnv` in section 6.1.3.

• Miscellaneous typographical corrections.

Additional minor corrections (in process) were made after December 1, 2006:

• Corrected spelling of `ACTIVE_ATTRIBUTE_MAX_LENGTH` in table 6.40.

• Clarified in section 4.1.12 that logical operations are also performed for each sample when multisampling is active (Khronos internal bug 1022, Bill Licea-Kane).

• Added missing `QUERY_RESULT` and `QUERY_RESULT_AVAILABLE` to state table 6.52.

• Specified in section 3.7.3 that floating-point `PixelFormat` entries are clamped to the range `[0, 1]` when specified (bug 3379, Mark Kilgard).

• Corrected description of type of `TEXTURE_MIN_LOD` and `TEXTURE_MAX_LOD` in section 3.9.11 (bug 2861, Bill Licea-Kane).

• Changed number of sets of texture state from two to four in section 3.9.11 (bug 2862, Bill Licea-Kane).

• Specified in section 5.4 that floating-point offsets to `CallLists` are truncated towards negative infinity (bug 3379, Mark Kilgard).

• Specified that the data conversion rules for integer queries of floating-point state defined in section 6.1.2 also apply to the enumerated queries in section 6.1.3 (bug 3379, Mark Kilgard).

• Removed `DRAW_BUFFER` state from table 6.26 and added descriptions of `DRAW_BUFFERi` and `DRAW_BUFFER` in section 4.2.1 (bug 1029, Jeff Juliano).

M.6 Acknowledgements

OpenGL 2.1 is the result of the contributions of many people. The editor especially thanks the ongoing work of the ARB GLSL working group, led by Bill Licea-Kane and with specifications edited by John Kessenich and Barthold Lichtenbelt, in updating the OpenGL Shading Language to revision 1.20. Ralf Biermann, Derek Cornish, Matt Craighead, and Mark Kilgard edited the `EXT_pixel_buffer_object`
proposal later adopted and developed by the ARB Pixel Buffer Object working group, and Mark Kilgard edited the `EXT_texture_sRGB` extension.

A partial list of other contributors, including the company that they represented at the time of their contribution, follows:

- Aaftab Munshi, ATI
- Avi Shapira, Graphic Remedy
- Barthold Lichtenbelt, 3Dlabs / NVIDIA
- Benjamin Lipchak, ATI
- Benji Bowman, Imagination Technologies
- Bill Armstrong, Evans and Sutherland
- Bill Licea-Kane, ATI
- Bimal Poddar, Intel
- Bob Beretta, Apple
- Brian Paul, Tungsten Graphics
- Cass Everitt, NVIDIA
- Chris Dodd, NVIDIA
- Chris Starkey, 3Dlabs
- Dale Kirkland, 3Dlabs
- Daniel Vogel, Epic Games
- Dave Shreiner, SGI
- Derek Cornish, Obsession
- Evan Hart, ATI
- Folker Schamel, Spinor GMBH
- Geoff Stahl, Apple
- Howard Miller, Apple
- Ian Romanick, IBM
- James A. McCombe, Apple
- Jeff Juliano, NVIDIA
- Jeff Weyman, ATI
- Jeremy Sandmel, Apple / ATI
- John Kessenich, 3Dlabs / Intel
- John Rosasco, Apple
- John Scott
- Jon Leech, SGI / Independent
- Jon Trulson, Xi Graphics
- Ken Severson, NVIDIA
- Kenneth Dyke, Apple
- Kenneth Russell, Sun
- Kent Lin, Intel

Version 3.0 (with changes marked - August 11, 2008)
M.6. ACKNOWLEDGEMENTS

Marc Olano, U. Maryland
Mark Kilgard, NVIDIA
Michael Gold, NVIDIA
Neeraj Srivastava, Dell
Neil Trevett, 3Dlabs / NVIDIA
Nick Burns, Apple
Pat Brown, NVIDIA
Paul Martz, SimAuthor
Paul Ramsey, Sun
Pierre Boudier, ATI
Ralf Biermann, NVIDIA
Randi Rost, 3Dlabs
Rob Mace, ATI
Robert Simpson, Bitboys / ATI
Saifuddin Fakhruddin, Intel
Shawn Underwood, SGI
Steve Demlow, Vital Images
Steve Koren, 3Dlabs
Steven Zhu, Intel
Thomas Roell, NVIDIA
Tom Lanzoni, Dell
Travis Bryson, Sun
Yaki Tebeka, Graphic Remedy
Appendix N

Version 3.0

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.

N.1 New Features

New features in OpenGL 3.0, including the extension or extensions if any on which they were based, include:

- API support for the new texture lookup, texture format, and integer and unsigned integer capabilities of the OpenGL Shading Language 1.30 specification (GL_EXT_gpu_shader4).
- Conditional rendering (GL_NV_conditional_render).
- Fine control over mapping buffer subranges into client space and flushing modified data (GL_APPLE_flush_buffer_range).
- Floating-point color and depth internal formats for textures and renderbuffers (GL_ARB_color_buffer_float, GL_NV_depth_buffer_float).
N.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.
N.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 N.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 N.1: New token names and the old names they replace.

N.4 Credits and Acknowledgements

OpenGL 3.0 is the result of the contributions of many people and companies. Members of the Khronos OpenGL ARB Working Group during the development of OpenGL 3.0, including the company that they represented at the time of their contributions, follow. Some major contributions made by individuals are listed together with their name, including specific functionality developed in the form of new ARB extensions together with OpenGL 3.0. In addition, many people participated in developing earlier vendor and EXT extensions on which the OpenGL 3.0 functionality is based in part; those individuals are listed in the respective extension specifications in the OpenGL Extension Registry.

Aaftab Munshi, Apple
Alain Bouchard, Matrox
Alexis Mather, AMD (Chair, ARB Marketing TSG)
Andreas Wolf, AMD
Avi Shapira, Graphic Remedy
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

Version 3.0 (with changes marked - August 11, 2008)
N.4. CREDITS AND ACKNOWLEDGEMENTS

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
David Garcia, Imagination Technologies
Eric Boumaour, AMD
Eskil Steenberg, Obsession
Evan Hart, NVIDIA
Folker Schamel, Spinor GMBH
Gavriel State, Transgaming
Geoff Stahl, Apple
Georg Kolling, Imagination Technologies
Gregory Prisament, NVIDIA
Guillaume Portier, HI Corp
Ian Romanick, IBM / Intel (Vertex array objects; GLX protocol)
James Helferty, Transgaming (Instanced rendering)
James Jones, NVIDIA
Jamie Gennis, NVIDIA
Jason Green, Transgaming
Jeff Bolz, NVIDIA
Jeff Juliano, NVIDIA
Jeremy Sandmel, Apple (Chair, ARB Nextgen (OpenGL 3.0) TSG)
John Kessenich, Intel (OpenGL Shading Language Specification Editor; depre-
cation model)
John Rosasco, Apple
Jon Leech, Independent (Chair, ARB Ecosystem TSG; OpenGL API Specifica-
tion 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
based)
Matti Paavola, Nokia
Michael Gold, NVIDIA (Framebuffer objects and instanced rendering)
Neil Trevett, NVIDIA (President, Khronos Group)
Nick Burns, Apple

Version 3.0 (with changes marked - August 11, 2008)
N.4. CREDITS AND ACKNOWLEDGEMENTS

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 O

ARB Extensions

OpenGL extensions that have been approved by the OpenGL Architectural Review Board (ARB) are described in this chapter. These 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.

In order not to compromise the readability of the core specification, ARB extensions are not integrated into the core language; instead, they are made available online in the OpenGL Extension Registry (as are a much larger number of vendor-specific extensions, as well as extensions to window system binding APIs, such as GLX and WGL). Extensions are documented as changes to the Specification. The Registry is available on the World Wide Web at URL

http://www.opengl.org/registry/

Brief descriptions of ARB extensions are provided below.

O.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 present in the EXTENSIONS string described in section 6.1.11.

- All functions defined by the extension will have names of the form FunctionARB
O.2. PROMOTING EXTENSIONS TO CORE FEATURES

- All enumerants defined by the extension will have names of the form \texttt{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 \texttt{"GLX\_"} and \texttt{"WGL\_"} respectively. Not all GLX and WGL ARB extensions are described here, but all such extensions are included in the registry.

O.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 \texttt{EXTENSIONS} string, 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 appendices I, J, K, and L respectively.

O.3 Multitexture

The name string for multitexture is \texttt{GL\_ARB\_multitexture}. It was promoted to a core feature in OpenGL 1.3.

O.4 Transpose Matrix

The name string for transpose matrix is \texttt{GL\_ARB\_transpose\_matrix}. It was promoted to a core feature in OpenGL 1.3.

O.5 Multisample

The name string for multisample is \texttt{GL\_ARB\_multisample}. It was promoted to a core feature in OpenGL 1.3.
O.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.

O.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.

O.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.

O.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.

O.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.

O.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.
O.12  Matrix Palette

Matrix palette extends vertex blending to include a palette of model-view matrices. Each vertex may be transformed by a different set of matrices chosen from the palette.

The name string for matrix palette is GL_ARB_matrix_palette.

O.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.

O.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.

O.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.

O.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.

O.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.

O.18  Shadow

The name string for shadow is GL_ARB_shadow. It was promoted to a core feature in OpenGL 1.4.
O.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`.

O.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.

O.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`.

O.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`.

O.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.
O.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.

O.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.

O.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.

O.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.

O.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.

O.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.
O.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.

O.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.

O.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.

O.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-normalizes texture coordinates.

The name string for texture rectangles is GL_ARB_texture_rectangle.

O.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
O.35  HALF-PRECISION FLOATING POINT

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.

O.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.

O.36  Floating-Point Textures

Floating-point textures stored in both 32- and 16-bit formats may be defined using new internalformat arguments to commands which specify and read back texture images.

