The OpenGL® Graphics System: A Specification (Version 3.2 (Compatibility Profile) - December 7, 2009)

Mark Segal
Kurt Akeley

Editor (version 1.1): Chris Frazier
Editor (versions 1.2-3.2): Jon Leech
Editor (version 2.0): Pat Brown

Copyright © 2006-2009 The Khronos Group Inc. All Rights Reserved.

This specification is protected by copyright laws and contains material proprietary to the Khronos Group, Inc. It or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of Khronos Group. You may use this specification for implementing the functionality therein, without altering or removing any trademark, copyright or other notice from the specification, but the receipt or possession of this specification does not convey any rights to reproduce, disclose, or distribute its contents, or to manufacture, use, or sell anything that it may describe, in whole or in part.

Khronos Group grants express permission to any current Promoter, Contributor or Adopter member of Khronos to copy and redistribute UNMODIFIED versions of this specification in any fashion, provided that NO CHARGE is made for the specification and the latest available update of the specification for any version of the API is used whenever possible. Such distributed specification may be reformatted AS LONG AS the contents of the specification are not changed in any way. The specification may be incorporated into a product that is sold as long as such product includes significant independent work developed by the seller. A link to the current version of this specification on the Khronos Group web-site should be included whenever possible with specification distributions.

Khronos Group makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this specification, including, without limitation, any implied warranties of merchantability or fitness for a particular purpose or non-infringement of any intellectual property. Khronos Group makes no, and expressly disclaims any, warranties, express or implied, regarding the correctness, accuracy, completeness, timeliness, and reliability of the specification. Under no circumstances will the Khronos Group, or any of its Promoters, Contributors or Members or their respective partners, officers, directors, employees, agents or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.

Khronos is a trademark of The Khronos Group Inc. OpenGL is a registered trademark, and OpenGL ES is a trademark, of Silicon Graphics International.

Contents

1 Introduction **1**
- 1.1 Formatting of Optional Features 1
- 1.2 What is the OpenGL Graphics System? 1
- 1.3 Programmer's View of OpenGL 2
- 1.4 Implementor's View of OpenGL 2
- 1.5 Our View 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 3

2 OpenGL Operation **5**
- 2.1 OpenGL Fundamentals 5
  - 2.1.1 Floating-Point Computation 7
  - 2.1.2 16-Bit Floating-Point Numbers 8
  - 2.1.3 Unsigned 11-Bit Floating-Point Numbers 9
  - 2.1.4 Unsigned 10-Bit Floating-Point Numbers 9
  - 2.1.5 Fixed-Point Data Conversions 10
- 2.2 GL State 12
  - 2.2.1 Shared Object State 13
- 2.3 GL Command Syntax 13
- 2.4 Basic GL Operation 15
- 2.5 GL Errors 17
- 2.6 Begin/End Paradigm 18
  - 2.6.1 Begin and End 22
  - 2.6.2 Polygon Edges 29
  - 2.6.3 GL Commands within **Begin/End** 29
- 2.7 Vertex Specification 30
- 2.8 Vertex Arrays 34

- 2.8.1 Drawing Commands 40
- 2.9 Buffer Objects 47
  - 2.9.1 Creating and Binding Buffer Objects 47
  - 2.9.2 Creating Buffer Object Data Stores 50
  - 2.9.3 Mapping and Unmapping Buffer Data 52
  - 2.9.4 Effects of Accessing Outside Buffer Bounds 56
  - 2.9.5 Copying Between Buffers 56
  - 2.9.6 Vertex Arrays in Buffer Objects 57
  - 2.9.7 Array Indices in Buffer Objects 58
  - 2.9.8 Buffer Object State 58
- 2.10 Vertex Array Objects 59
- 2.11 Rectangles 60
- 2.12 Fixed-Function Vertex Transformations 60
  - 2.12.1 Matrices 61
  - 2.12.2 Normal Transformation 66
  - 2.12.3 Generating Texture Coordinates 68
- 2.13 Fixed-Function Vertex Lighting and Coloring 70
  - 2.13.1 Lighting 72
  - 2.13.2 Lighting Parameter Specification 76
  - 2.13.3 **ColorMaterial** 77
  - 2.13.4 Lighting State 80
  - 2.13.5 Color Index Lighting 80
  - 2.13.6 Clamping or Masking 81
- 2.14 Vertex Shaders 82
  - 2.14.1 Shader Objects 82
  - 2.14.2 Program Objects 84
  - 2.14.3 Vertex Attributes 86
  - 2.14.4 Uniform Variables 89
  - 2.14.5 Samplers 106
  - 2.14.6 Varying Variables 106
  - 2.14.7 Shader Execution 109
  - 2.14.8 Required State 116
- 2.15 Geometry Shaders 118
  - 2.15.1 Geometry Shader Input Primitives 118
  - 2.15.2 Geometry Shader Output Primitives 120
  - 2.15.3 Geometry Shader Variables 120
  - 2.15.4 Geometry Shader Execution Environment 121
- 2.16 Coordinate Transformations 126
  - 2.16.1 Controlling the Viewport 126
- 2.17 Asynchronous Queries 127

2.18 Conditional Rendering . . . 129
2.19 Transform Feedback . . . 130
2.20 Primitive Queries . . . 133
2.21 Flatshading . . . 134
2.22 Primitive Clipping . . . 136
  2.22.1 Color and Associated Data Clipping . . . 138
2.23 Final Color Processing . . . 139
2.24 Current Raster Position . . . 140

3 Rasterization **144**
3.1 Discarding Primitives Before Rasterization . . . 146
3.2 Invariance . . . 146
3.3 Antialiasing . . . 146
  3.3.1 Multisampling . . . 148
3.4 Points . . . 149
  3.4.1 Basic Point Rasterization . . . 151
  3.4.2 Point Rasterization State . . . 155
  3.4.3 Point Multisample Rasterization . . . 155
3.5 Line Segments . . . 156
  3.5.1 Basic Line Segment Rasterization . . . 156
  3.5.2 Other Line Segment Features . . . 159
  3.5.3 Line Rasterization State . . . 162
  3.5.4 Line Multisample Rasterization . . . 162
3.6 Polygons . . . 162
  3.6.1 Basic Polygon Rasterization . . . 163
  3.6.2 Stippling . . . 165
  3.6.3 Antialiasing . . . 166
  3.6.4 Options Controlling Polygon Rasterization . . . 166
  3.6.5 Depth Offset . . . 167
  3.6.6 Polygon Multisample Rasterization . . . 168
  3.6.7 Polygon Rasterization State . . . 169
3.7 Pixel Rectangles . . . 169
  3.7.1 Pixel Storage Modes and Pixel Buffer Objects . . . 170
  3.7.2 The Imaging Subset . . . 171
  3.7.3 Pixel Transfer Modes . . . 172
  3.7.4 Transfer of Pixel Rectangles . . . 183
  3.7.5 Rasterization of Pixel Rectangles . . . 196
  3.7.6 Pixel Transfer Operations . . . 198
  3.7.7 Pixel Rectangle Multisample Rasterization . . . 207
3.8 Bitmaps . . . 208

3.9 Texturing 210
  3.9.1 Texture Image Specification 212
  3.9.2 Alternate Texture Image Specification Commands 227
  3.9.3 Compressed Texture Images 232
  3.9.4 Multisample Textures 236
  3.9.5 Buffer Textures 237
  3.9.6 Texture Parameters 239
  3.9.7 Depth Component Textures 242
  3.9.8 Cube Map Texture Selection 242
  3.9.9 Texture Minification 244
  3.9.10 Texture Magnification 253
  3.9.11 Combined Depth/Stencil Textures 254
  3.9.12 Texture Completeness 254
  3.9.13 Texture State and Proxy State 255
  3.9.14 Texture Objects 258
  3.9.15 Texture Environments and Texture Functions 260
  3.9.16 Texture Comparison Modes 266
  3.9.17 sRGB Texture Color Conversion 267
  3.9.18 Shared Exponent Texture Color Conversion 268
  3.9.19 Texture Application 268
3.10 Color Sum 269
3.11 Fog 271
3.12 Fragment Shaders 273
  3.12.1 Shader Variables 273
  3.12.2 Shader Execution 274
3.13 Antialiasing Application 279
3.14 Multisample Point Fade 279

4 Per-Fragment Operations and the Framebuffer 281
4.1 Per-Fragment Operations 283
  4.1.1 Pixel Ownership Test 283
  4.1.2 Scissor Test 284
  4.1.3 Multisample Fragment Operations 284
  4.1.4 Alpha Test 286
  4.1.5 Stencil Test 286
  4.1.6 Depth Buffer Test 288
  4.1.7 Occlusion Queries 289
  4.1.8 Blending 289
  4.1.9 sRGB Conversion 294
  4.1.10 Dithering 295

4.1.11 Logical Operation . . . 295
4.1.12 Additional Multisample Fragment Operations . . . 297
4.2 Whole Framebuffer Operations . . . 298
4.2.1 Selecting a Buffer for Writing . . . 298
4.2.2 Fine Control of Buffer Updates . . . 302
4.2.3 Clearing the Buffers . . . 303
4.2.4 The Accumulation Buffer . . . 307
4.3 Drawing, Reading, and Copying Pixels . . . 308
4.3.1 Writing to the Stencil or Depth/Stencil Buffers . . . 308
4.3.2 Reading Pixels . . . 308
4.3.3 Copying Pixels . . . 316
4.3.4 Pixel Draw/Read State . . . 321
4.4 Framebuffer Objects . . . 321
4.4.1 Binding and Managing Framebuffer Objects . . . 322
4.4.2 Attaching Images to Framebuffer Objects . . . 324
4.4.3 Feedback Loops Between Textures and the Framebuffer . 333
4.4.4 Framebuffer Completeness . . . 335
4.4.5 Effects of Framebuffer State on Framebuffer Dependent Values . . . 340
4.4.6 Mapping between Pixel and Element in Attached Image . 341
4.4.7 Layered Framebuffers . . . 342

5 Special Functions 344
5.1 Evaluators . . . 344
5.2 Selection . . . 350
5.3 Feedback . . . 352
5.4 Display Lists . . . 354
5.4.1 Commands Not Usable In Display Lists . . . 358
5.5 Flush and Finish . . . 360
5.6 Sync Objects and Fences . . . 360
5.6.1 Waiting for Sync Objects . . . 362
5.6.2 Signalling . . . 364
5.7 Hints . . . 364

6 State and State Requests 366
6.1 Querying GL State . . . 366
6.1.1 Simple Queries . . . 366
6.1.2 Data Conversions . . . 367
6.1.3 Enumerated Queries . . . 368
6.1.4 Texture Queries . . . 372

6.1.5 Stipple Query 374
6.1.6 Color Matrix Query 375
6.1.7 Color Table Query 375
6.1.8 Convolution Query 376
6.1.9 Histogram Query 376
6.1.10 Minmax Query 377
6.1.11 Pointer and String Queries 378
6.1.12 Asynchronous Queries 380
6.1.13 Sync Object Queries 381
6.1.14 Buffer Object Queries 382
6.1.15 Vertex Array Object Queries 384
6.1.16 Shader and Program Queries 384
6.1.17 Framebuffer Object Queries 389
6.1.18 Renderbuffer Object Queries 391
6.1.19 Saving and Restoring State 392
6.2 State Tables 395

A Invariance **454**
A.1 Repeatability 454
A.2 Multi-pass Algorithms 455
A.3 Invariance Rules 455
A.4 What All This Means 457

B Corollaries **458**

C Compressed Texture Image Formats **461**
C.1 RGTC Compressed Texture Image Formats 461
C.1.1 Format `COMPRESSED_RED_RGTC1` 462
C.1.2 Format `COMPRESSED_SIGNED_RED_RGTC1` 463
C.1.3 Format `COMPRESSED_RG_RGTC2` 464
C.1.4 Format `COMPRESSED_SIGNED_RG_RGTC2` 464

D Shared Objects and Multiple Contexts **465**
D.1 Object Deletion Behavior 465
D.1.1 Automatic Unbinding of Deleted Objects 465
D.1.2 Deleted Object and Object Name Lifetimes 466
D.2 Sync Objects and Multiple Contexts 466
D.3 Propagating State Changes 467
D.3.1 Determining Completion of Changes to an object 467
D.3.2 Definitions 468

D.3.3 Rules 468

E Profiles and the Deprecation Model 470
E.1 Core and Compatibility Profiles 471
E.2 Deprecated and Removed Features 471
E.2.1 Deprecated But Still Supported Features 471
E.2.2 Removed Features 472

F Version 3.0 and Before 477
F.1 New Features 477
F.2 Deprecation Model 478
F.3 Changed Tokens 479
F.4 Change Log 479
F.5 Credits and Acknowledgements 481

G Version 3.1 484
G.1 New Features 484
G.2 Deprecation Model 485
G.3 Change Log 485
G.4 Credits and Acknowledgements 486

H Version 3.2 489
H.1 New Features 489
H.2 Deprecation Model 490
H.3 Changed Tokens 490
H.4 Change Log 491
H.5 Credits and Acknowledgements 493

I Extension Registry, Header Files, and ARB Extensions 496
I.1 Extension Registry 496
I.2 Header Files 496
I.3 ARB Extensions 497
I.3.1 Naming Conventions 497
I.3.2 Promoting Extensions to Core Features 498
I.3.3 Multitexture 498
I.3.4 Transpose Matrix 498
I.3.5 Multisample 499
I.3.6 Texture Add Environment Mode 499
I.3.7 Cube Map Textures 499
I.3.8 Compressed Textures 499

I.3.9 Texture Border Clamp 499
I.3.10 Point Parameters 499
I.3.11 Vertex Blend 499
I.3.12 Matrix Palette 500
I.3.13 Texture Combine Environment Mode 500
I.3.14 Texture Crossbar Environment Mode 500
I.3.15 Texture Dot3 Environment Mode 500
I.3.16 Texture Mirrored Repeat 500
I.3.17 Depth Texture 500
I.3.18 Shadow 500
I.3.19 Shadow Ambient 500
I.3.20 Window Raster Position 501
I.3.21 Low-Level Vertex Programming 501
I.3.22 Low-Level Fragment Programming 501
I.3.23 Buffer Objects 501
I.3.24 Occlusion Queries 501
I.3.25 Shader Objects 501
I.3.26 High-Level Vertex Programming 502
I.3.27 High-Level Fragment Programming 502
I.3.28 OpenGL Shading Language 502
I.3.29 Non-Power-Of-Two Textures 502
I.3.30 Point Sprites 502
I.3.31 Fragment Program Shadow 502
I.3.32 Multiple Render Targets 502
I.3.33 Rectangular Textures 503
I.3.34 Floating-Point Color Buffers 503
I.3.35 Half-Precision Floating Point 503
I.3.36 Floating-Point Textures 504
I.3.37 Pixel Buffer Objects 504
I.3.38 Floating-Point Depth Buffers 504
I.3.39 Instanced Rendering 504
I.3.40 Framebuffer Objects 504
I.3.41 sRGB Framebuffers 504
I.3.42 Geometry Shaders 505
I.3.43 Half-Precision Vertex Data 505
I.3.44 Instanced Rendering 505
I.3.45 Flexible Buffer Mapping 505
I.3.46 Texture Buffer Objects 505
I.3.47 RGTC Texture Compression Formats 506
I.3.48 One- and Two-Component Texture Formats 506