The name string for floating-point textures is GL_ARB_texture_float.

O.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.
Index

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS, ALPHABITS</td>
<td>293, 406</td>
</tr>
<tr>
<td>ACCESS, ALPHA_INTEGER</td>
<td>152</td>
</tr>
<tr>
<td>ACCESS, ALPHA_BIAS</td>
<td>163</td>
</tr>
<tr>
<td>ACCESS, ALPHA_TEST</td>
<td>241, 406</td>
</tr>
<tr>
<td>AlphaFunc</td>
<td>241, 406</td>
</tr>
<tr>
<td>ALWAYS</td>
<td>201, 224, 242–244, 361</td>
</tr>
<tr>
<td>AMBIENT, ALPHABITS</td>
<td>81, 83</td>
</tr>
<tr>
<td>AMBIENT, AND_DIFFUSE</td>
<td>81, 83</td>
</tr>
<tr>
<td>AND, 252</td>
<td></td>
</tr>
<tr>
<td>AND, INVERTED</td>
<td>252</td>
</tr>
<tr>
<td>AND, REVERSE</td>
<td>252</td>
</tr>
<tr>
<td>ALEMBIANT, AND苦しע, 81, 83</td>
<td></td>
</tr>
<tr>
<td>AND, 252</td>
<td></td>
</tr>
<tr>
<td>AND, INVERTED</td>
<td>252</td>
</tr>
<tr>
<td>AND, REVERSE</td>
<td>252</td>
</tr>
<tr>
<td>Antialiasing</td>
<td>128</td>
</tr>
<tr>
<td>AreaTexturesResident</td>
<td>217, 309, 405</td>
</tr>
<tr>
<td>ARRAY_BUFFER</td>
<td>38, 40–43, 45–48, 330, 331</td>
</tr>
<tr>
<td>ARRAY_BUFFER_BINDING</td>
<td>46</td>
</tr>
<tr>
<td>ArrayElement</td>
<td>24, 32–34, 46, 106, 307</td>
</tr>
<tr>
<td>ATTACHED_SHADERS</td>
<td>333</td>
</tr>
<tr>
<td>AttachShader</td>
<td>90, 309</td>
</tr>
<tr>
<td>AUTO_NORMAL</td>
<td>103, 298</td>
</tr>
<tr>
<td>AUX, 254–256, 261, 318</td>
<td></td>
</tr>
<tr>
<td>AUX0, 254</td>
<td></td>
</tr>
<tr>
<td>AUX_BUFFERS</td>
<td>254, 279</td>
</tr>
<tr>
<td>BACK, 80, 82, 83, 129, 130, 132, 243, 246, 254, 255, 257–259, 267, 415</td>
<td></td>
</tr>
<tr>
<td>ACTIVE_ATTRIBUTE_MAX_LENGTH,</td>
<td>93, 333, 451</td>
</tr>
<tr>
<td>ACTIVE_ATTRIBUTES</td>
<td>93, 333</td>
</tr>
<tr>
<td>ACTIVE_TEXTURE</td>
<td>25, 55, 56, 70, 218, 297, 315</td>
</tr>
<tr>
<td>ACTIVE_UNIFORM_MAX_LENGTH</td>
<td>97, 333</td>
</tr>
<tr>
<td>ACTIVE_UNIFORMS</td>
<td>96, 333</td>
</tr>
<tr>
<td>ActiveTexture</td>
<td>55, 56, 100, 226</td>
</tr>
<tr>
<td>ADD, 218, 220, 221, 262, 263, 424</td>
<td></td>
</tr>
<tr>
<td>ADD, 221</td>
<td></td>
</tr>
<tr>
<td>ADD, 218, 220, 221, 262, 263, 424</td>
<td></td>
</tr>
<tr>
<td>ADD, 221</td>
<td></td>
</tr>
<tr>
<td>ALL_ATTRIB_BITS</td>
<td>337, 338, 406</td>
</tr>
</tbody>
</table>
INDEX

276, 316, 354, 404
BACK_LEFT, 255, 256, 261, 318
BACK_RIGHT, 255, 256, 261, 318
BeginConditionalRender, 62, 63
BeginQuery, 61, 62, 67, 244
BeginTransformFeedback, 63–66
BGR, 152, 267, 270, 321
BGR_INTEGER, 152
BGRA, 152, 155, 159, 267, 321, 413
BGRA_INTEGER, 152
BindAttribLocation, 94, 309
BindBuffer, 38, 46, 65, 309
BindBufferBase, 64–66, 309
BindBufferRange, 64–66, 309
BindFramebuffer, 277, 279, 291, 309
BindRenderbuffer, 280, 281, 309
BindTexture, 56, 100, 215, 216, 450
BindVertexArray, 48, 309
BITMAP, 131, 140, 143, 150, 151, 160, 173, 270, 323
Bitmap, 62, 113, 173, 234, 405
BITMAP_TOKEN, 306
BLEND, 218, 220, 245, 246, 248, 251
BlendColor, 248, 431
BlendEquation, 246, 431
BlendEquationSeparate, 246, 444
BlendFunc, 248, 431
BlendFuncSeparate, 248, 432
BlitFramebuffer, 272, 275, 309
BLUE_BIAS, 163
BLUE_BITS, 293, 406
BLUE_INTEGER, 152
BLUE_SCALE, 163
BOOL, 97
BOOL_VEC2, 97
BOOL_VEC3, 97
BOOL_VEC4, 97
BUFFER_ACCESS, 39, 41, 42
BUFFER_MAP_POINTER, 39, 41–43, 331
BUFFER_MAPPED, 39, 41–43
BUFFER_SIZE, 39, 41, 45, 65
BUFFER_USAGE, 39, 41, 42
BufferData, 40, 41, 309
BufferSubData, 41, 42, 309
bvec2, 99
BYTE, 29, 151, 270, 271, 308
C3F_V3F, 36, 37
C4F_N3F_V3F, 36, 37
C4UB_V2F, 36, 37
C4UB_V3F, 36, 37
CallList, 24, 307, 308, 406
CallLists, 24, 307, 308, 406, 451
CCW, 79, 354
CheckFramebufferStatus, 291, 292, 309
CLAMP, 201, 205, 206, 405
CLAMP_FRAGMENT_COLOR, 161, 275
CLAMP_READ_COLOR, 269
CLAMP_TO_BORDER, 201, 206, 405, 424
CLAMP_TO_EDGE, 201, 206, 275, 414
CLAMP_VERTEX_COLOR, 85
ClampColor, 85, 161, 404
CLEAR, 252
Clear, 62, 113, 259, 260, 262, 406
ClearAccum, 260, 406
ClearBuffer, 261, 262

Version 3.0 (with changes marked - August 11, 2008)
INDEX

ClearBufferf, 262
ClearBufferfi, 261
ClearBufferfv, 261
ClearBufferi, 262
ClearBufferiv, 261
ClearBufferui, 261
ClearColor, 259, 261
ClearDepth, 259, 261
ClearIndex, 260
ClearStencil, 260, 261
CLIENT_ACTIVE_TEXTURE, 31, 315
CLIENT_ALL_ATTRIB_BITS, 337, 338, 406
CLIENT_PIXEL_STORE_BIT, 338
CLIENT_VERTEX_ARRAY_BIT, 338
ClientActiveTexture, 24, 31, 309, 403
CLIP_DISTANCEi, 456
CLIP_PLANEi, 67, 68, 456
CLIP_PLANE0, 68
ClipPlane, 67
COEFF, 318
COLOR, 51, 56, 57, 141, 145, 146, 191, 272
Color, 24, 26, 63, 73, 83, 87, 92
Color*, 403
Color3, 26
Color4, 26
Color[size][type]v, 32
COLOR_ARRAY, 31, 36
COLOR_ARRAY_POINTER, 327
COLOR_ATTACHMENTi, 254, 255, 267, 284, 289
COLOR_ATTACHMENTm, 254, 257
COLOR_ATTACHMENTn, 278
COLOR_ATTACHMENT0, 254, 257, 267, 278
COLOR_BUFFER_BIT, 259, 262, 274, 275, 338
COLOR_INDEX, 131, 140, 143, 148, 150, 152, 164, 173, 267, 272, 322, 323, 403
COLOR_INDEXES, 81, 84
COLOR_LOGIC_OP, 251
COLOR_MATERIAL, 83, 404
COLOR_MATRIX, 323
COLOR_MATRIX_STACK_DEPTH, 323
COLOR_SUM, 228, 406
COLOR_TABLE, 140, 143, 165
COLOR_TABLE_ALPHA_SIZE, 324
COLOR_TABLE_BIAS, 140, 141, 324
COLOR_TABLE_BLUE_SIZE, 324
COLOR_TABLE_FORMAT, 324
COLOR_TABLE_GREEN_SIZE, 324
COLOR_TABLE_INTENSITY_SIZE, 324
COLOR_TABLE_LUMINANCE_SIZE, 324
COLOR_TABLE_RED_SIZE, 324
COLOR_TABLE_SCALE, 140, 141, 324
COLOR_TABLE_WIDTH, 324
COLOR_WRITEMASK, 258
ColorMask, 257–259
ColorMaski, 258
ColorMaterial, 82, 83, 298, 395, 404, 411
ColorPointer, 24, 29, 30, 36, 309, 403
ColorSubTable, 136, 141, 142
ColorTable, 136, 139, 141–143, 147, 148, 170, 171, 310
ColorTableParameter, 141
ColorTableParameterfv, 140
Colorub, 87
Colorui, 87
Colorus, 87
COMBINE, 218, 219, 221, 226, 424, 432
COMBINE_ALPHA, 218, 219, 221, 222
COMBINE_RGB, 218, 219, 221, 222
COMPARE_R_TO_TEXTURE, 456
COMPARE_REF_TO_TEXTURE, 201, 224, 456
COMPILE, 307, 395
COMPILE_AND_EXECUTE, 307, 308
COMPILE_STATUS, 89, 332
CompileShader, 89, 310
COMPRESSED_ALPHA, 185
COMPRESSED_INTENSITY, 185
COMPRESSED_LUMINANCE, 185
INDEX