I.3.49 Vertex Array Objects 506
I.3.50 Versioned Context Creation 506
I.3.51 Uniform Buffer Objects 506
I.3.52 Restoration of features removed from OpenGL 3.0 506
I.3.53 Fast Buffer-to-Buffer Copies 507
I.3.54 Shader Texture Level of Detail Control 507
I.3.55 Depth Clamp Control 507
I.3.56 Base Vertex Offset Drawing Commands 507
I.3.57 Fragment Coordinate Convention Control 508
I.3.58 Provoking Vertex Control 508
I.3.59 Seamless Cube Maps 508
I.3.60 Fence Sync Objects 508
I.3.61 Multisample Textures 508
I.3.62 BGRA Attribute Component Ordering 508
I.3.63 Per-Buffer Blend Control 508
I.3.64 Sample Shading Control 509
I.3.65 Cube Map Array Textures 509
I.3.66 Texture Gather 509
I.3.67 Texture Level-Of-Detail Queries 509
I.3.68 Profiled Context Creation 509

List of Figures

2.1 Block diagram of the GL. 15
2.2 Creation of a processed vertex from a transformed vertex and current values. 19
2.3 Primitive assembly and processing. 21
2.4 Triangle strips, fans, and independent triangles. 23
2.5 Quadrilateral strips and independent quadrilaterals. 24
2.6 Lines with adjacency. 25
2.7 Triangles with adjacency. 25
2.8 Triangle strips with adjacency. 27
2.9 Vertex transformation sequence. 60
2.10 Processing of RGBA colors. 70
2.11 Processing of color indices. 70
2.12 ColorMaterial operation. 77
2.13 Current raster position. 141

3.1 Rasterization. 144
3.2 Rasterization of non-antialiased wide points. 152
3.3 Rasterization of antialiased wide points. 152
3.4 Visualization of Bresenham's algorithm. 157
3.5 Rasterization of non-antialiased wide lines. 160
3.6 The region used in rasterizing an antialiased line segment. 161
3.7 Transfer of pixel rectangles. 183
3.8 Selecting a subimage from an image 188
3.9 A bitmap and its associated parameters. 209
3.10 A texture image and the coordinates used to access it. 226
3.11 Multitexture pipeline. 269

4.1 Per-fragment operations. 283
4.2 Operation of **ReadPixels**. 308

4.3 Operation of **CopyPixels**. 316

5.1 Map Evaluation. 346
5.2 Feedback syntax. 355

List of Tables

2.1 GL command suffixes 14
2.2 GL data types 16
2.3 Summary of GL errors 19
2.4 Triangles generated by triangle strips with adjacency. 28
2.5 Vertex array sizes (values per vertex) and data types 36
2.6 Variables that direct the execution of **InterleavedArrays**. 45
2.7 Buffer object binding targets. 48
2.8 Buffer object parameters and their values. 49
2.9 Buffer object initial state. 51
2.10 Buffer object state set by **MapBufferRange**. 54
2.11 Summary of lighting parameters. 73
2.12 Correspondence of lighting parameter symbols to names. 78
2.13 OpenGL Shading Language type tokens 98
2.14 Transform feedback modes 131
2.15 Provoking vertex selection. 135

3.1 **PixelStore** parameters. 170
3.2 **PixelTransfer** parameters. 172
3.3 **PixelMap** parameters. 173
3.4 Color table names. 174
3.5 Pixel data types. 185
3.6 Pixel data formats. 186
3.7 Swap Bytes bit ordering. 187
3.8 Packed pixel formats. 189
3.9 `UNSIGNED_BYTE` formats. Bit numbers are indicated for each component. 190
3.10 `UNSIGNED_SHORT` formats 191
3.11 `UNSIGNED_INT` formats 192
3.12 `FLOAT_UNSIGNED_INT` formats 193

3.13 Packed pixel field assignments. 194
3.14 Color table lookup. 201
3.15 Computation of filtered color components. 202
3.16 Conversion from RGBA, depth, and stencil pixel components to internal texture, table, or filter components. 214
3.17 Sized internal color formats. 219
3.18 Sized internal luminance and intensity formats. 220
3.19 Sized internal depth and stencil formats. 221
3.20 Generic and specific compressed internal formats. 221
3.21 Internal formats for buffer textures 240
3.22 Texture parameters and their values. 241
3.23 Selection of cube map images. 243
3.24 Texel location wrap mode application. 248
3.25 Correspondence of filtered texture components to texture source components. 262
3.26 Texture functions `REPLACE`, `MODULATE`, and `DECAL` 262
3.27 Texture functions `BLEND` and `ADD`. 263
3.28 `COMBINE` texture functions. 264
3.29 Arguments for `COMBINE_RGB` functions. 265
3.30 Arguments for `COMBINE_ALPHA` functions. 265
3.31 Depth texture comparison functions. 267

4.1 RGB and Alpha blend equations. 292
4.2 Blending functions. 293
4.3 Arguments to **LogicOp** and their corresponding operations. 296
4.4 Buffer selection for the default framebuffer 299
4.5 Buffer selection for a framebuffer object 299
4.6 **DrawBuffers** buffer selection for the default framebuffer 300
4.7 **PixelStore** parameters. 310
4.8 **ReadPixels** index masks. 314
4.9 **ReadPixels** GL data types and reversed component conversion formulas. 315
4.10 Effective **ReadPixels** format for `DEPTH_STENCIL` **CopyPixels** operation. 318
4.11 Correspondence of renderbuffer sized to base internal formats. . . 327
4.12 Framebuffer attachment points. 329
4.13 Layer numbers for cube map texture faces. 343

5.1 Values specified by the *target* to **Map1**. 345
5.2 Correspondence of feedback type to number of values per vertex. . 354

5.3 Initial properties of a sync object created with **FenceSync**. 361
5.4 Hint targets and descriptions 365

6.1 Texture, table, and filter return values. 374
6.2 Context profile bits 379
6.3 Attribute groups 394
6.4 State Variable Types 396
6.5 GL Internal begin-end state variables (inaccessible) 397
6.6 Current Values and Associated Data 398
6.7 Vertex Array Object State 399
6.8 Vertex Array Object State (cont.) 400
6.9 Vertex Array Object State (cont.) 401
6.10 Vertex Array Object State (cont.) 402
6.11 Vertex Array Data (not in Vertex Array objects) 403
6.12 Buffer Object State 404
6.13 Transformation state 405
6.14 Coloring 406
6.15 Lighting (see also table 2.11 for defaults) 407
6.16 Lighting (cont.) 408
6.17 Rasterization 409
6.18 Rasterization (cont.) 410
6.19 Multisampling 411
6.20 Textures (state per texture unit and binding point) 412
6.21 Textures (state per texture unit and binding point)(cont.) 413
6.22 Textures (state per texture object) 414
6.23 Textures (state per texture image) 415
6.24 Texture Environment and Generation 416
6.25 Texture Environment and Generation (cont.) 417
6.26 Pixel Operations 418
6.27 Pixel Operations (cont.) 419
6.28 Framebuffer Control 420
6.29 Framebuffer (state per target binding point) 421
6.30 Framebuffer (state per framebuffer object) 422
6.31 Framebuffer (state per attachment point) 423
6.32 Renderbuffer (state per target and binding point) 424
6.33 Renderbuffer (state per renderbuffer object) 425
6.34 Pixels 426
6.35 Pixels (cont.) 427
6.36 Pixels (cont.) 428
6.37 Pixels (cont.) 429

6.38 Pixels (cont.) 430
6.39 Pixels (cont.) 431
6.40 Evaluators (**GetMap** takes a map name) 432
6.41 Shader Object State 433
6.42 Program Object State 434
6.43 Program Object State (cont.) 435
6.44 Program Object State (cont.) 436
6.45 Program Object State (cont.) 437
6.46 Vertex and Geometry Shader State 438
6.47 Query Object State 439
6.48 Transform Feedback State 440
6.49 Sync (state per sync object) 441
6.50 Hints 442
6.51 Implementation Dependent Values 443
6.52 Implementation Dependent Values (cont.) 444
6.53 Implementation Dependent Values (cont.) 445
6.54 Implementation Dependent Version and Extension Support 446
6.55 Implementation Dependent Vertex Shader Limits 447
6.56 Implementation Dependent Geometry Shader Limits 448
6.57 Implementation Dependent Fragment Processing Limits 449
6.58 Implementation Dependent Aggregate Shader Limits † The minimum value for each stage is `MAX_`*`stage`*`_UNIFORM_BLOCKS` × `MAX_`*`stage`*`_UNIFORM_BLOCK_SIZE` + `MAX_`*`stage`*`_UNIFORM_COMPONENTS` 450
6.59 Implementation Dependent Values (cont.) 451
6.60 Framebuffer Dependent Values 452
6.61 Miscellaneous 453

F.1 New token names 479

H.1 New token names 491

Chapter 1

Introduction

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

1.1 Formatting of 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.

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

1.4 Implementor's View of OpenGL

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

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

1.5 Our View

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

1.6 The Deprecation Model

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

1.7 Companion Documents

1.7.1 OpenGL Shading Language

This specification should be read together with a companion document titled *The OpenGL Shading Language*. The latter document (referred to as the OpenGL Shading Language Specification hereafter) defines the syntax and semantics of the programming language used to write vertex and fragment shaders (see sections 2.14 and 3.12). These sections may include references to concepts and terms (such as shading language variable types) defined in the companion document.

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

1.7.2 Window System Bindings

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

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

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

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

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

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

Chapter 2

OpenGL Operation

2.1 OpenGL Fundamentals

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

The GL draws *primitives* subject to a number of selectable modes and shader programs. Each primitive is a point, line segment, 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 such as positional coordinates, colors, normals, texture coordinates, etc. are associated with a vertex and each vertex is processed independently, in order, and in the same way. The only exception to this rule is if the group of vertices must be *clipped* so that the indicated primitive fits within a specified region; in this case vertex data may be modified and new vertices created. The type of clipping depends on which primitive the group of vertices represents.

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

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

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

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

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

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

Allocation and configuration of the default framebuffer occurs outside of the GL in conjunction with the window system, using companion APIs described in section 1.7.2.

Allocation and initialization of GL contexts is also done using these companion APIs. GL contexts can typically be associated with different default framebuffers, and some context state is determined at the time this association is performed.

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

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

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

2.1.1 Floating-Point Computation

The GL must perform a number of floating-point operations during the course of its operation. In some cases, the representation and/or precision of such operations is defined or limited; by the OpenGL Shading Language Specification for operations in shaders, and in some cases implicitly limited by the specified format of vertex, texture, or renderbuffer data consumed by the GL. Otherwise, the representation of such floating-point numbers, and the details of how operations on them are performed, is not specified. We require simply that numbers' floating-point parts contain enough bits and that their exponent fields are large enough so that individual results of floating-point operations are accurate to about 1 part in 10^5. The maximum representable magnitude of a floating-point number used to represent positional, normal, or texture coordinates must be at least 2^{32}; the maximum representable magnitude for colors must be at least 2^{10}. The maximum representable magnitude for all other floating-point values must be at least 2^{32}. $x \cdot 0 = 0 \cdot x = 0$ 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 Inf and $-Inf$ encode values with magnitudes too large to be represented; the special value NaN encodes "Not A Number" values resulting from undefined arithmetic operations such as $\frac{1}{0}$. Implementations are permitted, but not required, to support Infs and 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:

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

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

$$\begin{aligned} S &= \left\lfloor \frac{N \mod 65536}{32768} \right\rfloor \\ E &= \left\lfloor \frac{N \mod 32768}{1024} \right\rfloor \\ M &= N \mod 1024. \end{aligned}$$

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

2.1.3 Unsigned 11-Bit Floating-Point Numbers

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

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

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

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

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

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

2.1.4 Unsigned 10-Bit Floating-Point Numbers

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

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

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.

2.1.5 Fixed-Point Data Conversions

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

In the remainder of this section, b denotes the bit width of the fixed-point integer representation. When the integer is one of the types defined in table 2.2, b is the minimum required bit width of that type. When the integer is a texture or renderbuffer color or depth component (see section 3.9.1), b is the number of bits allocated to that component in the internal format of the texture or renderbuffer.

When the integer is a framebuffer color or depth component (see section 4), b is the number of bits allocated to that component in the framebuffer. For framebuffer and renderbuffer A components, b must be at least 2 if the buffer does not contain an A component, or if there is only 1 bit of A in the buffer.

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

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

Conversion from Normalized Fixed-Point to Floating-Point

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

$$f = \frac{c}{2^b - 1}. \qquad (2.1)$$

Signed normalized fixed-point integers represent numbers in the range $[-1, 1]$. The conversion from a signed normalized fixed-point value c to the corresponding floating-point value f may be performed in two ways:

$$f = \frac{2c + 1}{2^b - 1} \qquad (2.2)$$

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

Alternatively, conversion may be performed using

$$f = max\left\{\frac{c}{2^{b-1} - 1}, -1.0\right\}. \qquad (2.3)$$

In this case only the range $[-2^{b-1} + 1, 2^{b-1} - 1]$ is used to represent signed fixed-point values in the range $[-1, 1]$. For example, if $b = 8$, then the integer value -127 corresponds to -1.0 and the value 127 corresponds to 1.0. Note that while zero can be exactly expressed in this representation, one value (-128 in the example) is outside the representable range, and must be clamped before use. In general, this representation is used for signed normalized fixed-point texture or framebuffer values.

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

Conversion from Floating-Point to Normalized Fixed-Point

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

$$f' = f \times (2^b - 1). \tag{2.4}$$

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

The conversion from a floating-point value f to the corresponding signed normalized fixed-point value c may be performed in two ways, both beginning by clamping f to the range $[-1, 1]$:

$$\frac{f' = f \times (2^b - 1) - 1}{2} \tag{2.5}$$

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

Alternatively, conversion may be performed using

$$f' = f \times (2^{b-1} - 1). \tag{2.6}$$

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

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

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

2.2 GL State

The GL maintains considerable state. This document enumerates each state variable and describes how each variable can be changed. For purposes of discussion, state variables are categorized somewhat arbitrarily by their function. Although we describe the operations that the GL performs on the framebuffer, the framebuffer is not a part of GL state.

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

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

2.2.1 Shared Object State

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

2.3 GL Command Syntax

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

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

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

and

Type Descriptor	Corresponding `GL` Type
b	`byte`
s	`short`
i	`int`
i64	`int64`
f	`float`
d	`double`
ub	`ubyte`
us	`ushort`
ui	`uint`

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

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

These examples show the ANSI `C` declarations for these commands. In general, a command declaration has the form[1]

rtype **Name**$\{\epsilon$**1234**$\}\{\epsilon$ **b s i i64 f d ub us ui**$\}\{\epsilon$**v**$\}$
(*[args ,] T arg1 , . . . , T argN [, args]*) ;

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

For example,

`void` **Uniform**$\{$**1234**$\}\{$**if**$\}$(`int` *location*, `T` *value*);

indicates the eight declarations

`void` **Uniform1i**(`int` *location*, `int` *value*);

---

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

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

Arguments whose type is fixed (i.e. not indicated by a suffix on the command) are of one of the GL data types summarized in table 2.2, or pointers to one of these types.