COMPRESSED_LUMINANCE_ALPHA, 185
COMPRESSED_RED, 185
COMPRESSED_RED_RGTC1, 179, 185, 398–400
COMPRESSED_RG, 185
COMPRESSED_RG_RGTC2, 179, 185, 400
COMPRESSED_RGB, 185
COMPRESSED_RGBA, 185
COMPRESSED_SIGNED_RED_RGTC1, 179, 185, 399, 400
COMPRESSED_SIGNED_RG_RGTC2, 179, 185, 400
COMPRESSED_SLUMINANCE, 185, 225
COMPRESSED_SLUMINANCE_ALPHA, 185, 225
COMPRESSED_SRGB, 185, 225
COMPRESSED_SRGB_ALPHA, 185, 225
COMPRESSED_TEXTURE_FORMATS, 178
CompressedTexImage, 198
CompressedTexImage*, 291
CompressedTexImage1D, 196–198
CompressedTexImage2D, 196–198
CompressedTexImage3D, 196–198
CompressedTexSubImage1D, 197–199
CompressedTexSubImage2D, 197–199
CompressedTexSubImage3D, 197–199
CONSTANT, 222, 223, 360
CONSTANT_ALPHA, 249, 431
CONSTANT_ATTENUATION, 81
CONSTANT_BORDER, 167, 168
CONSTANT_COLOR, 249, 431
CONTEXT_FLAG_FORWARD_COMPATIBLE_BIT, 328
CONTEXT_FLAGS, 328
CONVOLUTION_1D, 144, 145, 165, 188, 324, 325
CONVOLUTION_2D, 143–145, 165, 188, 324, 325
CONVOLUTION_BORDER_COLOR, 168, 325
CONVOLUTION_BORDER_MODE, 167, 325
CONVOLUTION_FILTER_BIAS, 143–145, 325
CONVOLUTION_FILTER_SCALE, 143–146, 325
CONVOLUTION_FORMAT, 325
CONVOLUTION_HEIGHT, 325
CONVOLUTION_WIDTH, 325
ConvolutionFilter1D, 136, 144–146
ConvolutionFilter2D, 136, 143–146
ConvolutionParameter, 144, 167
ConvolutionParameterfv, 143, 144, 168
ConvolutionParameteriv, 145, 168
COORD_REPLACE, 118, 121
COPY, 252, 362
COPY_INVERTED, 252
COPY_PIXEL_TOKEN, 306
CopyColorSubTable, 142, 272
CopyColorTable, 141, 142, 272
CopyConvolutionFilter*, 272
CopyConvolutionFilter1D, 145, 146
CopyConvolutionFilter2D, 145, 146
CopyTexImage, 293
CopyTexImage*, 272, 284, 291
CopyTexImage1D, 165, 191, 193, 195, 209
CopyTexImage2D, 165, 189, 191, 193, 195, 209
CopyTexImage3D, 193
CopyTexSubImage, 293
CopyTexSubImage*, 195, 199, 272, 284
CopyTexSubImage1D, 165, 192–195
CopyTexSubImage2D, 165, 192–195
CopyTexSubImage3D, 165, 192, 193, 195
CreateProgram, 90, 309
CreateShader, 88, 309, 443
CreateShaderObjectARB, 443
CULL_FACE, 129
CullFace, 129, 130, 134
CURRENT_BIT, 338

Version 3.0 (with changes marked - August 11, 2008)
INDEX

<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT_FOG_COORD</td>
<td>438</td>
</tr>
<tr>
<td>CURRENT_FOG_COORDINATE</td>
<td>438</td>
</tr>
<tr>
<td>CURRENT_QUERY</td>
<td>329</td>
</tr>
<tr>
<td>CURRENT_RASTER_SECONDARY_COLOR</td>
<td>449</td>
</tr>
<tr>
<td>CURRENT_RASTER_TEXTURE_COORDS</td>
<td>70, 394</td>
</tr>
<tr>
<td>CURRENT_TEXTURE_COORDS</td>
<td>25</td>
</tr>
<tr>
<td>CURRENT_VERTEX_ATTRIB</td>
<td>335</td>
</tr>
<tr>
<td>CW</td>
<td>79</td>
</tr>
<tr>
<td>DECAL</td>
<td>218, 220</td>
</tr>
<tr>
<td>DECR</td>
<td>243</td>
</tr>
<tr>
<td>DECR_WRAP</td>
<td>243, 432</td>
</tr>
<tr>
<td>DELETE_STATUS</td>
<td>89, 332</td>
</tr>
<tr>
<td>DeleteBuffers</td>
<td>39, 309</td>
</tr>
<tr>
<td>DeleteFramebuffers</td>
<td>277, 279, 309</td>
</tr>
<tr>
<td>DeleteLists</td>
<td>309, 406</td>
</tr>
<tr>
<td>DeleteProgram</td>
<td>91, 309</td>
</tr>
<tr>
<td>DeleteQueries</td>
<td>62, 309</td>
</tr>
<tr>
<td>DeleteRenderbuffers</td>
<td>281, 291, 309</td>
</tr>
<tr>
<td>DeleteShader</td>
<td>89, 309</td>
</tr>
<tr>
<td>DeleteTextures</td>
<td>216, 291, 309</td>
</tr>
<tr>
<td>DeleteVertexArrays</td>
<td>47, 48, 309</td>
</tr>
<tr>
<td>DEPTH</td>
<td>191, 272, 356, 370, 431</td>
</tr>
<tr>
<td>DEPTH24_STENCIL8</td>
<td>180, 184</td>
</tr>
<tr>
<td>DEPTH32F_STENCIL8</td>
<td>180, 184</td>
</tr>
<tr>
<td>DEPTH_ATTACHMENT</td>
<td>261, 278, 284, 289</td>
</tr>
<tr>
<td>DEPTH_BIAS</td>
<td>138, 163</td>
</tr>
<tr>
<td>DEPTH_BITS</td>
<td>274, 293, 406</td>
</tr>
<tr>
<td>DEPTH_BUFFER</td>
<td>261, 318</td>
</tr>
<tr>
<td>DEPTH_BUFFER_BIT</td>
<td>259, 262, 274, 275, 338</td>
</tr>
<tr>
<td>DEPTH_COMPONENT16</td>
<td>179, 184</td>
</tr>
<tr>
<td>DEPTH_COMPONENT24</td>
<td>179, 184</td>
</tr>
<tr>
<td>DEPTH_COMPONENT32</td>
<td>184</td>
</tr>
<tr>
<td>DEPTH_COMPONENT32F</td>
<td>179, 184</td>
</tr>
<tr>
<td>DEPTH_SCALE</td>
<td>138, 163</td>
</tr>
<tr>
<td>DEPTH_STENCIL_ATTACHMENT</td>
<td>261, 283, 284, 286, 319</td>
</tr>
<tr>
<td>DEPTH_TEST</td>
<td>244</td>
</tr>
<tr>
<td>DEPTH_TEXTURE_MODE</td>
<td>201, 214, 223, 224, 405</td>
</tr>
<tr>
<td>DepthFunc</td>
<td>244</td>
</tr>
<tr>
<td>DepthMask</td>
<td>258, 259, 264</td>
</tr>
<tr>
<td>DepthRange</td>
<td>51, 71, 314, 395</td>
</tr>
<tr>
<td>DepthTest</td>
<td>264</td>
</tr>
<tr>
<td>DetachShader</td>
<td>90, 309</td>
</tr>
<tr>
<td>dFdx</td>
<td>311</td>
</tr>
<tr>
<td>dFdy</td>
<td>311</td>
</tr>
<tr>
<td>DIFFUSE</td>
<td>81, 83</td>
</tr>
<tr>
<td>Disable</td>
<td>55–57, 60, 68, 75, 79, 80, 83, 113, 115, 117, 123, 125, 129, 131, 134, 170, 171, 225, 228, 240–242, 244, 246, 251, 297, 298, 403, 404, 406</td>
</tr>
<tr>
<td>DisableClientState</td>
<td>24, 31, 36, 38, 309, 403</td>
</tr>
<tr>
<td>Disablei</td>
<td>245</td>
</tr>
<tr>
<td>DisableVertexAttribArray</td>
<td>31, 309, 335</td>
</tr>
<tr>
<td>DITHER</td>
<td>251</td>
</tr>
<tr>
<td>DOMAIN</td>
<td>318</td>
</tr>
<tr>
<td>DONT_CARE</td>
<td>311, 312, 382</td>
</tr>
<tr>
<td>DOT3_RGB</td>
<td>221</td>
</tr>
<tr>
<td>DOT3_RGBA</td>
<td>221</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>29, 32</td>
</tr>
<tr>
<td>DRAW_BUFFER</td>
<td>254, 257, 267, 451</td>
</tr>
<tr>
<td>DRAW_BUFFERi</td>
<td>245, 246, 257, 258, 290, 451</td>
</tr>
<tr>
<td>DRAW_BUFFER0</td>
<td>257</td>
</tr>
<tr>
<td>DRAW_FRAMEBUFFER</td>
<td>263, 277–279, 283, 292, 318, 364</td>
</tr>
<tr>
<td>DRAW_FRAMEBUFFER_BINDING</td>
<td>23, 150, 173, 208, 253, 254, 256, 272, 275, 292–294</td>
</tr>
<tr>
<td>DRAW_PIXEL_TOKEN</td>
<td>306</td>
</tr>
<tr>
<td>DrawArrays</td>
<td>33, 34, 46, 48, 107, 307</td>
</tr>
<tr>
<td>DrawBuffer</td>
<td>252–255, 257, 259, 262</td>
</tr>
<tr>
<td>DrawBuffers</td>
<td>253, 254, 256, 257</td>
</tr>
<tr>
<td>Command</td>
<td>Pages</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>DrawElements</td>
<td>34, 35, 46, 48, 64, 107, 307, 415</td>
</tr>
<tr>
<td>DrawRangeElements</td>
<td>35, 46, 107, 307, 385</td>
</tr>
<tr>
<td>DST_ALPHA</td>
<td>249</td>
</tr>
<tr>
<td>DST_COLOR</td>
<td>249, 431</td>
</tr>
<tr>
<td>DYNAMIC_COPY</td>
<td>39, 41</td>
</tr>
<tr>
<td>DYNAMIC_DRAW</td>
<td>39, 40</td>
</tr>
<tr>
<td>DYNAMIC_READ</td>
<td>39, 41</td>
</tr>
<tr>
<td>EDGE_FLAG_ARRAY</td>
<td>31, 36</td>
</tr>
<tr>
<td>EDGE_FLAG_ARRAY_POINTER</td>
<td>327</td>
</tr>
<tr>
<td>EdgeFlag</td>
<td>23, 24</td>
</tr>
<tr>
<td>EdgeFlag*</td>
<td>403</td>
</tr>
<tr>
<td>EdgeFlagPointer</td>
<td>24, 29, 30, 309, 403</td>
</tr>
<tr>
<td>ElementArrayBuffer</td>
<td>38, 40, 42, 43, 45–47, 330, 331</td>
</tr>
<tr>
<td>EMISSION</td>
<td>81, 83</td>
</tr>
<tr>
<td>Enable</td>
<td>55–57, 60, 68, 75, 79, 80, 83, 113, 115, 117, 123, 125, 129, 131, 134, 170, 171, 225, 228, 240–242, 244, 246, 251, 297, 298, 314, 403, 404, 406</td>
</tr>
<tr>
<td>ENABLE_BIT</td>
<td>338</td>
</tr>
<tr>
<td>EnableClientState</td>
<td>24, 31, 36, 38, 309, 403</td>
</tr>
<tr>
<td>Enabler</td>
<td>245</td>
</tr>
<tr>
<td>EnableVertexAttribArray</td>
<td>31, 48, 309, 335</td>
</tr>
<tr>
<td>End</td>
<td>16, 20–24, 33, 34, 49, 62, 80, 83, 86, 122, 129, 132, 292, 299, 300, 305, 403</td>
</tr>
<tr>
<td>EndConditionalRender</td>
<td>62, 63</td>
</tr>
<tr>
<td>EndList</td>
<td>307, 406</td>
</tr>
<tr>
<td>EndQuery</td>
<td>61, 62, 244, 245</td>
</tr>
<tr>
<td>EndTransformFeedback</td>
<td>64</td>
</tr>
<tr>
<td>EQUAL</td>
<td>201, 224, 242–244</td>
</tr>
<tr>
<td>EQUIV</td>
<td>252</td>
</tr>
<tr>
<td>EVAL_BIT</td>
<td>338</td>
</tr>
<tr>
<td>EvalCoord</td>
<td>24, 297, 298</td>
</tr>
<tr>
<td>EvalCoord*</td>
<td>406</td>
</tr>
<tr>
<td>EvalCoord1</td>
<td>298–300</td>
</tr>
<tr>
<td>EvalCoordId</td>
<td>299</td>
</tr>
<tr>
<td>EvalCoordIff</td>
<td>299</td>
</tr>
<tr>
<td>EvalCoord2</td>
<td>298, 300, 301</td>
</tr>
<tr>
<td>EvalMesh*</td>
<td>406</td>
</tr>
<tr>
<td>EvalMesh1</td>
<td>62, 299</td>
</tr>
<tr>
<td>EvalMesh2</td>
<td>62, 299, 300</td>
</tr>
<tr>
<td>EvalPoint</td>
<td>24</td>
</tr>
<tr>
<td>EvalPoint*</td>
<td>406</td>
</tr>
<tr>
<td>EvalPoint1</td>
<td>300</td>
</tr>
<tr>
<td>EvalPoint2</td>
<td>300</td>
</tr>
<tr>
<td>EXP</td>
<td>229, 230, 350</td>
</tr>
<tr>
<td>EXP2</td>
<td>229</td>
</tr>
<tr>
<td>EXT_pixel_buffer_object</td>
<td>451</td>
</tr>
<tr>
<td>EXT_texture_sRGB</td>
<td>452</td>
</tr>
<tr>
<td>EXTENSIONS</td>
<td>137, 327, 328, 406, 459, 460</td>
</tr>
<tr>
<td>EYE_LINEAR</td>
<td>59, 60, 316, 359</td>
</tr>
<tr>
<td>EYE_PLANE</td>
<td>59</td>
</tr>
<tr>
<td>FASTEST</td>
<td>311, 312</td>
</tr>
<tr>
<td>FEEDBACK</td>
<td>302–304, 396</td>
</tr>
<tr>
<td>FEEDBACK_BUFFER_POINTER</td>
<td>327</td>
</tr>
<tr>
<td>FeedbackBuffer</td>
<td>303, 304, 309, 327, 406</td>
</tr>
<tr>
<td>FILL</td>
<td>132, 134, 135, 299, 354, 395, 409</td>
</tr>
<tr>
<td>Finish</td>
<td>310, 311, 394</td>
</tr>
<tr>
<td>FIXED_ONLY</td>
<td>85, 161, 269, 275, 276, 350</td>
</tr>
<tr>
<td>FLAT</td>
<td>85, 395</td>
</tr>
<tr>
<td>float</td>
<td>85</td>
</tr>
<tr>
<td>float</td>
<td>92</td>
</tr>
</tbody>
</table>

Version 3.0 (with changes marked - August 11, 2008)
INDEX

GetPixelMapuiv, 318
GetPixelMapusv, 318
GetPointerv, 327
GetPolygonStipple, 266, 323
GetProgramInfoLog, 91, 333
GetProgramiv, 90, 93, 96, 97, 102, 108, 332–334
GetQueryiv, 329
GetQueryObject[u]iv, 330
GetQueryObjectiv, 329
GetQueryObjectuiv, 329
GetRenderbufferParameteriv, 293, 316, 320
GetSeparableFilter, 266, 325
GetShaderInfoLog, 89, 333
GetShaderiv, 89, 332, 334
GetShaderSource, 334
GetString, 327, 328, 406
GetStringi, 328
GetTexImage, 215, 266, 321, 322, 324–326
GetTexLevelParameter, 316, 317
GetTexParameter, 293, 316, 317
GetTexParameterfv, 215, 217
GetTexParameterI, 316
GetTexParameterIiv, 317
GetTexParameterIuiv, 317
GetTexParameteriv, 215, 217
GetTexParameteriv, 316
GetTransformFeedbackVarying, 102, 103
GetUniform*, 336
GetUniformfv, 335
GetUniformiv, 336
GetUniformLocation, 96, 97, 100
GetUniformui, 336
GetVertexAttribdv, 334, 335
GetVertexAttribfv, 334, 335
GetVertexAttribI, 334, 335
GetVertexAttribIuiv, 334, 335
GetVertexAttribiv, 334, 335
GetVertexAttribPointer, 335
gl, 102, 235
GL_ARB_color_buffer_float, 466
GL_ARB_depth_texture, 431, 462
GL_ARB_draw_buffers, 443, 465
GL_ARB_fragment_program, 437, 463, 465
GL_ARB_fragment_program_shadow, 465
GL_ARB_fragment_shader, 436, 437, 442, 464
GL_ARB_half_float_pixel, 466
GL_ARB_matrix_palette, 462
GL_ARB_multisample, 423, 460
GL_ARB_multitexture, 424, 460
GL_ARB_occlusion_query, 437, 464
GL_ARB_pixel_buffer_object, 449, 466
GL_ARB_point_parameters, 432, 461
GL_ARB_point_sprite, 444, 465
GL_ARB_shader_objects, 436, 438, 442, 464
GL_ARB_shading_language_100, 443, 464
GL_ARB_shadow, 431, 462, 465
GL_ARB_shadow_ambient, 463
GL_ARB_texture_border_clamp, 425, 461
GL_ARB_texture_compression, 422, 461
GL_ARB_texture_cube_map, 423, 461
GL_ARB_texture_env_add, 424, 461
GL_ARB_texture_env_combine, 424, 462
GL_ARB_texture_env_crossbar, 433, 462
GL_ARB_texture_env_dot3, 424, 462
GL_ARB_texture_float, 466
GL_ARB_texture_mirrormed_repeat, 433, 462
GL_ARB_texture_non_power_of_two, 443, 464
GL_ARB_texture_rectangle, 465
GL_ARB_transpose_matrix, 425, 460
GL_ARB_vertex_blend, 461
GL_ARB_vertex_buffer_object, 436, 437, 463
GL_ARB_vertex_program, 430, 433, 463