2.4 Basic GL Operation

Figure 2.1 shows a schematic diagram of the GL. Commands enter the GL on the left. Some commands specify geometric objects to be drawn while others control how the objects are handled by the various stages. 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. It operates on geometric primitives described by vertices: points, line segments, and polygons. In this stage vertices are transformed and lit, followed by assembly into geometric primitives, which may optionally be used by the next stage, geometry shading, to generate new primitives. The final resulting primitives are clipped to a viewing volume in preparation for the next stage, rasterization. The rasterizer produces a series of framebuffer addresses and values using a two-dimensional description of a point, line segment, or polygon. Each *fragment* so produced is fed to the next stage that performs operations on individual fragments before they finally alter the framebuffer. These operations include conditional updates into the framebuffer based on incoming and previously stored depth values (to effect depth buffering), blending of incoming fragment colors with stored colors, as well as masking and other logical operations on fragment values.

Finally, 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

GL Type	Minimum Bit Width	Description
`boolean`	1	Boolean
`byte`	8	Signed twos complement binary integer
`ubyte`	8	Unsigned binary integer
`char`	8	Characters making up strings
`short`	16	Signed twos complement binary integer
`ushort`	16	Unsigned binary integer
`int`	32	Signed twos complement binary integer
`uint`	32	Unsigned binary integer
`int64`	64	Signed twos complement binary integer
`uint64`	64	Unsigned binary integer
`sizei`	32	Non-negative binary integer size
`enum`	32	Enumerated binary integer value
`intptr`	*ptrbits*	Signed twos complement binary integer
`sizeiptr`	*ptrbits*	Non-negative binary integer size
`sync`	*ptrbits*	Sync object handle (see section 5.6)
`bitfield`	32	Bit field
`half`	16	Half-precision floating-point value encoded in an unsigned scalar
`float`	32	Floating-point value
`clampf`	32	Floating-point value clamped to $[0, 1]$
`double`	64	Floating-point value
`clampd`	64	Floating-point value clamped to $[0, 1]$

Table 2.2: GL data types. GL types are not C types. Thus, for example, GL type `int` is referred to as `GLint` outside this document, and is not necessarily equivalent to the C type `int`. An implementation may use more bits than the number indicated in the table to represent a GL type. Correct interpretation of integer values outside the minimum range is not required, however.
ptrbits is the number of bits required to represent a pointer type; in other words, types `intptr`, `sizeiptr`, and `sync` must be sufficiently large as to store any address.

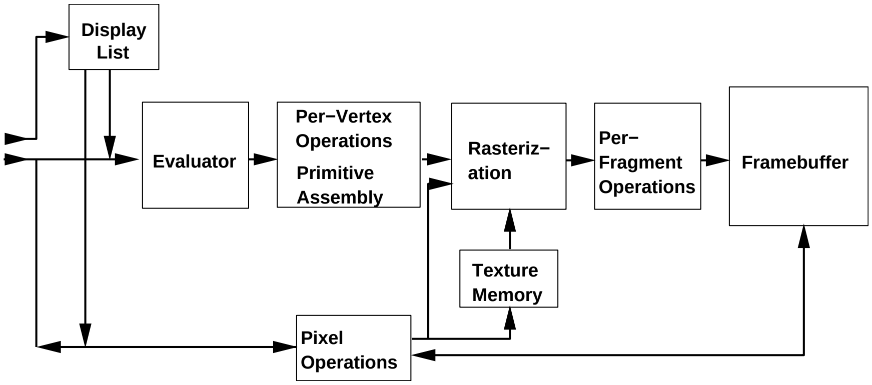


Figure 2.1. Block diagram of the GL.

back from the framebuffer or copied from one portion of the framebuffer to another. These transfers may include some type of decoding or encoding.

This ordering is meant only as a tool for describing the GL, not as a strict rule of how the GL is implemented, and we present it only as a means to organize the various operations of the GL. Objects such as curved surfaces, for instance, may be transformed before they are converted to polygons.

2.5 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

`enum` **GetError**(`void`);

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

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

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

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

- If a command that requires an enumerated value is passed a symbolic constant that is not one of those specified as allowable for that command, the error `INVALID_ENUM` is generated. This is the case even if the argument is a pointer to a symbolic constant, if the value pointed to is not allowable for the given command.
- If a negative number is provided where an argument of type `sizei` or `sizeiptr` is specified, the error `INVALID_VALUE` is generated.
- 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. Points, lines, polygons, and a variety of related geometric objects (see section 2.6.1) can be drawn in this way.

Each vertex is specified with two, three, or four coordinates. In addition, a *current normal*, multiple *current texture coordinate sets*, multiple *current generic*

Error	Description	Offending command ignored?
`INVALID_ENUM`	`enum` argument out of range	Yes
`INVALID_VALUE`	Numeric argument out of range	Yes
`INVALID_OPERATION`	Operation illegal in current state	Yes
`INVALID_FRAMEBUFFER_OPERATION`	Framebuffer object is not complete	Yes
`STACK_OVERFLOW`	Command would cause a stack overflow	Yes
`STACK_UNDERFLOW`	Command would cause a stack underflow	Yes
`OUT_OF_MEMORY`	Not enough memory left to execute command	Unknown
`TABLE_TOO_LARGE`	The specified table is too large	Yes

Table 2.3: Summary of GL errors

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 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 2.14) 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

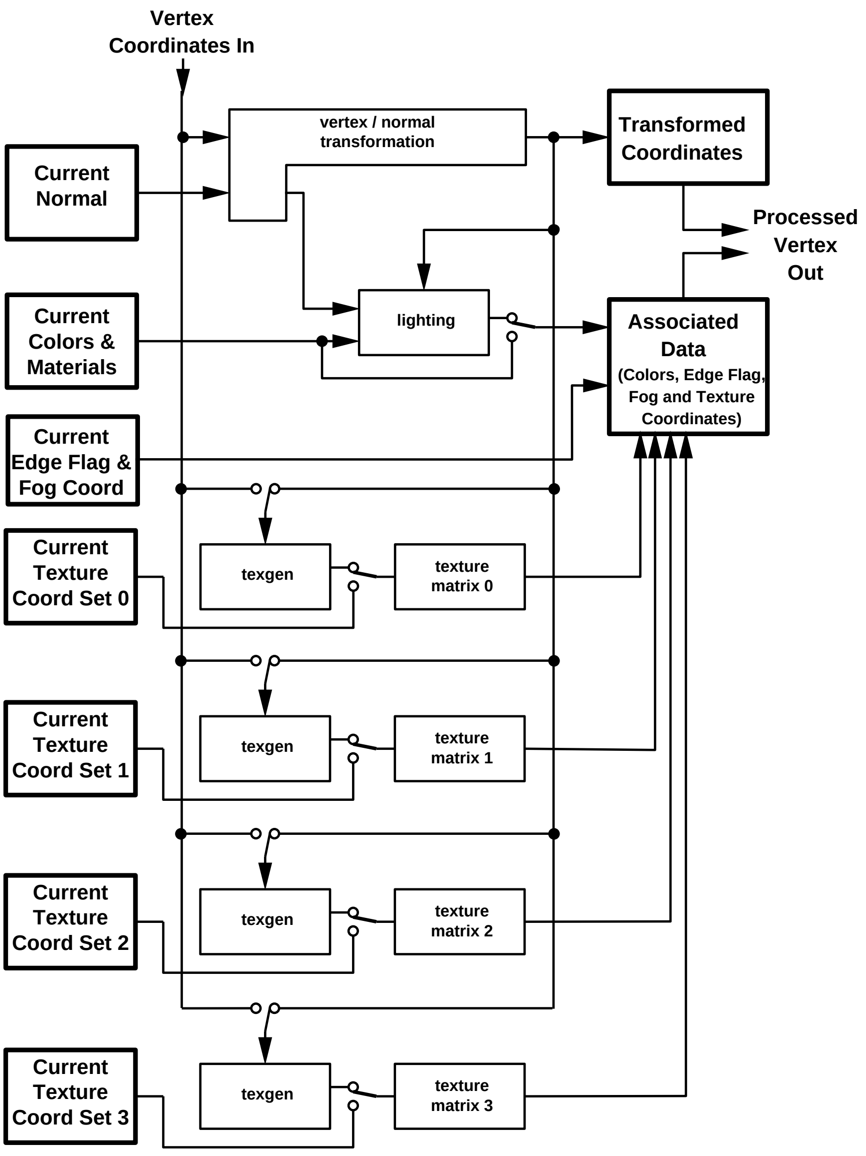


Figure 2.2. Association of current values with a vertex. The heavy lined boxes represent GL state. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation.

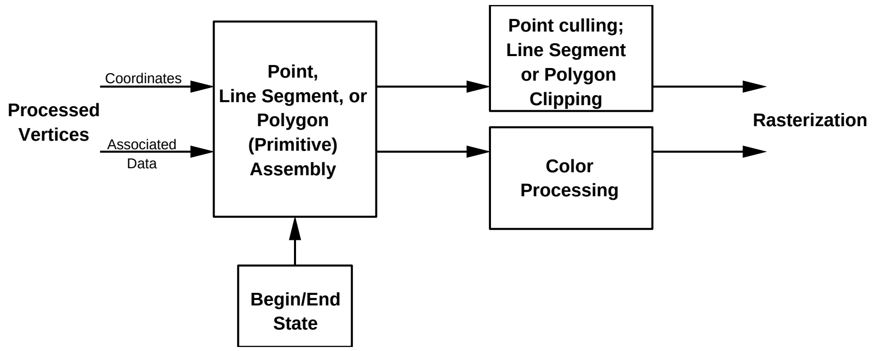

Processed Vertices
Coordinates
Associated Data
Point, Line Segment, or Polygon (Primitive) Assembly
Begin/End State
Point culling; Line Segment or Polygon Clipping
Color Processing
Rasterization

Figure 2.3. Primitive assembly and processing.

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, 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 2.13.2), 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 *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.

2.6.1 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

> `void` **Begin**(`enum` *mode*);
> `void` **End**(`void`);

There is no limit on the number of vertices that may be specified between a **Begin** and an **End**. The *mode* parameter of **Begin** determines the type of primitives to be drawn using the vertices. The types, and the corresponding *mode* parameters, are:

Points

A series of individual points may be specified with *mode* `POINTS`. Each vertex defines a separate point. 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 may be specified with *mode* `LINE_STRIP`. In this case, the first vertex specifies the first segment's start point while the second vertex specifies the first segment's endpoint and the second segment's start point. In general, the ith vertex (for $i > 1$) specifies the beginning of the ith segment and the end of the $i - 1$st. The last vertex specifies the end of the last segment. If only one vertex is specified, then no primitive is generated.

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

Line Loops

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

Separate Lines

Individual line segments, each specified by a pair of vertices, may be specified with *mode* `LINES`. The first two vertices 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 line strips but it is used differently: a processed vertex holding the first vertex of the current segment, and a boolean flag indicating whether the current vertex is odd or even (a segment start or end).

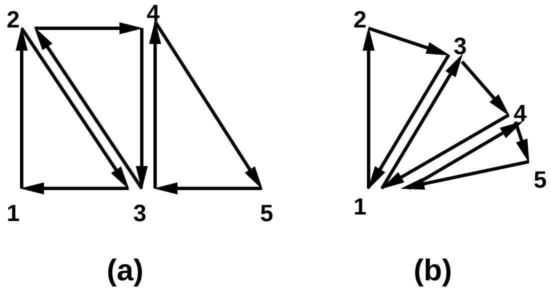


Figure 2.4. (a) A triangle strip. (b) A triangle fan. (c) Independent triangles. The numbers give the sequencing of the vertices in order within the vertex arrays. Note that in (a) and (b) triangle edge ordering is determined by the first triangle, while in (c) the order of each triangle's edges is independent of the other triangles.

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. A polygon described with fewer than three vertices does not generate a primitive.

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.

Triangle Strips

A triangle strip is a series of triangles connected along shared edges, and may be specified with *mode* `TRIANGLE_STRIP`. In this case, the first three vertices define the first triangle (and their order is significant, just as for polygons). Each subsequent vertex defines a new triangle using that point along with two vertices from the previous triangle. If fewer than three vertices are specified, no primitive is produced. See figure 2.4.

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

Triangle Fans

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

Separate Triangles

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

Quadrilateral (quad) strips

Quad strips generate a series of edge-sharing quadrilaterals from vertices appearing between **Begin** and **End**, when **Begin** is called with `QUAD_STRIP`. If the m vertices between the **Begin** and **End** are $v_1,\ldots,v_m$, where v_j is the jth 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.

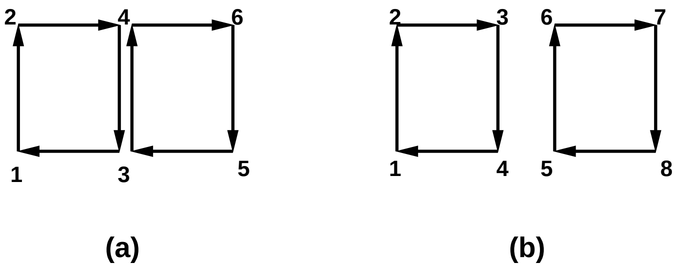
A quad strip with fewer than four vertices generates no primitive. If the number of vertices specified for a quadrilateral strip between **Begin** and **End** is odd, the final vertex is ignored.

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 $0\le k\le 3$; if k is not zero, the final k vertices are ignored. Separate quads are generated by calling **Begin** with the argument value `QUADS`.

Lines with Adjacency

Lines with adjacency are independent line segments where each endpoint has a corresponding *adjacent* vertex that can be accessed by a geometry shader (sec-



(a) (b)

Figure 2.5. (a) A quad strip. (b) Independent quads. The numbers give the sequencing of the vertices between **Begin** and **End**.

tion 2.15). If a geometry shader is not active, the adjacent vertices are ignored. They are generated with *mode* `LINES_ADJACENCY`.

A line segment is drawn from the $4i + 2$nd vertex to the $4i + 3$rd vertex for each $i = 0, 1, \ldots, n - 1$, where there are $4n + k$ vertices between a **Begin** and **End** pair. k is either 0, 1, 2, or 3; if k is not zero, the final k vertices are ignored. For line segment i, the $4i + 1$st and $4i + 4$th vertices are considered adjacent to the $4i + 2$nd and $4i + 3$rd vertices, respectively (see figure 2.6).