Version 3.0 (with changes marked - August 11, 2008)
INDEX

GL_ARB_vertex_shader, 436, 438, 442, 464
GL_ARB_window_pos, 433, 463
GL_ATI_separate_stencil, 444
gl_BackColor, 78, 85, 107
gl_BackSecondaryColor, 78, 85, 107
gl_ClipDistance[], 68
gl_ClipVertex, 68, 107
gl_Colour, 233
GL_EXT_bgra, 413
GL_EXT_blend_color, 417
GL_EXT_blend_equation_separate, 444
GL_EXT_blend_func_separate, 432
GL_EXT_blend_logic_op, 409, 444
GL_EXT_blend_minmax, 417
GL_EXT_blend_subtract, 417
GL_EXT_color_subtable, 416
GL_EXT_color_table, 416
GL_EXT_convolution, 416
GL_EXT_copy_texture, 410
GL_EXT_draw_range_elements, 415
GL_EXT_fog_coord, 431
GL_EXT_histogram, 417
GL_EXT_multi_draw_arrays, 431
GL_EXT_packed_pixels, 414
GL_EXT_polygon_offset, 409
GL_EXT_rescale_normal, 414
GL_EXT_secondary_color, 432
GL_EXT_separate_specular_color, 414
GL_EXT_shadow_funcs, 437
GL_EXT_stencil_two_side, 444
GL_EXT_stencil_wrap, 432
GL_EXT_subtexture, 410
GL_EXT_texture, 409, 410
GL_EXT_texture3D, 413
GL_EXT_texture_lod_bias, 433
GL_EXT_texture_object, 410
GL_EXT_texture_sRGB, 449
GL_EXT_vertex_array, 408
gl_FogFragCoord, 70, 107
gl_FragColor, 234, 256
gl_FragCoord, 233
gl_FragCoord.z, 393
gl_FragData, 234, 256
gl_FragData[n], 234
gl_FragDepth, 234, 235, 393
gl_FrontColor, 78, 85, 107
gl_FrontFacing, 233
gl_FrontSecondaryColor, 78, 85, 107
GL_HP_convolution_border_modes, 416
GL_NV_blend_square, 431
GL_PERSPECTIVE_CORRECT_HINT, 449
gl_PointSize, 107, 117
gl_Position, 101, 107
gl_PrimitiveID, 233, 234
gl_SecondaryColor, 233
GL_SGI_color_matrix, 416
GL_SGIS_generate_mipmap, 430
GL_SGIS_multitexture, 421
GL_SGIS_texture_edge_clamp, 415
GL_SGIS_texture_lod, 415
gl_TexCoord, 107
gl_VertexID, 106, 107, 234
gl_hyperpage, 101
GLX_ARB_fboconfig_float, 466
GREATER, 201, 224, 242–244
GREEN_BIAS, 163
GREEN_BITS, 293, 406
GREEN_INTEGER, 152
GREEN_SCALE, 163
HALF_FLOAT, 29, 151, 268, 269, 271
Hint, 311, 406
HINT_BIT, 338
HISTOGRAM, 146, 147, 171, 325, 326
Histogram, 146, 147, 171, 310
HISTOGRAM_ALPHA_SIZE, 326
HISTOGRAM_BLUE_SIZE, 326
HISTOGRAM_FORMAT, 326
HISTOGRAM_GREEN_SIZE, 326
HISTOGRAM_LUMINANCE_SIZE, 326
HISTOGRAM_RED_SIZE, 326
HISTOGRAM_SINK, 326
HISTOGRAM_WIDTH, 326
INCR, 243

Version 3.0 (with changes marked - August 11, 2008)
INDEX

INCR_WRAP, 243, 432
INDEX, 319, 388
Index, 24, 26
Index*, 403
Index[type]v, 32
INDEX_ARRAY, 31, 36
INDEX_ARRAY_POINTER, 327
INDEX_LOGIC_OP, 251
INDEX_OFFSET, 138, 163, 370
INDEX_SHIFT, 138, 163, 370

IndexMask, 257, 259
IndexPointer, 24, 29, 30, 309, 403
INFO_LOG_LENGTH, 332, 334
InitNames, 301, 406

INT, 29, 93, 97, 102, 151, 270, 271, 308, 317, 319
INT_SAMPLER_1D, 97
INT_SAMPLER_1D_ARRAY, 97
INT_SAMPLER_2D, 97
INT_SAMPLER_2D_ARRAY, 97
INT_SAMPLER_3D, 97
INT_SAMPLER_CUBE, 97
INT_VEC2, 93, 97, 102
INT_VEC3, 93, 97, 102
INT_VEC4, 93, 97, 102

INTENSITY, 148, 165, 166, 177, 183, 185, 200, 201, 219, 220, 224, 322, 358, 371, 405, 409
INTENSITY12, 183
INTENSITY16, 183
INTENSITY4, 183
INTENSITY8, 183
INTERLEAVED_ATTRIBS, 65, 101, 102, 333
INTERLEAVED_ATTRIBS, 378
InterleavedArrays, 24, 36, 37, 309
INTERPOLATE, 221

INVALID_FRAMEBUFFER_OPERATION, 17, 23, 142, 146, 150, 173, 195, 267, 272, 275, 293
INVALID_OPERATION, 17, 24, 39, 42,
INDEX

LIGHT_MODEL_AMBIENT, 81
LIGHT_MODEL_COLOR_CONTROL, 81
LIGHT_MODEL_LOCAL_VIEWER, 81
LIGHT_MODEL_TWO_SIDE, 81
LIGHTING, 75, 404
LIGHTING_BIT, 338
LightModel, 79, 81
LightModel*, 404
LINE, 132, 134, 299, 300, 354, 409
LINE_BIT, 338
LINE_LOOP, 20, 64
LINE_RESET_TOKEN, 306
LINE_SMOOTH, 123, 128
LINE_SMOOTH_HINT, 311
LINE_STIPPLE, 125, 404
LINE_TOKEN, 306
LINEAR, 104, 201, 206–208, 210–212, 214, 229, 274, 275, 287, 319
LINEAR_ATTENUATION, 81
LINEAR_MIPMAP_LINEAR, 201, 208, 210, 288
LINEAR_MIPMAP_NEAREST, 201, 208, 210, 288
LINES, 20, 64, 126
LineStipple, 125, 404
LineWidth, 123, 404
LINK_STATUS, 90, 332
LinkProgram, 66, 90, 91, 93, 94, 96, 100–102, 235, 310
LIST_BIT, 338
ListBase, 308, 310, 406
LOAD, 262, 263
LoadIdentity, 53, 403
LoadMatrix, 52, 53, 403
LoadMatrix[fd], 52
LoadName, 301, 302, 406
LoadTransposeMatrix, 52, 403
LoadTransposeMatrix[fd], 52
LOGIC_OP, 251, 444, 446
LogicOp, 251, 252
LOWER_LEFT, 118, 121
LUMINANCE, 152, 160, 165, 166, 177, 183, 185, 200, 201, 214, 219, 220, 224, 267, 269, 270, 321, 322, 357, 358, 371, 373, 398, 405, 409
LUMINANCE12, 183
LUMINANCE12_ALPHA12, 183
LUMINANCE12_ALPHA4, 183
LUMINANCE16, 183
LUMINANCE16_ALPHA16, 183
LUMINANCE4, 183
LUMINANCE4_ALPHA4, 183
LUMINANCE6_ALPHA2, 183
LUMINANCE8, 183
LUMINANCE8_ALPHA8, 183
LUMINANCE_ALPHA, 152, 160, 165, 166, 177, 183, 185, 219, 220, 267, 269, 270, 321, 322, 405
MAJOR_VERSION, 328
Map*, 406
Map1, 295–297, 314
MAP_COLOR, 296
MAP_INDEX, 296
MAP_NORMAL, 296
MAP_TEXTURE_COORD_1, 296, 298
MAP_TEXTURE_COORD_2, 296, 298
MAP_TEXTURE_COORD_3, 296
MAP_TEXTURE_COORD_4, 296
MAP_VERTEX_3, 296
MAP_VERTEX_4, 296
MAP_COLOR, 138, 163, 164
MAP_FLUSH_EXPLICIT_BIT, 44, 45
MAP_INVALIDATE_BUFFER_BIT, 44, 45
MAP_INVALIDATE_RANGE_BIT, 44, 45
MAP_READ_BIT, 43–45
MAP_STENCIL, 138, 164
MAP_UNSYNCHRONIZED_BIT, 44, 45
MAP_VERTEX_3, 298
MAP_VERTEX_4, 298

Version 3.0 (with changes marked - August 11, 2008)
INDEX

MAP_WRITE_BIT, 44, 45
Map{12}, 297
MapBuffer, 41, 42, 65, 66, 309
MapBufferRange, 43, 44
MapGrid*, 406
MapGrid1, 299
MapGrid2, 299
mat2, 92
mat2x3, 92
mat2x4, 92
mat3, 92
mat3x2, 92
mat3x4, 92
mat4, 92
mat4x2, 92
mat4x3, 92
Material, 24, 79–81, 84, 395
Material*, 403
MATRIX_MODE, 55
MatrixMode, 51, 403
MAX, 246, 247
MAX_3D_TEXTURE_SIZE, 187, 285
MAX_ARRAY_TEXTURE LAYERS, 187
MAX_ATTRIB_STACK_DEPTH, 337, 406
MAX_CLIENT_ATTRIB_STACK_DEPTH, 337, 406
MAX_CLIP_DISTANCES, 456
MAX_CLIP_PLANES, 456
MAX_COLOR_ATTACHMENT S, 254, 255, 257, 277, 284, 292
MAX_COLOR_MATRIX_STACK_DEPTH, 323
MAX_COMBINED_TEXTURE_IMAGE_UNITS, 56, 105, 315
MAX_CONVOLUTION_HEIGHT, 144, 325
MAX_CONVOLUTION_WIDTH, 144, 325
MAX_CUBE_MAP_TEXTURE_SIZE, 187, 285
MAX_DRAW_BUFFERS, 235, 246, 248, 256, 258
MAX_ELEMENTS_INDICES, 35
MAX_ELEMENTS_VERTICES, 35
MAX_EVAL_ORDER, 296, 297
MAX_FRAGMENT_UNIFORM_COMPONENTS, 231
MAX_PIXEL_MAP_TABLE, 139, 163
MAX_PROGRAM_TEXEL_OFFSET, 204
MAX_RENDERBUFFER_SIZE, 281
MAX_SAMPLES, 281, 282
MAX_TEXTURECOORDS, 25, 28, 38, 55, 56, 315, 445
MAX_TEXTURE_IMAGE_UNITS, 105, 233, 445
MAX_TEXTURE_LOD_BIAS, 203
MAX_TEXTURE_SIZE, 187, 285
MAX_TEXTURE_UNITS, 17, 56, 226, 337, 445
MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS, 102
MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS, 65, 66, 101, 331
MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS, 102
MAX_VARYING_COMPONENTS, 100, 101, 456
MAX_VARYING_FLOATS, 456
MAX_VERTEX_ATTRIBS, 26–29, 31, 38, 92, 94, 335
MAX_VERTEX_TEXTURE_IMAGE_UNITS, 105
MAX_VERTEX_UNIFORM_COMPONENTS, 95
MAX_VIEWPORT_DIMS, 329
MIN_PROGRAM_TEXEL_OFFSET, 204
MINMAX, 148, 171, 326, 327
Minmax, 148, 172
MINMAX_FORMAT, 327
MINMAX_SINK, 327
MINOR_VERSION, 328
MIRRORED_REPEAT, 201, 206, 433
MODELVIEW, 51, 56, 57
MODELVIEW_MATRIX, 315