Line Strips with Adjacency

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

A line segment is drawn from the $i + 2$nd vertex to the $i + 3$rd vertex for each $i = 0, 1, \ldots, n - 1$, where there are $n + 3$ vertices between a **Begin** and **End** pair. If there are fewer than four vertices, all vertices are ignored. For line segment i, the $i + 1$st and $i + 4$th vertex are considered adjacent to the $i + 2$nd and $i + 3$rd vertices, respectively (see figure 2.6).

Triangles with Adjacency

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

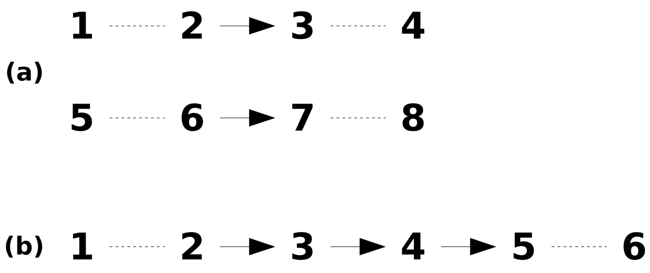


Figure 2.6. Lines with adjacency (a) and line strips with adjacency (b). The vertices connected with solid lines belong to the main primitives; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

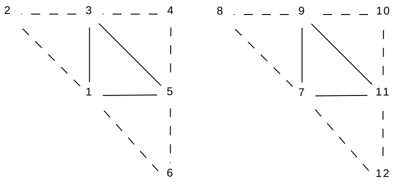


Figure 2.7. Triangles with adjacency. The vertices connected with solid lines belong to the main primitive; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

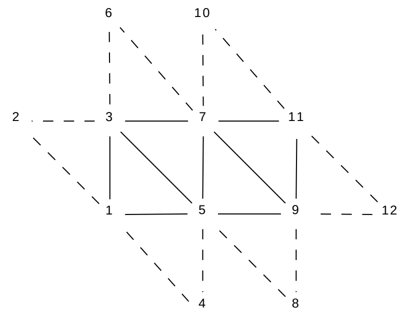


Figure 2.8. Triangle strips with adjacency. The vertices connected with solid lines belong to the main primitives; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

The $6i+1$st, $6i+3$rd, and $6i+5$th vertices (in that order) determine a triangle for each $i = 0, 1, \ldots, n-1$, where there are $6n+k$ vertices between a **Begin** and **End** pair. k is either 0, 1, 2, 3, 4, or 5; if k is non-zero, the final k vertices are ignored. For triangle i, the $i+2$nd, $i+4$th, and $i+6$th vertices are considered adjacent to edges from the $i+1$st to the $i+3$rd, from the $i+3$rd to the $i+5$th, and from the $i+5$th to the $i+1$st vertices, respectively (see figure 2.7).

Triangle Strips with Adjacency

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

In triangle strips with adjacency, n triangles are drawn where there are $2(n+2)+k$ vertices between a **Begin** and **End** pair. k is either 0 or 1; if k is 1, the final vertex is ignored. If there are fewer than 6 vertices, the entire primitive is ignored. Table 2.4 describes the vertices and order used to draw each triangle, and which vertices are considered adjacent to each edge of the triangle (see figure 2.8).

Depending on the current state of the GL, a *polygon primitive* generated from a drawing command with *mode* `POLYGON`, `QUADS`, `QUAD_STRIP`, `TRIANGLE_FAN`, `TRIANGLE_STRIP`, `TRIANGLES`, `TRIANGLES_ADJACENCY`, or `TRIANGLE_STRIP_ADJACENCY` may be rendered in one of several ways, such as outlining its border or filling its interior. The order of vertices in such a prim-

	Primitive Vertices			Adjacent Vertices		
Primitive	1st	2nd	3rd	1/2	2/3	3/1
only ($i = 0$, $n = 1$)	1	3	5	2	6	4
first ($i = 0$)	1	3	5	2	7	4
middle (i odd)	$2i + 3$	$2i + 1$	$2i + 5$	$2i - 1$	$2i + 4$	$2i + 7$
middle (i even)	$2i + 1$	$2i + 3$	$2i + 5$	$2i - 1$	$2i + 7$	$2i + 4$
last ($i = n - 1$, i odd)	$2i + 3$	$2i + 1$	$2i + 5$	$2i - 1$	$2i + 4$	$2i + 6$
last ($i = n - 1$, i even)	$2i + 1$	$2i + 3$	$2i + 5$	$2i - 1$	$2i + 6$	$2i + 4$

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

itive is significant in lighting, polygon rasterization, and fragment shading (see sections 2.13.1, 3.6.1, and 3.12.2). 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 for **Begin** and **End** consists of an fifteen-valued integer indicating either one of the fourteen possible **Begin/End** *mode*s, 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 polygon primitive generated 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

`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 2.13.2), 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:

`void` **Vertex**{**234**}{**sifd**}(`T` *coords*);
`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 2.5. A current value may be changed at any time by issuing an appropriate command. The commands

`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 **TexCoord1** 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

> `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 `TEXTURE`i, indicating that texture coordinate set i is to be modified. The constants obey `TEXTURE`i = `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`.

Gets 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

> `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 described in equation 2.2 for the corresponding (signed) type.

The current fog coordinate is set using

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

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

`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 2.13.)

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 2.13 on colors and coloring). Values outside $[0, 1]$ are not clamped.

The command

`void` **Index**{**sifd ub**}(`T` *index*);
`void` **Index**{**sifd 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 2.14) 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`.

Current generic attribute values define generic attributes for a vertex. The current values of a generic shader attribute declared as a floating-point scalar, vector, or matrix may be changed at any time by issuing one of the commands

`void` **VertexAttrib**{**1234**}{**sfd**}(`uint` *index*, `T` *values*);
`void` **VertexAttrib**{**123**}{**sfd**}**v**(`uint` *index*, `T` *values*);
`void` **VertexAttrib4**{**bsifd ub us ui**}**v**(`uint` *index*, `T` *values*);
`void` **VertexAttrib4Nub**(`uint` *index*, `T` *values*);
`void` **VertexAttrib4N**{**bsi ub us ui**}**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 described in equations 2.1 and 2.2, respectively, while the other commands specify values that are converted directly to the internal floating-point representation.

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

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

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

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

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

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

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

Setting generic vertex attribute zero specifies a vertex; the four vertex coordinates 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 the value of `MAX_VERTEX_ATTRIBS` $- 1$ four-component 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 $(0.0, 0.0, 0.0, 1.0)$.

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

```
void VertexPointer( int size, enum type, sizei stride,
   void *pointer );
void NormalPointer( enum type, sizei stride,
   void *pointer );
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*);
`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 always type `boolean`, **EdgeFlagPointer** has no *type* argument. *size*, when present, indicates the number of values per vertex that are stored in the array as well as their component ordering. 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 2.5 indicates the allowable values for *size* and *type* (when present). For *type* the values `BYTE`, `SHORT`, `INT`, `FLOAT`, `HALF_FLOAT`, and `DOUBLE` indicate types `byte`, `short`, `int`, `float`, `half`, and `double`, respectively; and the values `UNSIGNED_BYTE`, `UNSIGNED_SHORT`, and `UNSIGNED_INT` indicate types `ubyte`, `ushort`, and `uint`, respectively. The error `INVALID_VALUE` is generated if *size* is specified with a value other than that indicated in the table, if *size* is `BGRA` and *type* is not `UNSIGNED_BYTE`, or by **VertexAttribPointer** if *size* is `BGRA` and *normalized* is `FALSE`.

An `INVALID_OPERATION` error is generated if any of the ***Pointer** commands specifying the location and organization of vertex array data are called while a non-zero vertex array object is bound (see section 2.10), zero is bound to the `ARRAY_BUFFER` buffer object binding point (see section 2.9.6), and the *pointer* argument is not `NULL`[2].

The *index* parameter in the **VertexAttribPointer** and **VertexAttribIPointer** commands identifies the generic vertex attribute array being described. The error `INVALID_VALUE` is generated if *index* is greater than or equal to the value of `MAX_VERTEX_ATTRIBS`. Generic attribute arrays with integer *type* arguments can be handled in one of three ways: converted to float by normalizing to $[0, 1]$ or $[-1, 1]$ as described in equations 2.1 and 2.2, respectively; converted directly to float, or left as integers. Data for an array specified by **VertexAttribPointer** will be converted to floating-point by normalizing if *normalized* is `TRUE`, and converted

---

[2] This error makes it impossible to create a vertex array object containing client array pointers, while still allowing buffer objects to be unbound.

Command	Sizes and Component Ordering	Integer Handling	Types
VertexPointer	2, 3, 4	cast	`short`, `int`, `float`, `half`, `double`
NormalPointer	3	normalize	`byte`, `short`, `int`, `float`, `half`, `double`
ColorPointer	3, 4, `BGRA`	normalize	`byte`, `ubyte`, `short`, `ushort`, `int`, `uint`, `float`, `half`, `double`
SecondaryColorPointer	3, `BGRA`	normalize	`byte`, `ubyte`, `short`, `ushort`, `int`, `uint`, `float`, `half`, `double`
IndexPointer	1	cast	`ubyte`, `short`, `int`, `float`, `double`
FogCoordPointer	1	n/a	`float`, `half`, `double`
TexCoordPointer	1, 2, 3, 4	cast	`short`, `int`, `float`, `half`, `double`
EdgeFlagPointer	1	integer	`boolean`
VertexAttribPointer	1, 2, 3, 4, `BGRA`	flag	`byte`, `ubyte`, `short`, `ushort`, `int`, `uint`, `float`, `half`, `double`
VertexAttribIPointer	1, 2, 3, 4	integer	`byte`, `ubyte`, `short`, `ushort`, `int`, `uint`

Table 2.5: 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 are 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**. If *size* is `BGRA`, vertex array values are always normalized, irrespective of the "normalize" table entry.

directly to floating-point otherwise. Data for an array specified by **VertexAttribIPointer** will always be left as integer values; such data are referred to as *pure* integers.

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

`void` **EnableClientState**(`enum` *array*);
`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

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

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

The command

`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

`void` **ArrayElement**(`int` *i*);

transfers the ith element of every enabled array to the GL. The effect of **ArrayElement**(i) is the same as the effect of the command sequence

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

where *textureUnits* and *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 **Begin** and the corresponding execution of **End** may affect calls to **ArrayElement** that are made within the same **Begin/End** period in non-sequential ways. That is, a call to **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 `INVALID_VALUE` is recommended in this case.

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

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

and

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

with *target* `PRIMITIVE_RESTART`. The command

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

specifies the index of a vertex array element that is treated specially when primitive restarting is enabled. This value is called the *primitive restart index*. When **ArrayElement** is called between an execution of **Begin** and the corresponding execution of **End**, if i is equal to the primitive restart index, then no vertex data is dereferenced, and no current vertex state is modified. Instead, it is as if **End** were called, followed by a call to **Begin** where *mode* is the same as the mode used by the previous **Begin**.

When one of the ***BaseVertex** drawing commands specified in section 2.8.1 is used, the primitive restart comparison occurs before the *basevertex* offset is added to the array index.

2.8.1 Drawing Commands

The command

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

constructs a sequence of geometric primitives using elements *first* through $first + count - 1$ of each enabled array. *mode* specifies what kind of primitives are constructed, and accepts the same token values as the *mode* parameter of the **Begin** command. The effect of

DrawArrays ($mode, first, count$);

is the same as the effect of the command sequence

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

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

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

The command

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

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

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

The command

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

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

```
DrawElements (mode, count, type, indices);
```

is the same as the effect of the command sequence

```
if (mode, count, or type is invalid )
  generate appropriate error
else {
  Begin(mode);
  for (int i = 0; i < count ; i++)
    ArrayElement(indices[i]);
  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 the execution of **DrawElements**, if the corresponding array is enabled. Current values corresponding to disabled arrays are not modified by the execution of **DrawElements**.

The command

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

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

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

The command

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

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

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

The error `INVALID_VALUE` is generated if $end < start$. Invalid *mode*, *count*, or *type* parameters generate the same errors as would the corresponding call to **DrawElements**. It is an error for index values other than the primitive restart index to lie outside the range $[start, end]$, but implementations are not required to check for this. Such indices will cause implementation-dependent behavior.

The internal counter *instanceID* is a 32-bit integer value which may be read by a vertex shader as `gl_InstanceID`, as described in section 2.14.4. The value of this counter is always zero, except as noted below.

The command

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

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

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

The command

```
void DrawElementsInstanced( enum mode, sizei count,
    enum type, const void *indices, sizei primcount );
```

behaves identically to **DrawElements** except that *primcount* instances of the set of elements are executed, and the value of *instanceID* advances for each iteration. It has the same effect as:

```
if ( $mode$, $count$, or $type$ is invalid )
    generate appropriate error
else {
  for (int i = 0; i < $primcount$ ; i++) {
    instanceID = i;
    DrawElements( $mode$, $count$, $type$, $indices$ );
  }
  instanceID = 0;
}
```

The commands

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

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

The command

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

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

```
for (int i = 0; i < primcount ; i++)
  if (count[i] > 0)
    DrawElementsBaseVertex(mode, count[i], type,
      indices[i], basevertex[i]);
```

The command

void **InterleavedArrays**(enum *format*, sizei *stride*,
void **pointer*);

efficiently initializes the six arrays and their enables to one of 14 configurations. *format* must be one of 14 symbolic constants: `V2F`, `V3F`, `C4UB_V2F`, `C4UB_V3F`, `C3F_V3F`, `N3F_V3F`, `C4F_N3F_V3F`, `T2F_V3F`, `T4F_V4F`, `T2F_C4UB_V3F`, `T2F_C3F_V3F`, `T2F_N3F_V3F`, `T2F_C4F_N3F_V3F`, or `T4F_C4F_N3F_V4F`.

The effect of

InterleavedArrays($format, stride, pointer$);

is the same as the effect of the command sequence

$format$	e_t	e_c	e_n	s_t	s_c	s_v	t_c
V2F	*False*	*False*	*False*			2	
V3F	*False*	*False*	*False*			3	
C4UB_V2F	*False*	*True*	*False*		4	2	UNSIGNED_BYTE
C4UB_V3F	*False*	*True*	*False*		4	3	UNSIGNED_BYTE
C3F_V3F	*False*	*True*	*False*		3	3	FLOAT
N3F_V3F	*False*	*False*	*True*			3	
C4F_N3F_V3F	*False*	*True*	*True*		4	3	FLOAT
T2F_V3F	*True*	*False*	*False*	2		3	
T4F_V4F	*True*	*False*	*False*	4		4	
T2F_C4UB_V3F	*True*	*True*	*False*	2	4	3	UNSIGNED_BYTE
T2F_C3F_V3F	*True*	*True*	*False*	2	3	3	FLOAT
T2F_N3F_V3F	*True*	*False*	*True*	2		3	
T2F_C4F_N3F_V3F	*True*	*True*	*True*	2	4	3	FLOAT
T4F_C4F_N3F_V4F	*True*	*True*	*True*	4	4	4	FLOAT

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

Table 2.6: 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).

```
if ($format$ or $stride$ is invalid)
  generate appropriate error
else {
  int str;
  set $e_t, e_c, e_n, s_t, s_c, s_v, t_c, p_c, p_n, p_v$, and $s$ as a function
    of table 2.6 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 ($e_t$) {
    EnableClientState(TEXTURE_COORD_ARRAY);
    TexCoordPointer($s_t$, FLOAT, str, pointer);
  } else
    DisableClientState(TEXTURE_COORD_ARRAY);
  if ($e_c$) {
    EnableClientState(COLOR_ARRAY);
    ColorPointer($s_c$, $t_c$, str, pointer + $p_c$);
  } else
    DisableClientState(COLOR_ARRAY);
  if ($e_n$) {
    EnableClientState(NORMAL_ARRAY);
    NormalPointer(FLOAT, str, pointer + $p_n$);
  } else
    DisableClientState(NORMAL_ARRAY);
  EnableClientState(VERTEX_ARRAY);
  VertexPointer($s_v$, FLOAT, str, pointer + $p_v$);
}
```

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, n boolean values indicating whether the attribute values are pure integers, and an unsigned integer representing the restart index.