Version 3.0 (with changes marked - August 11, 2008)
INDEX

MODULATE, 218, 220, 221, 223, 359
MULT, 262, 263
MultiDrawArrays, 34, 46, 107, 431
MultiDrawElements, 35, 46, 47, 107, 431
MULTISAMPLE, 116, 122, 128, 134, 172, 173, 240, 252, 253
MULTISAMPLE_BIT, 338
MultiTexCoord, 24, 25, 31
MultiTexCoord[size][type]v, 32
MultMatrix, 52, 53, 403
MultMatrix[fd], 53
MultTransposeMatrix, 52, 403
MultTransposeMatrix[fd], 53
N3F_V3F, 36, 37
NAND, 252
NEAREST, 104, 201, 205, 206, 208, 210–212, 224, 274, 287
NEAREST_MIPMAP_LINEAR, 201, 208, 210, 212, 214, 287
NEAREST_MIPMAP_NEAREST, 201, 208, 210, 212, 224, 287
NEVER, 201, 224, 242–244
NewList, 307, 308, 406
NICEST, 311, 312
NO_ERROR, 15
NOOP, 252
noperspective, 87
NOR, 252
Normal, 24, 25, 92
Normal3, 12, 25
Normal3*, 403
Normal3[type]v, 32
Normal3d, 12
Normal3dv, 13
Normal3f, 12
Normal3fv, 13
NORMAL_ARRAY, 31, 36, 38
NORMAL_ARRAY_BUFFER_BINDING, 46
NORMAL_ARRAY_POINTER, 327
NORMAL_MAP, 59, 60, 423
NORMALIZE, 57, 403
NormalPointer, 24, 28–30, 36, 46, 309, 403
NOTEQUAL, 201, 224, 242–244
NULL, 38, 39, 41–44, 47, 89, 93, 97, 102, 331, 333, 334, 340, 345, 348
NUM_COMPRESSED_TEXTURE_FORMATS, 178
NUM_EXTENSIONS, 328
OBJECT_LINEAR, 59, 60, 316
OBJECT_PLANE, 59
ONE, 248, 249, 362
ONE_MINUS_CONSTANT_ALPHA, 249, 431
ONE_MINUS_CONSTANT_COLOR, 249, 431
ONE_MINUS_DST_ALPHA, 249
ONE_MINUS_DST_COLOR, 249, 431
ONE_MINUS_SRC_ALPHA, 222, 249
ONE_MINUS_SRC_COLOR, 222, 249, 430
OPERANDn_ALPHA, 218, 221, 222, 226, 450
OPERANDn_RGB, 218, 221, 222, 226, 450
OR, 252
OR_INVERTED, 252
OR_REVERSE, 252
ORDER, 318
Ortho, 53, 54, 395, 403
OUT_OF_MEMORY, 16, 17, 41, 42, 281, 307
PACK_ALIGNMENT, 266, 369
PACK_IMAGE_HEIGHT, 266, 322, 369
PACK_LSB_FIRST, 266, 369
PACK_ROW_LENGTH, 266, 369
PACK_SKIP_IMAGES, 266, 322, 369
PACK_SKIP_PIXELS, 266, 369
PACK_SKIP_ROWS, 266, 369
PACK_SWAP_BYTES, 266, 369
PASS_THROUGH_TOKEN, 306

Version 3.0 (with changes marked - August 11, 2008)
PassThrough, 305, 406
PERSPECTIVE_CORRECTION_HINT, 311, 406
PIXEL_MAP_A_TO_A, 139, 163
PIXEL_MAP_B_TO_B, 139, 163
PIXEL_MAP_G_TO_G, 139, 163
PIXEL_MAP_I_TO_A, 139, 164
PIXEL_MAP_I_TO_B, 139, 164
PIXEL_MAP_I_TO_G, 139, 164
PIXEL_MAP_I_TO_I, 139, 164
PIXEL_MAP_I_TO_R, 139, 164
PIXEL_MAP_S_TO_S, 139, 164
PIXEL_MODE_BIT, 338
PIXEL_PACK_BUFFER, 38, 40, 42, 43, 45, 136, 264, 330, 331, 449
PIXEL_PACK_BUFFER_BINDING, 269, 318, 321
PIXEL_UNPACK_BUFFER, 38, 40, 42, 43, 45, 136, 264, 330, 331, 449
PIXEL_UNPACK_BUFFER_BINDING, 139, 150, 196
PixelMap, 135, 138, 139, 276, 451
PixelStore, 24, 135, 136, 138, 266, 276, 309
PixelTransfer, 135, 137, 138, 169, 276
PixelZoom, 162, 172, 405
POINT, 132, 134, 299, 300, 354, 404
POINT_BIT, 338
POINT_DISTANCE_ATTENUATION, 117
POINT_FADE_THRESHOLD_SIZE, 117
POINT_SIZE_MAX, 117
POINT_SIZE_MIN, 117
POINT_SMOOTH, 117, 122, 404
POINT_SMOOTH_HINT, 311
POINT_SPRITE, 117, 118, 122, 218, 316, 404, 451
POINT_SPRITE_COORD_ORIGIN, 118, 121, 444, 445
POINT_TOKEN, 306
PointParameter, 117, 432
PointParameter*, 118
POINTS, 20, 64, 299
PointSize, 116
POLYGON, 20, 23, 64, 234, 404
POLYGON_BIT, 338
POLYGON_OFFSET_FILL, 134
POLYGON_OFFSET_LINE, 134
POLYGON_OFFSET_POINT, 134
POLYGON_SMOOTH, 129, 134
POLYGON_SMOOTH_HINT, 311
POLYGON_STIPPLE, 131, 404
POLYGON_STIPPLE_BIT, 338
POLYGON_TOKEN, 306
PolygonMode, 128, 132, 134, 135, 302, 304, 404
PolygonOffset, 133
PolygonStipple, 131, 136, 404
PopAttrib, 336, 337, 339, 396, 406, 450
PopClientAttrib, 24, 309, 336, 337, 339, 406
PopMatrix, 56, 403
PopName, 301, 406
POSITION, 81, 316
POST_COLOR_MATRIX_x_BIAS, 138
POST_COLOR_MATRIX_x_SCALE, 138
POST_COLOR_MATRIX_ALPHA_BIAS, 170
POST_COLOR_MATRIX_ALPHA_SCALE, 170
POST_COLOR_MATRIX_BLUE_BIAS, 170
POST_COLOR_MATRIX_BLUE_SCALE, 170
POST_COLOR_MATRIX_COLOR_TABLE, 140, 171
POST_COLOR_MATRIX_GREEN_BIAS, 170
POST_COLOR_MATRIX_GREEN_SCALE, 170
POST_COLOR_MATRIX_RED_BIAS, 170
POST_COLOR_MATRIX_RED_SCALE, 170
POST_CONVOLUTION_x_BIAS, 138
POST_CONVOLUTION_x_SCALE, 138
INDEX

POST_CONVOLUTION_ALPHA_BIAS, 169
POST_CONVOLUTION_ALPHA_SCALE, 169
POST_CONVOLUTION_BLUE_BIAS, 169
POST_CONVOLUTION_BLUE_SCALE, 169
POST_CONVOLUTION_COLOR_TABLE, 140, 170
POST_CONVOLUTION_GREEN_BIAS, 169
POST_CONVOLUTION_GREEN_SCALE, 169
POST_CONVOLUTION_RED_BIAS, 169
POST_CONVOLUTION_RED_SCALE, 169
PREVIOUS, 222, 223, 360
PRIMARY_COLOR, 222
PRIMITIVES_GENERATED, 67, 329
PrioritizeTextures, 217, 218, 405
PROJECTION, 51, 56, 57
PROXY_COLOR_TABLE, 140, 143, 310
PROXY_HISTOGRAM, 146, 147, 310, 326
PROXY_POST_COLOR_MATRIX_COLOR_TABLE, 140, 310
PROXY_POST_CONVOLUTION_COLOR_TABLE, 140, 310
PROXY_TEXTURE_1D, 178, 188, 215, 310, 317
PROXY_TEXTURE_1D_ARRAY, 178, 187, 215, 310, 317
PROXY_TEXTURE_2D, 178, 187, 215, 310, 317
PROXY_TEXTURE_2D_ARRAY, 176, 178, 215, 310, 317
PROXY_TEXTURE_3D, 176, 214, 215, 310, 317
PROXY_TEXTURE_CUBE_MAP, 178, 187, 215, 310, 317
PushAttrib, 336, 337, 339, 406, 450
PushClientAttrib, 24, 309, 336, 337, 339, 406
PushMatrix, 56, 403
PushName, 301, 406
Q, 59, 60, 316
QUAD_STRIP, 22, 64, 233, 404
QUADRATIC_ATTENUATION, 81
QUADS, 23, 64, 233, 404
QUERY_BY_REGION_NO_WAIT, 63
QUERY_BY_REGION_WAIT, 63
QUERY_COUNTER_BITS, 329
QUERY_NO_WAIT, 63
QUERY_RESULT, 329, 451
QUERY_RESULT_AVAILABLE, 330, 451
QUERY_WAIT, 63
R, 59, 60, 316, 455
R11F_G11F_B10F, 179, 180, 182
R16, 179, 181
R16F, 179, 182
R16I, 179, 182
R16UI, 179, 182
R32F, 179, 182
R32I, 179, 182
R32UI, 179, 182
R3_G3_B2, 181
R8, 179, 181
R8I, 179, 182
R8UI, 179, 182
RASTERIZER_DISCARD, 113
RasterPosition, 70, 108, 293, 302, 395, 433
RasterPosition*, 404
RasterPosition2, 70
RasterPosition3, 70
RasterPosition4, 70
READ_BUFFER, 267, 290, 294
READ_FRAMEBUFFER, 263, 277–279, 283, 284, 292, 318, 364
READ_FRAMEBUFFER_BINDING, 142, 146, 195, 264, 267, 268, 272, 275, 279
READONLY, 39, 42
READ_WRITE, 39, 41, 42, 348
ReadBuffer, 255, 266, 267, 276
INDEX