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, the normalized and pure integer flags are each false, and the restart index is zero.

2.9 Buffer Objects

Vertex array data (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.

The command

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

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

Buffer objects are deleted by calling

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

buffers contains *n* names of buffer objects to be deleted. After a buffer object is deleted it has no contents, and its name is again unused. Unused names in *buffers* are silently ignored, as is the value zero.

2.9.1 Creating and Binding Buffer Objects

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 the targets listed in table 2.7. 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 2.8.

Target name	Purpose	Described in section(s)
`ARRAY_BUFFER`	Vertex attributes	2.9.6
`COPY_READ_BUFFER`	Buffer copy source	2.9.5
`COPY_WRITE_BUFFER`	Buffer copy destination	2.9.5
`ELEMENT_ARRAY_BUFFER`	Vertex array indices	2.9.7
`PIXEL_PACK_BUFFER`	Pixel read target	4.3.2, 6.1
`PIXEL_UNPACK_BUFFER`	Texture data source	3.7
`TEXTURE_BUFFER`	Texture data buffer	3.9.5
`TRANSFORM_FEEDBACK_BUFFER`	Transform feedback buffer	2.19
`UNIFORM_BUFFER`	Uniform block storage	2.14.4

Table 2.7: Buffer object binding targets.

Buffer objects created by binding an unused name to any of the valid *target*s are formally equivalent, but the GL may make different choices about storage location and layout based on the initial binding.

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

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

If a buffer object is deleted while it is bound, all bindings to that object in the current context (i.e. in the thread that called **DeleteBuffers**) are reset to zero. Bindings to that buffer in other contexts and other threads are not affected, but attempting to use a deleted buffer in another thread produces undefined results, including but not limited to possible GL errors and rendering corruption. Using a deleted buffer in another context or thread may not, however, result in program termination.

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

Binding Buffer Objects to Indexed Targets

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

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

Table 2.8: Buffer object parameters and their values.

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

target must be `TRANSFORM_FEEDBACK_BUFFER` or `UNIFORM_BUFFER`. Additional language specific to each target is included in sections referred to for each target in table 2.7.

Each *target* represents an indexed array of buffer object binding points, as well as a single general binding point that can be used by other buffer object manipulation functions (e.g. **BindBuffer**, **MapBuffer**). Both commands bind the buffer object named by *buffer* to both the general binding point, and to the binding point in the array given by *index*. The error `INVALID_VALUE` is generated if *index* is greater than or equal to the number of *target*-specific indexed binding points.

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

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

2.9.2 Creating Buffer Object Data Stores

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

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

with *target* set to one of the targets listed in table 2.7, *size* set to the size of the data store in basic machine units, and *data* pointing to the source data in client memory. If *data* is non-null, then the source data is copied to the buffer object's data store. If *data* is null, then the contents of the buffer object's data store are undefined.

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

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

`STREAM_READ` The data store contents will be specified once by reading data from the GL, and queried at most a few times by the application.

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

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

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

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

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

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

Name	Value
`BUFFER_SIZE`	*size*
`BUFFER_USAGE`	*usage*
`BUFFER_ACCESS`	`READ_WRITE`
`BUFFER_ACCESS_FLAGS`	0
`BUFFER_MAPPED`	`FALSE`
`BUFFER_MAP_POINTER`	`NULL`
`BUFFER_MAP_OFFSET`	0
`BUFFER_MAP_LENGTH`	0

Table 2.9: Buffer object initial state.

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

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

BufferData deletes any existing data store, and sets the values of the buffer object's state variables as shown in table 2.9.

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

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

with *target* set to one of the targets listed in table 2.7. *offset* and *size* indicate the range of data in the buffer object that is to be replaced, in terms of basic machine units. *data* specifies a region of client memory *size* basic machine units in length, containing the data that replace the specified buffer range. An `INVALID_VALUE` error is generated if *offset* or *size* is less than zero or if $offset + size$ is greater than the value of `BUFFER_SIZE`. An `INVALID_OPERATION` error is generated if any part of the specified buffer range is mapped with **MapBufferRange** or **MapBuffer** (see section 2.9.3).

2.9.3 Mapping and Unmapping Buffer Data

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

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

with *target* set to one of the targets listed in table 2.7. *offset* and *length* indicate the range of data in the buffer object that is to be mapped, in terms of basic machine units. *access* is a bitfield containing flags which describe the requested mapping. These flags are described below.

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

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

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

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

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

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

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

- `MAP_FLUSH_EXPLICIT_BIT` indicates that one or more discrete subranges of the mapping may be modified. When this flag is set, modifications to each subrange must be explicitly flushed by calling **FlushMappedBufferRange**. No GL error is set if a subrange of the mapping is modified and not flushed, but data within the corresponding subrange of the buffer are undefined. This flag may only be used in conjunction with `MAP_WRITE_BIT`. When this option is selected, flushing is strictly limited to regions that are explicitly indicated with calls to **FlushMappedBufferRange** prior to unmap; if this option is not selected **UnmapBuffer** will automatically flush the entire mapped range when called.

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

A successful **MapBufferRange** sets buffer object state values as shown in table 2.10.

Errors

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

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

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

Name	Value
`BUFFER_ACCESS`	Depends on *access*[1]
`BUFFER_ACCESS_FLAGS`	*access*
`BUFFER_MAPPED`	`TRUE`
`BUFFER_MAP_POINTER`	pointer to the data store
`BUFFER_MAP_OFFSET`	*offset*
`BUFFER_MAP_LENGTH`	*length*

Table 2.10: Buffer object state set by **MapBufferRange**.
[1] `BUFFER_ACCESS` is set to `READ_ONLY`, `WRITE_ONLY`, or `READ_WRITE` if *access* & (`MAP_READ_BIT`|`MAP_WRITE_BIT`) is respectively `MAP_READ_BIT`, `MAP_WRITE_BIT`, or `MAP_READ_BIT`|`MAP_WRITE_BIT`.

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

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

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

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

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

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

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

and *mbaccess* is the value of the *access* `enum` parameter passed to **MapBuffer**.

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

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

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

with *target* set to one of the targets listed in table 2.7. *offset* and *length* indicate a modified subrange of the mapping, in basic machine units. The specified subrange to flush is relative to the start of the currently mapped range of buffer. **FlushMappedBufferRange** may be called multiple times to indicate distinct subranges of the mapping which require flushing.

Errors

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

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

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

Unmapping Buffers

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

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

with *target* set to one of the targets listed in table 2.7. Unmapping a mapped buffer object invalidates the pointer to its data store and sets the object's `BUFFER_MAPPED`, `BUFFER_MAP_POINTER`, `BUFFER_ACCESS_FLAGS`, `BUFFER_MAP_OFFSET`, and `BUFFER_MAP_LENGTH` state variables to the initial values shown in table 2.9.

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

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

Effects of Mapping Buffers on Other GL Commands

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

2.9.4 Effects of Accessing Outside Buffer Bounds

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

2.9.5 Copying Between Buffers

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

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

with *readtarget* and *writetarget* each set to one of the targets listed in table 2.7. While any of these targets may be used, the `COPY_READ_BUFFER` and `COPY_WRITE_BUFFER` targets are provided specifically for copies, so that they can be done without affecting other buffer binding targets that may be in use. *writeoffset* and *size* specify the range of data in the buffer object bound to *writetarget* that is to be replaced, in terms of basic machine units. *readoffset* and *size* specify the range of data in the buffer object bound to *readtarget* that is to be copied to the corresponding region of *writetarget*.

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

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

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

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

2.9.6 Vertex Arrays in Buffer Objects

Blocks of vertex array data 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 **VertexAttribPointer** command copies the value of `ARRAY_BUFFER_BINDING` (the queriable name of the buffer binding corresponding to the target `ARRAY_BUFFER`) to the client state variable `VERTEX_ATTRIB_ARRAY_BUFFER_BINDING` for the specified *index*.

Rendering commands **ArrayElement**, **DrawArrays**, and the other drawing commands defined in section 2.8.1 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.

2.9.7 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**, **DrawRangeElements**, and **DrawElementsInstanced** source their indices from that buffer object, using their *indices* parameters as offsets into the buffer object in the same fashion as described in section 2.9.6. **DrawElementsBaseVertex**, **DrawRangeElementsBaseVertex**, and **DrawElementsInstancedBaseVertex** also source their indices from that buffer object, adding the *basevertex* offset to the appropriate vertex index as a final step before indexing into the vertex buffer; this does not affect the calculation of the base pointer for the index array. Finally, **MultiDrawElements** and **MultiDrawElementsBaseVertex** also source their indices from that buffer object, using its *indices* parameter as a pointer to an array of pointers that represent offsets into the buffer object.

In some cases performance will be optimized by storing indices and array data in separate buffer objects, and by creating those buffer objects with the corresponding binding points.

2.9.8 Buffer Object State

The state required to support buffer objects consists of binding names for the array buffer, element buffer, pixel unpack buffer, and pixel pack buffer. Additionally, each vertex array has an associated binding so there is a buffer object binding for each of the vertex 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, two integers for the offset and size of the mapped region, a pointer to the mapped buffer (`NULL` if unmapped), and the sized array of basic machine units for the buffer data.

2.10 Vertex Array Objects

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

The command

`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, but they acquire array state only when they are first bound.

Vertex array objects are deleted by calling

`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 that is currently bound is deleted, the binding for that object reverts to zero and the default vertex array becomes current. Unused names in *arrays* are silently ignored, as is the value zero.

A vertex array object is created by binding a name returned by **GenVertexArrays** with the command

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

array is the vertex array object name. The resulting vertex array object is a new state vector, comprising all the state values listed in tables 6.7- 6.10.

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

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

BindVertexArray fails and an `INVALID_OPERATION` error is generated if *array* is not zero or a name returned from a previous call to **GenVertexArrays**, or if such a name has since been deleted with **DeleteVertexArrays**.

2.11 Rectangles

There is a set of GL commands to support efficient specification of rectangles as two corner vertices.

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

Rect (x_1, y_1, x_2, y_2);

is exactly the same as the following sequence of commands:

Begin(POLYGON);
  Vertex2(x_1, y_1);
  Vertex2(x_2, y_1);
  Vertex2(x_2, y_2);
  Vertex2(x_1, y_2);
End();

The appropriate **Vertex2** command would be invoked depending on which of the **Rect** commands is issued.

2.12 Fixed-Function Vertex Transformations

This section and the following discussion through section 2.13 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 2.14.

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 2.9 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. Clip coordinates are further processed as described in section 2.16.

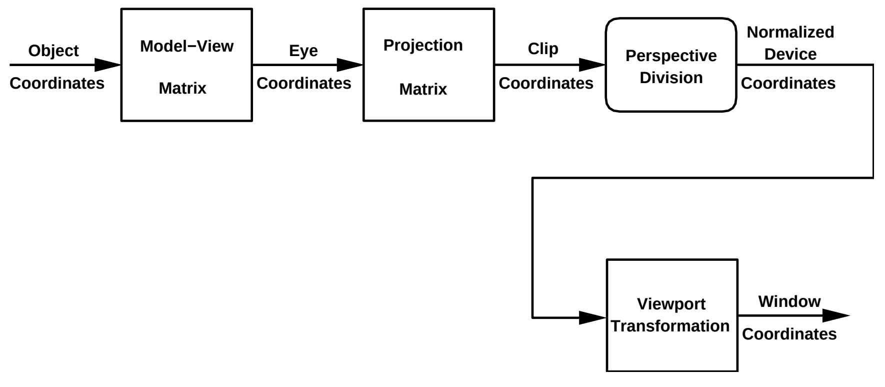


Figure 2.9. Vertex transformation sequence.

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 4×4.

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

$$\begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} = M \begin{pmatrix} x_o \\ y_o \\ z_o \\ w_o \end{pmatrix}.$$

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

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

2.12.1 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

```
void MatrixMode( enum mode );
```

which takes one of the pre-defined constants `TEXTURE`, `MODELVIEW`, `COLOR`, or `PROJECTION` as the argument value. `TEXTURE` is described later in section 2.12.1, and `COLOR` is described in section 3.7.3. If the current matrix mode is `MODELVIEW`, then matrix operations apply to the model-view matrix; if `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 4×4 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 4×4 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

MultTransposeMatrix[fd](m);

is the same as the effect of

MultMatrix[fd](m^T);

The command

`void` **LoadIdentity**(`void`);

effectively calls **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. **Rotate**, **Translate**, **Scale**, **Frustum**, and **Ortho** manipulate the current matrix. Each computes a matrix and then invokes **MultMatrix** with this matrix. In the case of

`void` **Rotate{fd}**(`T` θ, `T` x, `T` y, `T` z);

θ gives an angle of rotation in degrees; the coordinates of a vector $\mathbf{v}$ are given by $\mathbf{v} = (x\ y\ z)^T$. 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

$$\begin{pmatrix} & & & 0 \\ & R & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

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

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

then

$$R = \mathbf{u}\mathbf{u}^T + \cos\theta(I - \mathbf{u}\mathbf{u}^T) + \sin\theta S.$$

The arguments to

`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

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

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 4×4 matrix is applied to the corresponding texture coordinates. This matrix is applied as

$$\begin{pmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{pmatrix} \begin{pmatrix} s \\ t \\ r \\ q \end{pmatrix},$$

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 active texture unit selector (see section 3.9) 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 2.12.3), **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.

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.

`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.

`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 a four-valued integer indicating the current matrix mode, one stack of at least two 4×4 matrices for each of `COLOR`, `PROJECTION`, and each texture coordinate set, `TEXTURE`; and a stack of at least 32 4×4 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 matrix mode is `MODELVIEW`.

2.12.2 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 calling **Enable** and **Disable** 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:

$$\begin{pmatrix} n_x{}' & n_y{}' & n_z{}' & q' \end{pmatrix} = \begin{pmatrix} n_x & n_y & n_z & q \end{pmatrix} \cdot M^{-1}$$

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

$$q = \begin{cases} 0, & w = 0, \\ \dfrac{-\begin{pmatrix} n_x & n_y & n_z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}}{w}, & w \neq 0 \end{cases} \tag{2.7}$$

Implementations may choose instead to transform $\begin{pmatrix} n_x & n_y & n_z \end{pmatrix}$ to eye coordinates using

$$\begin{pmatrix} n_x{}' & n_y{}' & n_z{}' \end{pmatrix} = \begin{pmatrix} n_x & n_y & n_z \end{pmatrix} \cdot M_u{}^{-1}$$

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

Rescale multiplies the transformed normals by a scale factor

$$\begin{pmatrix} n_x{}'' & n_y{}'' & n_z{}'' \end{pmatrix} = f \begin{pmatrix} n_x{}' & n_y{}' & n_z{}' \end{pmatrix}$$

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

$$n_f = m \begin{pmatrix} n_x{}'' & n_y{}'' & n_z{}'' \end{pmatrix}$$

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.

2.12.3 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

```
void TexGen{ifd}( enum coord, enum pname, T param );
void TexGen{ifd}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 `TEXTURE_GEN_MODE`, `OBJECT_PLANE`, or `EYE_PLANE`. If *pname* is `TEXTURE_GEN_MODE`, then either *params* points to or *param* is an integer that is one of the symbolic constants `OBJECT_LINEAR`, `EYE_LINEAR`, `SPHERE_MAP`, `REFLECTION_MAP`, or `NORMAL_MAP`.

If `TEXTURE_GEN_MODE` indicates `OBJECT_LINEAR`, 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 `OBJECT_PLANE` 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 `TEXTURE_GEN_MODE` indicates `EYE_LINEAR`, 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 `EYE_PLANE` in correspondence with setting the coefficients in the `OBJECT_PLANE` 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 `TEXTURE_GEN_MODE` indicating `SPHERE_MAP` can simulate the reflected image of 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 $\mathbf{r} = \begin{pmatrix} r_x & r_y & r_z \end{pmatrix}^T$, the reflection vector, be given by

$$\mathbf{r} = \mathbf{u} - 2\mathbf{n_f}^T \left(\mathbf{n_f} \mathbf{u}\right),$$

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 *pname* indicates `SPHERE_MAP` generates the error `INVALID_ENUM`.

If `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 *pname* indicates `REFLECTION_MAP` generates the error `INVALID_ENUM`.

If `TEXTURE_GEN_MODE` indicates `NORMAL_MAP`, compute the normal vector n_f as described in section 2.12.2. Then the value assigned to an s coordinate is $s = n_{f_x}$; the value assigned to a t coordinate is $t = n_{f_y}$; and the value assigned to an r coordinate is $r = n_{f_z}$ (the values n_{f_x}, n_{f_y}, and n_{f_z} are the components of n_f.) Calling **TexGen** with a *coord* of `Q` when *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 `TEXTURE_GEN_S`, `TEXTURE_GEN_T`, `TEXTURE_GEN_R`, or `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`.

2.13 Fixed-Function Vertex Lighting and Coloring

Figures 2.10 and 2.11 diagram the processing of RGBA colors and color indices before rasterization. Incoming colors arrive in one of several formats. R, G, B, and A components specified with unsigned and signed integer versions of the **Color** command are converted to floating-point as described in equations 2.1 and 2.2, respectively. 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 2.13.6. A color index is converted to fixed-point and then its integer portion is masked (see section 2.13.6). 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.

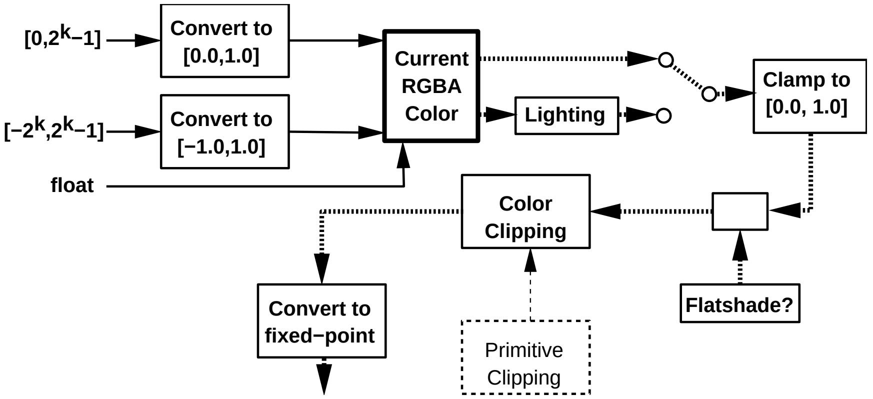


Figure 2.10. Processing of RGBA colors. The heavy dotted lines indicate both primary and secondary vertex colors, which are processed in the same fashion. k is the minimum required bit width of the integer type representing a color component.

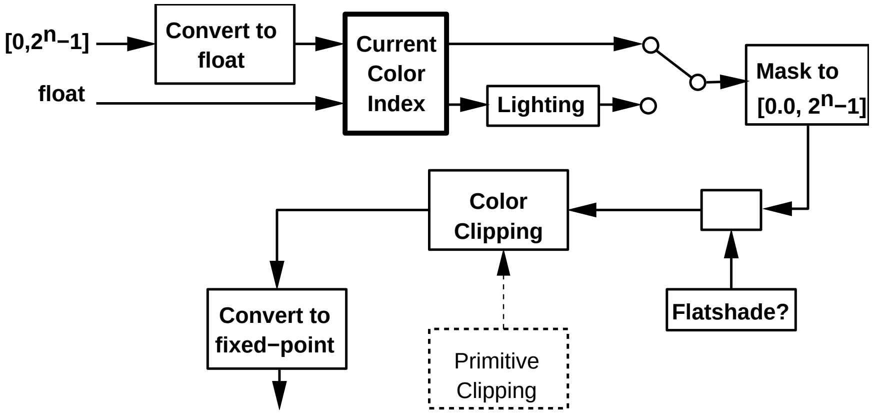


Figure 2.11. Processing of color indices. n is the number of bits in a color index.

2.13.1 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 2.13.5.)

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 $\mathbf{d}_{cli}$ and $\mathbf{s}_{cli}$ 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 $\mathbf{c}_1$ and $\mathbf{c}_2$ are colors without alpha where $\mathbf{c}_1 = (r_1, g_1, b_1)$ and $\mathbf{c}_2 = (r_2, g_2, b_2)$, then define $\mathbf{c}_1 * \mathbf{c}_2 = (r_1r_2, g_1g_2, b_1b_2)$. Addition of colors is accomplished by addition of 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 $\mathbf{P}_1$

Parameter	Type	Default Value	Description
Material Parameters			
$\mathbf{a}_{cm}$	color	$(0.2, 0.2, 0.2, 1.0)$	ambient color of material
$\mathbf{d}_{cm}$	color	$(0.8, 0.8, 0.8, 1.0)$	diffuse color of material
$\mathbf{s}_{cm}$	color	$(0.0, 0.0, 0.0, 1.0)$	specular color of material
$\mathbf{e}_{cm}$	color	$(0.0, 0.0, 0.0, 1.0)$	emissive color of material
s_{rm}	real	0.0	specular exponent (range: $[0.0, 128.0]$)
a_m	real	0.0	ambient color index
d_m	real	1.0	diffuse color index
s_m	real	1.0	specular color index
Light Source Parameters			
$\mathbf{a}_{cli}$	color	$(0.0, 0.0, 0.0, 1.0)$	ambient intensity of light i
$\mathbf{d}_{cli}(i = 0)$ $\mathbf{d}_{cli}(i > 0)$	color color	$(1.0, 1.0, 1.0, 1.0)$ $(0.0, 0.0, 0.0, 1.0)$	diffuse intensity of light 0 diffuse intensity of light i
$\mathbf{s}_{cli}(i = 0)$ $\mathbf{s}_{cli}(i > 0)$	color color	$(1.0, 1.0, 1.0, 1.0)$ $(0.0, 0.0, 0.0, 1.0)$	specular intensity of light 0 specular intensity of light i
$\mathbf{P}_{pli}$	position	$(0.0, 0.0, 1.0, 0.0)$	position of light i
$\mathbf{s}_{dli}$	direction	$(0.0, 0.0, -1.0)$	direction of spotlight for light i
s_{rli}	real	0.0	spotlight exponent for light i (range: $[0.0, 128.0]$)
c_{rli}	real	180.0	spotlight cutoff angle for light i (range: $[0.0, 90.0]$, 180.0)
k_{0i}	real	1.0	constant attenuation factor for light i (range: $[0.0, \infty)$)
k_{1i}	real	0.0	linear attenuation factor for light i (range: $[0.0, \infty)$)
k_{2i}	real	0.0	quadratic attenuation factor for light i (range: $[0.0, \infty)$)
Lighting Model Parameters			
$\mathbf{a}_{cs}$	color	$(0.2, 0.2, 0.2, 1.0)$	ambient color of scene
v_{bs}	boolean	`FALSE`	viewer assumed to be at $(0, 0, 0)$ in eye coordinates (`TRUE`) or $(0, 0, \infty)$ (`FALSE`)
c_{es}	enum	`SINGLE_COLOR`	controls computation of colors
t_{bs}	boolean	`FALSE`	use two-sided lighting mode

Table 2.11: Summary of lighting parameters. The range of individual color components is $(-\infty, +\infty)$.

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 corresponding to the direction specified by the x, y, and z coordinates of $\mathbf{P}_2$; if $\mathbf{P}_1$ has a zero w coordinate and $\mathbf{P}_2$ has a 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 $\|\mathbf{P}_1\mathbf{P}_2\|$ 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, c_{es}. If $c_{es} =$ `SINGLE_COLOR`, then the equations to compute $\mathbf{c}_{pri}$ and $\mathbf{c}_{sec}$ are

$$\begin{aligned}
\mathbf{c}_{pri} &= \mathbf{e}_{cm} \\
&+ \mathbf{a}_{cm} * \mathbf{a}_{cs} \\
&+ \sum_{i=0}^{n-1} (att_i)(spot_i)\,[\mathbf{a}_{cm} * \mathbf{a}_{cli} \\
&\qquad\qquad + (\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})\mathbf{d}_{cm} * \mathbf{d}_{cli} \\
&\qquad\qquad + (f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}} \mathbf{s}_{cm} * \mathbf{s}_{cli}] \\
\mathbf{c}_{sec} &= (0,0,0,1)
\end{aligned}$$

If $c_{es} =$ `SEPARATE_SPECULAR_COLOR`, then

$$\begin{aligned}
\mathbf{c}_{pri} &= \mathbf{e}_{cm} \\
&+ \mathbf{a}_{cm} * \mathbf{a}_{cs} \\
&+ \sum_{i=0}^{n-1} (att_i)(spot_i)\,[\mathbf{a}_{cm} * \mathbf{a}_{cli} \\
&\qquad\qquad + (\mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli})\mathbf{d}_{cm} * \mathbf{d}_{cli}] \\
\mathbf{c}_{sec} &= \sum_{i=0}^{n-1} (att_i)(spot_i)(f_i)(\mathbf{n} \odot \hat{\mathbf{h}}_i)^{s_{rm}} \mathbf{s}_{cm} * \mathbf{s}_{cli}
\end{aligned}$$

where

$$f_i = \begin{cases} 1, & \mathbf{n} \odot \overrightarrow{\mathbf{VP}}_{pli} \neq 0, \\ 0, & \text{otherwise}, \end{cases} \tag{2.8}$$

$$\mathbf{h}_i = \begin{cases} \overrightarrow{\mathbf{VP}}_{pli} + \overrightarrow{\mathbf{VP}}_e, & v_{bs} = \mathtt{TRUE}, \\ \overrightarrow{\mathbf{VP}}_{pli} + \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^T, & v_{bs} = \mathtt{FALSE}, \end{cases} \tag{2.9}$$

$$att_i = \begin{cases} \dfrac{1}{k_{0i} + k_{1i}\|\mathbf{VP}_{pli}\| + k_{2i}\|\mathbf{VP}_{pli}\|^2}, & \text{if } \mathbf{P}_{pli}\text{'s } w \neq 0, \\ 1.0, & \text{otherwise.} \end{cases} \tag{2.10}$$

$$spot_i = \begin{cases} (\overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli})^{s_{rli}}, & c_{rli} \neq 180.0, \overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli} \geq \cos(c_{rli}), \\ 0.0, & c_{rli} \neq 180.0, \overrightarrow{\mathbf{P}_{pli}\mathbf{V}} \odot \hat{\mathbf{s}}_{dli} < \cos(c_{rli}), \\ 1.0, & c_{rli} = 180.0. \end{cases} \tag{2.11}$$

All computations are carried out in eye coordinates.

The value of A produced by lighting is the alpha value associated with $\mathbf{d}_{cm}$. A is always associated with the primary color $\mathbf{c}_{pri}$; the alpha component of $\mathbf{c}_{sec}$ is always 1.

Results of lighting are undefined if the w_e coordinate (w in eye coordinates) of $\mathbf{V}$ is zero.

Lighting may operate in *two-sided* mode ($t_{bs} = \mathtt{TRUE}$), 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 $\mathbf{n}$ with $-\mathbf{n}$. If $t_{bs} = \mathtt{FALSE}$, then the back color and front color are both assigned the color computed using the front material with $\mathbf{n}$.

Additionally, vertex and geometry shaders can operate in two-sided color mode. When a vertex or geometry shader is active, front and back colors can be computed by the shader and written to the `gl_FrontColor`, `gl_BackColor`, `gl_FrontSecondaryColor` and `gl_BackSecondaryColor` outputs. If `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 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 performed based on the sign of the (clipped or unclipped) polygon's area a computed in window coordinates, as described in equation 3.8 of section 3.6.1. If the sign of a (including the possible reversal of this sign as indicated by the last call to **FrontFace**) is positive, the color of each vertex of the polygon becomes the front color computed for that vertex; otherwise the back color is selected.

2.13.2 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 which to set a single-valued parameter. (If *param* corresponds to a multi-valued parameter, the error `INVALID_ENUM` results.) For the **Material** command, *face* must be one of `FRONT`, `BACK`, or `FRONT_AND_BACK`, indicating that the property *name* of the front or back material, or both, respectively, should be set. In the case of **Light**, *light* is a symbolic constant of the form `LIGHT`i, indicating that light i is to have the specified parameter set. The constants obey `LIGHT`i $=$ `LIGHT0` $+ i$.

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 **Material** and **Light** are converted to floating-point values (if specified as integers) as described in equation 2.2. The error `INVALID_VALUE` occurs if a specified lighting parameter lies outside the allowable range given in table 2.11. (The symbol "∞" indicates the maximum representable magnitude for the indicated type.)

Material properties can be changed inside a **Begin**/**End** pair by calling **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 **End** command.

The current model-view matrix is applied to the position parameter indicated with **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 $\mathbf{M}_u$ 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 **Enable** or **Disable** with the symbolic value `LIGHT`i (i is in the range 0 to $n-1$, where n is the implementation-dependent number of lights). If light i is disabled, the ith term in the lighting equation is effectively removed from the summation.

2.13.3 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

`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 $\mathbf{e}_{cm}$, $\mathbf{a}_{cm}$, $\mathbf{d}_{cm}$ or $\mathbf{s}_{cm}$, respectively, will track the current color. If *mode* is `AMBIENT_AND_DIFFUSE`, both $\mathbf{a}_{cm}$ and $\mathbf{d}_{cm}$ track the current color. The replacements made to

Parameter	Name	Number of values
Material Parameters (**Material**)		
$\mathbf{a}_{cm}$	`AMBIENT`	4
$\mathbf{d}_{cm}$	`DIFFUSE`	4
$\mathbf{a}_{cm}, \mathbf{d}_{cm}$	`AMBIENT_AND_DIFFUSE`	4
$\mathbf{s}_{cm}$	`SPECULAR`	4
$\mathbf{e}_{cm}$	`EMISSION`	4
s_{rm}	`SHININESS`	1
a_m, d_m, s_m	`COLOR_INDEXES`	3
Light Source Parameters (**Light**)		
$\mathbf{a}_{cli}$	`AMBIENT`	4
$\mathbf{d}_{cli}$	`DIFFUSE`	4
$\mathbf{s}_{cli}$	`SPECULAR`	4
$\mathbf{P}_{pli}$	`POSITION`	4
$\mathbf{s}_{dli}$	`SPOT_DIRECTION`	3
s_{rli}	`SPOT_EXPONENT`	1
c_{rli}	`SPOT_CUTOFF`	1
k_0	`CONSTANT_ATTENUATION`	1
k_1	`LINEAR_ATTENUATION`	1
k_2	`QUADRATIC_ATTENUATION`	1
Lighting Model Parameters (**LightModel**)		
$\mathbf{a}_{cs}$	`LIGHT_MODEL_AMBIENT`	4
v_{bs}	`LIGHT_MODEL_LOCAL_VIEWER`	1
t_{bs}	`LIGHT_MODEL_TWO_SIDE`	1
c_{es}	`LIGHT_MODEL_COLOR_CONTROL`	1

Table 2.12: Correspondence of lighting parameter symbols to names. `AMBIENT_AND_DIFFUSE` is used to set $\mathbf{a}_{cm}$ and $\mathbf{d}_{cm}$ to the same value.

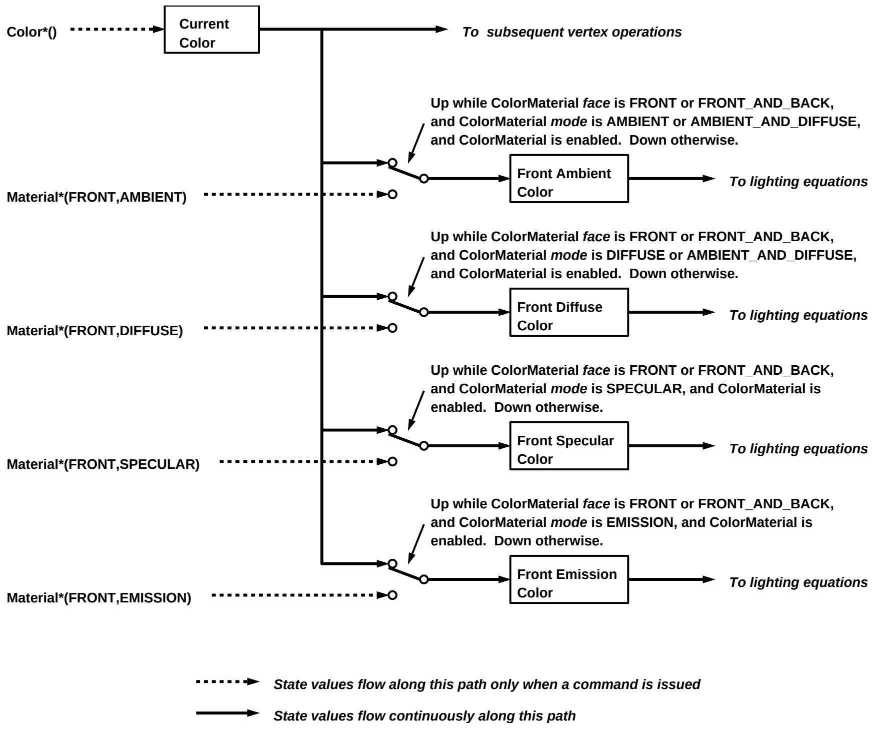

Color*()
Current Color
To subsequent vertex operations
Up while ColorMaterial *face* is FRONT or FRONT_AND_BACK, and ColorMaterial *mode* is AMBIENT or AMBIENT_AND_DIFFUSE, and ColorMaterial is enabled. Down otherwise.
Material*(FRONT,AMBIENT)
Front Ambient Color
To lighting equations
Up while ColorMaterial *face* is FRONT or FRONT_AND_BACK, and ColorMaterial *mode* is DIFFUSE or AMBIENT_AND_DIFFUSE, and ColorMaterial is enabled. Down otherwise.
Material*(FRONT,DIFFUSE)
Front Diffuse Color
To lighting equations
Up while ColorMaterial *face* is FRONT or FRONT_AND_BACK, and ColorMaterial *mode* is SPECULAR, and ColorMaterial is enabled. Down otherwise.
Material*(FRONT,SPECULAR)
Front Specular Color
To lighting equations
Up while ColorMaterial *face* is FRONT or FRONT_AND_BACK, and ColorMaterial *mode* is EMISSION, and ColorMaterial is enabled. Down otherwise.
Material*(FRONT,EMISSION)
Front Emission Color
To lighting equations
State values flow along this path only when a command is issued
State values flow continuously along this path

Figure 2.12. **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`.

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

ColorMaterial (`FRONT`, `AMBIENT`)

while `COLOR_MATERIAL` is enabled sets the front material $\mathbf{a}_{cm}$ 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.

2.13.4 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.

2.13.5 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 ($\mathbf{d}_{cli}$ and $\mathbf{s}_{cli}$, respectively) determine color index diffuse and specular light intensities, d_{li} and s_{li} from

$$d_{li} = (.30)R(\mathbf{d}_{cli}) + (.59)G(\mathbf{d}_{cli}) + (.11)B(\mathbf{d}_{cli})$$

and

$$s_{li} = (.30)R(\mathbf{s}_{cli}) + (.59)G(\mathbf{s}_{cli}) + (.11)B(\mathbf{s}_{cli}).$$

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