ReadPixels, 66, 135, 137, 151, 152, 154, 165, 263–267, 269, 270, 272, 274, 293, 309, 321, 323, 324, 405, 449
Rect, 48, 49, 129
Rect*, 404
RED_BIAS, 163
RED_BITS, 293, 406
RED_INTEGER, 152
RED_SCALE, 163
REDUCE, 167, 169, 372
REFLECTION_MAP, 59, 60, 423
RENDER, 302, 303, 389
RENDERBUFFER, 280, 281, 283, 293, 319, 320, 367
RENDERBUFFER_ALPHA_SIZE, 320
RENDERBUFFER_BINDING, 281
RENDERBUFFER_BLUE_SIZE, 320
RENDERBUFFER_DEPTH_SIZE, 320
RENDERBUFFER_GREEN_SIZE, 320
RENDERBUFFER_HEIGHT, 282, 320
RENDERBUFFER_INTERNAL_FORMAT, 282, 320
RENDERBUFFER_RED_SIZE, 320
RENDERBUFFER_SAMPLES, 282, 290–292, 320
RENDERBUFFER_STENCIL_SIZE, 320
RENDERBUFFER_WIDTH, 282, 320
RenderbufferStorage, 282, 291, 309
RenderbufferStorageMultisample, 281, 282, 309
RENDERER, 327
RenderMode, 302–304, 309, 406
REPEAT, 201, 206, 214, 357
REPLACE, 218, 220, 221, 243
REPLICATE_BORDER, 167, 168
RESCALE_NORMAL, 57, 403
ResetHistogram, 326
ResetMinmax, 326
RETURN, 262, 263
RG, 140, 152, 177, 181, 182, 185, 219, 220, 267, 270, 288, 321, 322, 455
RG16, 179, 181
RG16F, 179, 182
RG16I, 179, 182
RG16UI, 179, 182
RG32F, 179, 182
RG32I, 179, 182
RG32UI, 179, 182
RG8, 179, 181
RG8I, 179, 182
RG8UI, 179, 182
RG_INTEGER, 152
RGB, 152, 155, 165, 166, 177, 181, 182, 185, 219–221, 223, 249, 267, 269, 270, 288, 321, 322, 409
RGB10, 181
RGB10_A2, 179, 181
RGB12, 181
RGB16, 179, 181
RGB16F, 179, 182
RGB16I, 179, 182
RGB16UI, 179, 182
RGB32F, 179, 182
RGB32I, 179, 182
RGB32UI, 179, 182
RGB4, 181
RGB5, 181
RGB5_A1, 181
RGB8, 179, 181
RGB8I, 179, 182
RGB8UI, 179, 182
RGB9_E5, 140, 179, 180, 182, 225, 269
RGB_INTEGER, 152
RGB_SCALE, 218, 221, 450
RGB_A12, 182
RGB_A16, 179, 182
RGB_A16F, 179, 182
RGB_A16I, 179, 183

Version 3.0 (with changes marked - August 11, 2008)
INDEX

RGBA16UI, 179, 183
RGBA2, 181
RGBA32F, 179, 182
RGBA32I, 179, 183
RGBA32UI, 179, 183
RGBA4, 181
RGBA8, 179, 181
RGBA8I, 179, 182
RGBA8UI, 179, 182
RGBA_INTEGER, 152
RIGHT, 246, 254, 255, 257, 258, 267
Rotate, 53, 395, 403
S, 59, 60, 316
SAMPLE_ALPHA_TO_COVERAGE, 240
SAMPLE_ALPHA_TO_ONE, 240, 241
SAMPLE_BUFFERS, 115, 122, 128, 134, 172, 173, 240, 244, 252, 253, 259, 264, 266, 272, 275, 276, 292
SAMPLE_COVERAGE, 240, 241
SAMPLE_COVERAGE_INVERT, 240, 241
SAMPLE_COVERAGE_VALUE, 240, 241
SampleCoverage, 241
sampler2D, 100
SAMPLER_1D, 97
SAMPLER_1D_ARRAY, 97
SAMPLER_1D_ARRAY_SHADOW, 97
SAMPLER_1D_SHADOW, 97
SAMPLER_2D, 97
SAMPLER_2D_ARRAY, 97
SAMPLER_2D_ARRAY_SHADOW, 97
SAMPLER_2D_SHADOW, 97
SAMPLER_3D, 97
SAMPLER_CUBE, 97
SAMPLER_CUBE_SHADOW, 97
SAMPLES, 115, 116, 245, 276, 292
SAMPLES_PASSED, 62, 63, 244, 329
Scale, 53, 54, 395, 403
Scissor, 240
SCISSOR_BIT, 338
SCISSOR_TEST, 240
SECONDARY_COLOR_ARRAY, 31, 36
SECONDARY_COLOR_ARRAY_POINTER, 327
SecondaryColor, 24, 26, 432
SecondaryColor3, 26, 444
SecondaryColor3*, 403
SecondaryColor3[type]v, 32
SecondaryColorPointer, 24, 29, 30, 309, 403
SELECT, 302, 303, 396
SelectBuffer, 302, 303, 309, 327, 406
SEPARABLE_2D, 145, 165, 188, 325
SeparableFilter2D, 136, 144
SEPARATE_ATTRIBS, 65, 101, 102, 333
SEPARATE_SPECULAR_COLOR, 77
SET, 252
ShadeModel, 85, 404
SHADER_SOURCE_LENGTH, 332, 334
SHADER_TYPE, 109, 332
ShaderSource, 88, 89, 310, 334
SHADING_LANGUAGE_VERSION, 327, 443, 448
SHININESS, 81
SHORT, 29, 151, 270, 271, 308
SINGLE_COLOR, 76, 77, 351
SLUMINANCE, 183, 225
SLUMINANCE8, 225
SLUMINANCE8_ALPHA8, 225
SLUMINANCE_ALPHA, 225
SLUMINANCE_ALPHA8, 183
SMOOTH, 85, 350
SOURCE0_ALPHA, 438
SOURCE0_RGB, 438
SOURCE1_ALPHA, 438
SOURCE1_RGB, 438
SOURCE2_ALPHA, 438
SOURCE2_RGB, 438
SPECULAR, 81, 83
SPHERE_MAP, 59, 60, 423
SPOT_CUTOFF, 81
SPOT_DIRECTION, 81, 316

Version 3.0 (with changes marked - August 11, 2008)
INDEX

SPOT_EXPONENT, 81
SRC0_ALPHA, 438
SRC0_RGB, 438
SRC1_ALPHA, 438
SRC1_RGB, 438
SRC2_ALPHA, 438
SRC2_RGB, 438
SRC_ALPHA, 222, 223, 249, 360
SRC_ALPHA_SATURATE, 249
SRC_COLOR, 222, 223, 249, 360, 430
SRCn_ALPHA, 218, 221, 222, 226, 450
SRCn_RGB, 218, 221, 222, 226, 450
SRGB, 225, 246, 247, 250, 319
SRGB8, 179, 182, 225
SRGB8_ALPHA8, 179, 182, 225
SRGB_ALPHA, 225
STACK_OVERFLOW, 17, 56, 302, 337
STACK_UNDERFLOW, 17, 56, 302, 337
STATIC_COPY, 39, 40
STATIC_DRAW, 39, 40, 348
STATIC_READ, 39, 40
STENCIL, 272, 358, 366
STENCIL_ATTACHMENT, 261, 278, 284, 290
STENCIL_BITS, 274, 293, 406
STENCIL_BUFFER, 261, 318
STENCIL_BUFFER_BIT, 259, 262, 274, 275, 338
STENCIL_INDEX, 140, 143, 148, 150, 152, 162, 176, 263, 264, 266, 268, 272, 274, 282, 289, 322
STENCIL_INDEX1, 282
STENCIL_INDEX6, 282
STENCIL_INDEX4, 282
STENCIL_INDEX8, 282
STENCIL_TEST, 242
StencilFunc, 242, 243, 394
StencilFuncSeparate, 242, 243
StencilMask, 258, 259, 264, 394
StencilMaskSeparate, 258, 259, 264
StencilOp, 242, 243
StencilOpSeparate, 242, 243
STREAM_COPY, 39, 40
STREAM_DRAW, 39, 40
STREAM_READ, 39, 40
SUBTRACT, 221
T, 59, 316
T2F_C3F_V3F, 36, 37
T2F_C4F_N3F_V3F, 36, 37
T2F_C4UB_V3F, 36, 37
T2F_N3F_V3F, 36, 37
T2F_V3F, 36, 37
T4F_C4F_N3F_V4F, 36, 37
T4F_V4F, 36, 37
TABLE_TOO_LARGE, 17, 140, 147
TexCoord, 24, 25
TexCoord*, 403
TexCoord1, 25
TexCoord2, 25
TexCoord3, 25
TexCoord4, 25
TexCoordPointer, 24, 29–31, 36, 309, 403
TexEnv, 55, 56, 218, 226, 405, 450
TexEnv*, 118
TexGen, 55, 59, 60, 315
TexGen*, 403
TexImage, 56, 193
TexImage*, 405
TexImage1D, 136, 165, 167, 184, 188, 189, 191–193, 196, 209, 215, 310
TexImage2D, 136, 165, 167, 184, 187–189, 191, 193, 196, 209, 215, 310, 449
TexImage3D, 136, 176, 184, 186, 187, 189, 193, 196, 209, 214, 215, 310, 322
TexParameter, 56, 200, 405
TexParameter*, 405
TexParameter[if], 203, 209
TexParameterf, 217
TexParameterfv, 217
TexParameterI, 200
TexParameterI*, 217
TexParameterIv, 200
TexParameterIuv, 200
TexParameteriv, 200, 217