Next, let

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

where att_i and $spot_i$ are given by equations 2.10 and 2.11, respectively, and f_i and $\hat{\mathbf{h}}_i$ are given by equations 2.8 and 2.9, respectively. Let $s' = \min\{s, 1\}$. Finally, let

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

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_{bs} 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.

2.13.6 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 **ClampColor**, as described in section 3.7.5, with a *target* of `CLAMP_VERTEX_COLOR`.

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 vertex color clamping is a three-valued integer, initially set to `TRUE`.

2.14 Vertex Shaders

The sequence of operations described in sections 2.12 through 2.13 is a fixed-function method for processing vertex data. Applications can also use vertex shaders to describe the operations that occur on vertex values and their associated data.

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

To use a vertex shader, shader source code is first loaded into a *shader object* and then *compiled*. 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, *geometry shaders* and *fragment shaders* can be created, compiled, and linked into program objects. Geometry shaders affect the processing of primitives assembled from vertices (see section 2.15). Fragment shaders affect the processing of fragments during rasterization (see section 3.12). A single program object can contain all of vertex, geometry, and fragment shaders.

When the program object currently in use includes a vertex shader, its vertex shader is considered *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.

A vertex shader can reference a number of variables as it executes. *Vertex attributes* are the per-vertex values specified in section 2.7. *Uniforms* are per-program variables that are constant during program execution. *Samplers* are a 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. Each of these variable types is described in more detail below.

2.14.1 Shader Objects

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

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

To create a shader object, use the command

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

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

The command

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

> `void` **CompileShader**(`uint` *shader*);

Each shader object has a boolean status, `COMPILE_STATUS`, that is modified as a result of compilation. This status can be queried with **GetShaderiv** (see section 6.1.16). 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 6.1.16).

Shader objects can be deleted with the command

`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 6.1.16). **DeleteShader** will silently ignore the value zero.

2.14.2 Program Objects

The shader objects that are to be used by the programmable stages of the GL are collected together to form a *program object*. The programs that are executed by these programmable stages are called *executables*. All information necessary for defining an executable is encapsulated in a program object. A program object is created with the command

`uint` **CreateProgram**(`void`);

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

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

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

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

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

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

`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

`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 6.1.16). This status will be set to `TRUE` if a valid executable is created, and `FALSE` otherwise. Linking can fail for a variety of reasons as specified in the OpenGL Shading Language Specification. Linking will also fail if one or more of the shader objects, attached to *program* are not compiled successfully, or if more active uniform or active sampler variables are used in *program* than allowed (see sections 2.14.4, 2.14.5, and 2.15.3).

Linking will also fail if the program object contains objects to form a geometry shader (see section 2.15), and

- the program contains no objects to form a vertex shader;
- the input primitive type, output primitive type, or maximum output vertex count is not specified in any compiled geometry shader object; or
- the input primitive type, output primitive type, or maximum output vertex count is specified differently in multiple geometry shader objects.

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

Each program object has an information log that is overwritten as a result of a link operation. This information log can be queried with **GetProgramInfoLog** to obtain more information about the link operation or the validation information (see section 6.1.16).

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

`void` **UseProgram**(`uint` *program*);

This command will install the executable code as part of current rendering state if the program object *program* contains valid executable code, i.e. has been linked successfully. If **UseProgram** is called with *program* set to 0, 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

`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.

2.14.3 Vertex Attributes

Vertex shaders can access built-in vertex attribute variables corresponding to the per-vertex state set by commands such as **Vertex**, **Normal**, and **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:

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

This command provides information about the attribute selected by *index*. An *index* of 0 selects the first active attribute, and an *index* of `ACTIVE_ATTRIBUTES` -1 selects the last active attribute. The value of `ACTIVE_ATTRIBUTES` can be queried with **GetProgramiv** (see section 6.1.16). 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 6.1.16).

For the selected attribute, the type of the attribute is returned into *type*. The size of the attribute is returned into *size*. The value in *size* is in units of the type returned in *type*. The type returned can be any of `FLOAT`, `FLOAT_VEC2`, `FLOAT_VEC3`, `FLOAT_VEC4`, `FLOAT_MAT2`, `FLOAT_MAT3`, `FLOAT_MAT4`, `FLOAT_MAT2x3`, `FLOAT_MAT2x4`, `FLOAT_MAT3x2`, `FLOAT_MAT3x4`, `FLOAT_MAT4x2`, `FLOAT_MAT4x3`, `INT`, `INT_VEC2`, `INT_VEC3`, `INT_VEC4`, `UNSIGNED_INT`, `UNSIGNED_INT_VEC2`, `UNSIGNED_INT_VEC3`, or `UNSIGNED_INT_VEC4`.

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

This command will return as much information about active attributes as possible. If no information is available, *length* will be set to zero and *name* will be an empty string. This situation could arise if **GetActiveAttrib** is issued after a failed link.

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

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

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

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

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

Built-in attribute variables are automatically bound to conventional attributes, and can not have an assigned binding. The error `INVALID_OPERATION` is generated if *name* starts with the reserved `"gl_"` prefix.

When a program is linked, any active attributes without a binding specified through **BindAttribLocation** will automatically be bound to vertex attributes by the GL. Such bindings can be queried using the command **GetAttribLocation**. **LinkProgram** will fail if the assigned binding of an active attribute variable would cause the GL to reference a non-existent generic attribute (one greater than or equal to `MAX_VERTEX_ATTRIBS`). **LinkProgram** will fail if the attribute bindings assigned by **BindAttribLocation** do not leave not enough space to assign a location for an active matrix attribute, which requires multiple contiguous generic attributes. **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 (`gl_Vertex`).

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 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.

2.14.4 Uniform Variables

Shaders can declare named *uniform variables*, as described in the OpenGL Shading Language Specification. Values for these uniforms are constant over a primitive, and typically they are constant across many primitives. Uniforms are program object-specific state. They retain their values once loaded, and their values are restored whenever a program object is used, as long as the program object has not been re-linked. A uniform is considered *active* if it is determined by the compiler and linker that the uniform will actually be accessed when the executable code is executed. In cases where the compiler and linker cannot make a conclusive determination, the uniform will be considered active.

Sets of uniforms can be grouped into *uniform blocks*. The values of each uniform in such a set are extracted from the data store of a buffer object corresponding to the uniform block. OpenGL Shading Language syntax serves to delimit named blocks of uniforms that can be backed by a buffer object. These are referred to as *named uniform blocks*, and are assigned a *uniform block index*. Uniforms that are declared outside of a named uniform block are said to be part of the *default uniform block*. Default uniform blocks have no name or uniform block index. Like uniforms, uniform blocks can be active or inactive. Active uniform blocks are those that contain active uniforms after a program has been compiled and linked.

The amount of storage available for uniform variables in the default uniform block accessed by a vertex shader is specified by the value of the implementation-dependent constant `MAX_VERTEX_UNIFORM_COMPONENTS`. The total amount of combined storage available for uniform variables in all uniform blocks accessed by a vertex shader (including the default uniform block) is specified by the value of the implementation-dependent constant `MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS`. These values represent the numbers of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a vertex shader. A link error is generated if an attempt is made to utilize more than the space available for vertex shader uniform variables.

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

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

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

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

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

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

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

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

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

`uint` **GetUniformBlockIndex**(`uint` *program*, `const`
`char` **uniformBlockName*);

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

uniformBlockName must contain a null-terminated string specifying the name of a uniform block.

GetUniformBlockIndex returns the uniform block index for the uniform block named *uniformBlockName* of *program*. If *uniformBlockName* does not identify an active uniform block of *program*, or an error occurred, then `INVALID_INDEX` is returned. The indices of the active uniform blocks of a program are assigned in consecutive order, beginning with zero.

An active uniform block's name string can be queried from its uniform block index by calling `void` **GetActiveUniformBlockName**(`uint` *program*, `uint` *uniformBlockIndex*, `sizei` *bufSize*, `sizei` **length*, `char` **uniformBlockName*);

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

uniformBlockIndex must be an active uniform block index of *program*, in the range zero to the value of `ACTIVE_UNIFORM_BLOCKS` - 1. The value of `ACTIVE_UNIFORM_BLOCKS` can be queried with **GetProgramiv** (see section 6.1.16). If *uniformBlockIndex* is greater than or equal to the value of `ACTIVE_UNIFORM_BLOCKS`, the error `INVALID_VALUE` is generated.

The string name of the uniform block identified by *uniformBlockIndex* is returned into *uniformBlockName*. The name is null-terminated. The actual number of characters written into *uniformBlockName*, excluding the null terminator, is returned in *length*. If *length* is `NULL`, no length is returned.

bufSize contains the maximum number of characters (including the null terminator) that will be written back to *uniformBlockName*.

If an error occurs, nothing will be written to *uniformBlockName* or *length*.

Information about an active uniform block can be queried by calling

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

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

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

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

If *pname* is `UNIFORM_BLOCK_BINDING`, then the index of the uniform buffer binding point last selected by the uniform block specified by *uniformBlockIndex* for *program* is returned. If no uniform block has been previously specified, zero is returned.

If *pname* is `UNIFORM_BLOCK_DATA_SIZE`, then the implementation-dependent minimum total buffer object size, in basic machine units, required to hold all active uniforms in the uniform block identified by *uniformBlockIndex* is returned. It is neither guaranteed nor expected that a given implementation will arrange uniform values as tightly packed in a buffer object. The exception to this is the `std140` uniform block layout, which guarantees specific packing behavior and does not require the application to query for offsets and strides. In this case the minimum size may still be queried, even though it is determined in advance based only on the uniform block declaration (see "Standard Uniform Block Layout" in section 2.14.4).

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

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

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

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

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

Each active uniform, whether in a named uniform block or in the default block, is assigned an index when a program is linked. Indices are assigned in consecutive order, beginning with zero. The indices assigned to a set of uniforms in a program may be queried by calling

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

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

uniformCount indicates both the number of elements in the array of names *uniformNames* and the number of indices that may be written to *uniformIndices*.

uniformNames contains a list of *uniformCount* name strings identifying the uniform names to be queried for indices. For each name string in *uniformNames*, the index assigned to the active uniform of that name will be written to the corresponding element of *uniformIndices*. If a string in *uniformNames* is not the name of an active uniform, the value `INVALID_INDEX` will be written to the corresponding element of *uniformIndices*.

If an error occurs, nothing is written to *uniformIndices*.

The name of an active uniform may be queried from the corresponding uniform index by calling