Version 3.0 (with changes marked - August 11, 2008)
INDEX

TEXTURE_GEN_MODE, 59, 60
TEXTURE_GEN_Q, 60
TEXTURE_GEN_R, 60
TEXTURE_GEN_S, 60
TEXTURE_GEN_T, 60
TEXTURE_GREEN_SIZE, 317
TEXTURE_GREEN_TYPE, 317
TEXTURE_HEIGHT, 195, 197–199, 318
TEXTURE_INTENSITY_SIZE, 317
TEXTURE_INTENSITY_TYPE, 317
TEXTURE_INTERNAL_FORMAT, 197–199, 318
TEXTURE_LOD_BIAS, 201, 203, 218, 433
TEXTURE_LUMINANCE_SIZE, 317
TEXTURE_LUMINANCE_TYPE, 317
TEXTURE_MAG_FILTER, 201, 211, 212, 214, 224
TEXTURE_MAX_LEVEL, 200, 201, 209, 214, 288
TEXTURE_MAX_LOD, 200, 201, 203, 214, 451
TEXTURE_MIN_FILTER, 201, 205, 207, 208, 211–214, 224, 287
TEXTURE_MIN_LOD, 200, 201, 203, 214, 451
TEXTURE_PRIORITY, 200, 201, 214, 218, 405
TEXTURE_RECTANGLE_ARB, 465
TEXTURE_RED_SIZE, 317
TEXTURE_RED_TYPE, 317
TEXTURE_RESIDENT, 214, 217, 317
TEXTURE_SHARED_SIZE, 317
TEXTURE_STENCIL_SIZE, 317
TEXTURE_WIDTH, 195, 197–199, 318
TEXTURE_WRAP_R, 201, 205, 206, 405
TEXTURE_WRAP_S, 201, 205, 206, 405
TEXTURE_WRAP_T, 201, 205, 206, 405
TEXTUREn, 222, 226
TRANSFORM_BIT, 338

TRANSFORM_FEEDBACK_BUFFER,
64, 66
TRANSFORM_FEEDBACK_BUFFER_BINDING,
331
TRANSFORM_FEEDBACK_BUFFER_MODE,
333
TRANSFORM_FEEDBACK_BUFFER_SIZE,
331
TRANSFORM_FEEDBACK_BUFFER_START,
331
TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN,
66, 67, 329
TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH,
102, 333
TRANSFORM_FEEDBACK_VARYINGS,
102, 333
TransformFeedbackVaryings, 65, 101, 102, 309
Translate, 53, 54, 395, 403
TRANSPOSE_COLOR_MATRIX, 315, 323
TRANSPOSE_MODELVIEW_MATRIX, 315
TRANSPOSE_PROJECTION_MATRIX, 315
TRANSPOSE_TEXTURE_MATRIX, 315
TRIANGLE_FAN, 22, 64
TRIANGLE_STRIP, 21, 64
TRIANGLES, 22, 23, 64, 65
ui}v, 261
Uniform, 98
Uniform*, 95, 99, 100
Uniform*f{v}, 98, 99
Uniform*i{v}, 98
Uniform*ui{v}, 98, 99
Uniform1i{v}, 98, 100
| Uniform1iv, 99             | UNSIGNED_INT_5_9_9_9_REV,                                  |
| Uniform2{if ui}*, 99      | 151, 155, 158, 159, 181, 269, 271                        |
| Uniform4i{v}, 99          | UNSIGNED_INT_8_8_8_8, 151, 155, 158, 271                  |
| Uniform{1,2,3,4}ui, 98    | UNSIGNED_INT_8_8_8_REV, 151, 155, 158, 271               |
| UniformMatrix*, 445       | UNSIGNED_INT_SAMPLER_1D, 97                              |
| UniformMatrix2x4fv, 98    | UNSIGNED_INT_SAMPLER_1D_ARRAY, 97                       |
| UniformMatrix3fv, 99      | UNSIGNED_INT_SAMPLER_2D, 97                               |
| UniformMatrix{234}fv, 98  | UNSIGNED_INT_SAMPLER_2D_ARRAY, 97                       |
| UniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}fv, 98, 448 | UNSIGNED_INT_SAMPLER_3D, 97 |
| UnmapBuffer, 43, 44, 309  | UNSIGNED_INT_SAMPLER_CUBE, 97                           |
| UNPACK_ALIGNMENT, 136, 153, 176, 369 | UNSIGNED_INT_VEC2, 93, 97, 102 |
| UNPACK_IMAGE_HEIGHT, 136, 176, 369 | UNSIGNED_INT_VEC3, 93, 97, 103 |
| UNPACK_LSB_FIRST, 136, 160, 369 | UNSIGNED_INT_VEC4, 93, 97, 103 |
| UNPACK_ROW_LENGTH, 136, 153, 176, 369 | UNSIGNED_NORMALIZED, 317, 319 |
| UNPACK_SKIP_IMAGES, 136, 176, 369 | UNSIGNED_SHORT, 29, 34, 151, 270, 271, 308, 318, 322 |
| UNPACK_SKIP_PIXELS, 136, 153, 176, 369 | UNSIGNED_SHORT_1_5_5_5_REV, 151, 155, 157, 271 |
| UNPACK_SKIP_ROWS, 136, 153, 176, 369 | UNSIGNED_SHORT_4_4_4_4, 151, 155, 157, 271 |
| UNPACK_SWAP_BYTES, 136, 153, 176, 369 | UNSIGNED_SHORT_4_4_4_4_REV, 151, 155, 157, 271 |
| UNSIGNED_BYTE, 29, 34, 37, 151, 156, 270, 271, 308, 398 | UNSIGNED_SHORT_5_5_5_1, 151, 155, 157, 271 |
| UNSIGNED_BYTE_2_3_3_REV, 151, 155, 156, 271 | UNSIGNED_SHORT_5_6_5, 151, 155, 157, 271 |
| UNSIGNED_BYTE_3_3_2, 151, 155, 156, 271 | UNSIGNED_SHORT_5_6_5_REV, 151, 155, 157, 271 |
| UNSIGNED_INT, 29, 34, 93, 97, 102, 151, 158, 270, 271, 308, 317–319, 322 | UPPER_LEFT, 118, 121, 353 |
| USE_PROGRAM, 66, 91, 103  | USE_PROGRAM, 66, 91, 103                                 |
| V2F, 36, 37               | V2F, 36, 37                                               |
| V3F, 36, 37               | V3F, 36, 37                                               |
| ValidateProgram, 108, 109, 310, 332 | vec2, 92 |
| vec3, 92                  | vec3, 92                                                  |
| vec4, 92, 99              | vec4, 92, 99                                              |
| VENDOR, 327               | VENDOR, 327                                               |
INDEX

VERSION, 327, 328
Vertex, 11, 24, 70, 92, 298
Vertex*, 403
Vertex2, 24, 27, 49
Vertex2sv, 11
Vertex3, 24, 27
Vertex3f, 11
Vertex4, 24, 27
Vertex[size][type]v, 33
VERTEX_ARRAY, 31, 38
VERTEX_ARRAY_BINDING, 315, 335
VERTEX_ARRAY_POINTER, 327
VERTEX_ATTRIB_ARRAY_BUFFER_BINDING, 335, 446
VERTEX_ATTRIB_ARRAY_ENABLED, 335
VERTEX_ATTRIB_ARRAY_INTEGER, 335
VERTEX_ATTRIB_ARRAY_NORMALIZED, 335
VERTEX_ATTRIB_ARRAY_POINTER, 335
VERTEX_ATTRIB_ARRAY_SIZE, 335
VERTEX_ATTRIB_ARRAY_STRIDE, 335
VERTEX_ATTRIB_ARRAY_TYPE, 335
VERTEX_PROGRAM_POINT_SIZE, 117
VERTEX_PROGRAM_TWO_SIDE, 78, 79, 404
VERTEX_SHADER, 88, 332
VertexAttrib, 24, 26, 63
VertexAttrib*, 27, 28, 92, 403, 450
VertexAttrib1*, 27
VertexAttrib2*, 27
VertexAttrib3*, 27
VertexAttrib4, 26
VertexAttrib4*, 27
VertexAttrib4N, 27
VertexAttrib4Nub, 27
VertexAttrib[size][type]v, 32
VertexAttrib[size]N[type]v, 32
VertexAttribI, 27
VertexAttrib4I, 27
VertexAttribI[size][type]v, 32
VertexAttribIPointer, 29, 30, 48, 309, 335
VertexAttribIPointer, 24, 29, 30, 48, 309, 335, 404
VertexPointer, 24, 28, 30, 38, 309, 403
Viewport, 51
VIEWPORT_BIT, 338
WGL_ARB_multisample, 423
WGL_ARB_pixel_format_float, 466
WindowPos, 71, 302, 433, 444
WindowPos*, 404
WindowPos2, 71
WindowPos3, 71
WRITE_ONLY, 39, 42
XOR, 252
ZERO, 243, 248, 249, 362

Version 3.0 (with changes marked - August 11, 2008