```
void GetActiveUniformName( uint program,
   uint uniformIndex, sizei bufSize, sizei *length,
   char *uniformName );
```

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

uniformIndex must be an active uniform index of the program *program*, in the range zero to the value of `ACTIVE_UNIFORMS` - 1. The value of `ACTIVE_UNIFORMS` can be queried with **GetProgramiv**. If *uniformIndex* is greater than or equal to the value of `ACTIVE_UNIFORMS`, the error `INVALID_VALUE` is generated.

The name of the uniform identified by *uniformIndex* is returned as a null-terminated string in *uniformName*. The actual number of characters written into *uniformName*, 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 *uniformName*, including the null terminator, is specified by *bufSize*. The returned uniform name can be the name of built-in uniform state as well. The complete list of built-in uniform state is described in section 7.5 of the OpenGL Shading Language Specification. The length of the longest uniform name in *program* is given by the value of `ACTIVE_UNIFORM_MAX_LENGTH`, which can be queried with **GetProgramiv**.

If **GetActiveUniformName** is not successful, nothing is written to *length* or *uniformName*.

Each uniform variable, declared in a shader, is broken down into one or more strings using the `"."` (dot) and `"[]"` operators, if necessary, to the point that it is legal to pass each string back into **GetUniformLocation**, for default uniform block uniform names, or **GetUniformIndices**, for named uniform block uniform names.

Information about active uniforms can be obtained by calling either

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

or

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

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

These commands provide information about the uniform or uniforms selected by *index* or *uniformIndices*, respectively. In **GetActiveUniform**, an *index* of 0 selects the first active uniform, and an *index* of the value of `ACTIVE_UNIFORMS` - 1 selects the last active uniform. In **GetActiveUniformsiv**, *uniformIndices* is an array of such active uniform indices. If any index is greater than or equal to the value of `ACTIVE_UNIFORMS`, the error `INVALID_VALUE` is generated.

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

Each uniform variable, declared in a shader, is broken down into one or more strings using the `"."` (dot) and `"[]"` operators, if necessary, to the point that it is legal to pass each string back into **GetUniformLocation**, for default uniform block uniform names, or **GetUniformIndices**, for named uniform block uniform names.

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

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

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

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

Type Name Token	Keyword
`FLOAT`	`float`
`FLOAT_VEC2`	`vec2`
`FLOAT_VEC3`	`vec3`
`FLOAT_VEC4`	`vec4`
`INT`	`int`
`INT_VEC2`	`ivec2`
`INT_VEC3`	`ivec3`
`INT_VEC4`	`ivec4`
`UNSIGNED_INT`	`unsigned int`
`UNSIGNED_INT_VEC2`	`uvec2`
`UNSIGNED_INT_VEC3`	`uvec3`
`UNSIGNED_INT_VEC4`	`uvec4`
`BOOL`	`bool`
`BOOL_VEC2`	`bvec2`
`BOOL_VEC3`	`bvec3`
`BOOL_VEC4`	`bvec4`
`FLOAT_MAT2`	`mat2`
`FLOAT_MAT3`	`mat3`
`FLOAT_MAT4`	`mat4`
`FLOAT_MAT2x3`	`mat2x3`
`FLOAT_MAT2x4`	`mat2x4`
`FLOAT_MAT3x2`	`mat3x2`
`FLOAT_MAT3x4`	`mat3x4`
`FLOAT_MAT4x2`	`mat4x2`
`FLOAT_MAT4x3`	`mat4x3`
`SAMPLER_1D`	`sampler1D`
`SAMPLER_2D`	`sampler2D`

(Continued on next page)

OpenGL Shading Language Type Tokens (continued)

Type Name Token	Keyword
`SAMPLER_3D`	`sampler3D`
`SAMPLER_CUBE`	`samplerCube`
`SAMPLER_1D_SHADOW`	`sampler1DShadow`
`SAMPLER_2D_SHADOW`	`sampler2DShadow`
`SAMPLER_1D_ARRAY`	`sampler1DArray`
`SAMPLER_2D_ARRAY`	`sampler2DArray`
`SAMPLER_1D_ARRAY_SHADOW`	`sampler1DArrayShadow`
`SAMPLER_2D_ARRAY_SHADOW`	`sampler2DArrayShadow`
`SAMPLER_2D_MULTISAMPLE`	`sampler2DMS`
`SAMPLER_2D_MULTISAMPLE_ARRAY`	`sampler2DMSArray`
`SAMPLER_CUBE_SHADOW`	`samplerCubeShadow`
`SAMPLER_BUFFER`	`samplerBuffer`
`SAMPLER_2D_RECT`	`sampler2DRect`
`SAMPLER_2D_RECT_SHADOW`	`sampler2DRectShadow`
`INT_SAMPLER_1D`	`isampler1D`
`INT_SAMPLER_2D`	`isampler2D`
`INT_SAMPLER_3D`	`isampler3D`
`INT_SAMPLER_CUBE`	`isamplerCube`
`INT_SAMPLER_1D_ARRAY`	`isampler1DArray`
`INT_SAMPLER_2D_ARRAY`	`isampler2DArray`
`INT_SAMPLER_2D_MULTISAMPLE`	`isampler2DMS`
`INT_SAMPLER_2D_MULTISAMPLE_ARRAY`	`isampler2DMSArray`
`INT_SAMPLER_BUFFER`	`isamplerBuffer`
`INT_SAMPLER_2D_RECT`	`isampler2DRect`
`UNSIGNED_INT_SAMPLER_1D`	`usampler1D`
`UNSIGNED_INT_SAMPLER_2D`	`usampler2D`
`UNSIGNED_INT_SAMPLER_3D`	`usampler3D`
`UNSIGNED_INT_SAMPLER_CUBE`	`usamplerCube`
`UNSIGNED_INT_SAMPLER_1D_ARRAY`	`usampler1DArray`

(Continued on next page)

OpenGL Shading Language Type Tokens (continued)	
Type Name Token	Keyword
`UNSIGNED_INT_SAMPLER_2D_ARRAY`	`usampler2DArray`
`UNSIGNED_INT_SAMPLER_2D_MULTISAMPLE`	`usampler2DMS`
`UNSIGNED_INT_SAMPLER_2D_MULTISAMPLE_ARRAY`	`usampler2DMSArray`
`UNSIGNED_INT_SAMPLER_BUFFER`	`usamplerBuffer`
`UNSIGNED_INT_SAMPLER_2D_RECT`	`usampler2DRect`

Table 2.13: OpenGL Shading Language type tokens returned by **GetActiveUniform** and **GetActiveUniformsiv**, and corresponding shading language keywords declaring each such type.

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

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

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

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

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

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

If *pname* is `UNIFORM_ARRAY_STRIDE`, then an array identifying the stride between elements, in basic machine units, of each of the uniforms specified by the corresponding array of *uniformIndices* is returned. The stride of a uniform associated with the default uniform block is -1. Note that this information only makes sense for uniforms that are arrays. For uniforms that are not arrays, but are declared in a named uniform block, an array stride of zero is returned.

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

If *pname* is `UNIFORM_IS_ROW_MAJOR`, then an array identifying whether each of the uniforms specified by the corresponding array of *uniformIndices* is a row-major matrix or not is returned. A value of one indicates a row-major matrix, and a value of zero indicates a column-major matrix, a matrix in the default uniform block, or a non-matrix.

Loading Uniform Variables In The Default Uniform Block

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

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

The given values are loaded into the default uniform block uniform variable location identified by *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*ui**{**v**} commands will load *count* sets of one to four unsigned integer values into a uniform location defined as a unsigned integer, an unsigned integer vector, an array of unsigned integers or an array of unsigned integer vectors.

The **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**}, **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**{**if 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.

Uniform Blocks

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

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

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

When a named uniform block is declared by multiple shaders in a program, it must be declared identically in each shader. The uniforms within the block must be declared with the same names and types, and in the same order. If a program contains multiple shaders with different declarations for the same named uniform block differs between shader, the program will fail to link.

Uniform Buffer Object Storage

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

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

vectors, each consisting of C components. The row vectors will be stored in order, with row zero at the lowest offset. The difference in offsets between consecutive rows of the matrix will be referred to as the row stride, and is constant across the matrix. The row stride, `UNIFORM_MATRIX_STRIDE`, is an implementation-dependent value and may be queried after a program is linked.

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

Standard Uniform Block Layout

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

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

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

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

1. If the member is a scalar consuming N basic machine units, the base alignment is N.

2. If the member is a two- or four-component vector with components consuming N basic machine units, the base alignment is $2N$ or $4N$, respectively.
3. If the member is a three-component vector with components consuming N basic machine units, the base alignment is $4N$.
4. If the member is an array of scalars or vectors, the base alignment and array stride are set to match the base alignment of a single array element, according to rules (1), (2), and (3), and rounded up to the base alignment of a vec4. The array may have padding at the end; the base offset of the member following the array is rounded up to the next multiple of the base alignment.
5. If the member is a column-major matrix with C columns and R rows, the matrix is stored identically to an array of C column vectors with R components each, according to rule (4).
6. If the member is an array of S column-major matrices with C columns and R rows, the matrix is stored identically to a row of $S \times C$ column vectors with R components each, according to rule (4).
7. If the member is a row-major matrix with C columns and R rows, the matrix is stored identically to an array of R row vectors with C components each, according to rule (4).
8. If the member is an array of S row-major matrices with C columns and R rows, the matrix is stored identically to a row of $S \times R$ row vectors with C components each, according to rule (4).
9. If the member is a structure, the base alignment of the structure is N, where N is the largest base alignment value of any of its members, and rounded up to the base alignment of a vec4. The individual members of this sub-structure are then assigned offsets by applying this set of rules recursively, where the base offset of the first member of the sub-structure is equal to the aligned offset of the structure. The structure may have padding at the end; the base offset of the member following the sub-structure is rounded up to the next multiple of the base alignment of the structure.
10. If the member is an array of S structures, the S elements of the array are laid out in order, according to rule (9).

Uniform Buffer Object Bindings

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

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

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

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

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

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

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

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

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

2.14.5 Samplers

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

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

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

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

Active samplers are samplers actually being used in a program object. The **LinkProgram** command determines if a sampler is active or not. The **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 `INVALID_OPERATION` error will then be generated.

2.14.6 Varying Variables

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

If a geometry shader is not active, the values of all varying and special variables are expected to be interpolated across the primitive being rendered, unless flatshaded. Otherwise the values of all varying and special variables are collected by the primitive assembly stage and passed on to the geometry shader once enough data for one primitive has been collected (see section 2.15).

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

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

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

Each program object can specify a set of one or more vertex or geometry shader output variables to be recorded in transform feedback mode (see section 2.19). When a geometry shader is active (see section 2.15), transform feedback records the values of the selected geometry shader output variables from the emitted vertices. Otherwise, the values of the selected vertex shader output variables are recorded. The values to record are specified with the command

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

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

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

provides information about the varying variable selected by *index*. An *index* of 0 selects the first varying variable specified in the *varyings* array of **TransformFeedbackVaryings**, and an *index* of `TRANSFORM_FEEDBACK_VARYINGS`-1 selects the last such varying variable. The value of `TRANSFORM_FEEDBACK_VARYINGS` can be queried with **GetProgramiv** (see section 6.1.16). If *index* is greater than or equal to `TRANSFORM_FEEDBACK_VARYINGS`, the error `INVALID_VALUE` is generated. The parameter *program* is the name of a program object for which the command **LinkProgram** has been issued in the past. If *program* has not been linked, the error `INVALID_OPERATION` is generated. If a new set of varying variables is specified by **TransformFeedbackVaryings** after a program object has been linked, the information returned by **GetTransformFeedbackVarying** will not reflect those variables until the program is re-linked.

The name of the selected varying is returned as a null-terminated string in *name*. The actual number of characters written into *name*, excluding the null terminator, is returned in *length*. If *length* is `NULL`, no length is returned. The maximum number of characters that may be written into *name*, including the null terminator, is specified by *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 `gl_`, see the OpenGL Shading Language Specification for a complete list). The length of the longest varying name in *program* is given by `TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH`, which can be queried with **GetProgramiv** (see section 6.1.16).

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

2.14.7 Shader Execution

If a successfully linked program object that contains a vertex shader is made current by calling **UseProgram**, the executable version of the vertex shader is used to process incoming vertex values, rather than the fixed-function vertex processing described in sections 2.12 through 2.13. 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 2.12.1).

- Normals are not transformed to eye coordinates, and are not rescaled or normalized (section 2.12.2).
- Normalization of `AUTO_NORMAL` evaluated normals is not performed. (section 5.1).
- Texture coordinates are not generated automatically (section 2.12.3).
- Per vertex lighting is not performed (section 2.13.1).
- Color material computations are not performed (section 2.13.3).
- Color index lighting is not performed (section 2.13.5).
- All of the above applies when setting the current raster position (section 2.24).

If a geometry shader (see section 2.15) is active, vertices processed by the vertex shader are passed to the geometry shader for further processing. Otherwise, the following operations are applied to vertices processed by the vertex shader:

- Color clamping or masking (section 2.13.6).
- Perspective division on clip coordinates (section 2.16).
- Viewport mapping, including depth range scaling (section 2.16.1).
- Flatshading (section 2.21).
- Clipping, including client-defined clip planes (section 2.22).
- Front face determination (section 2.13.1).
- Color, texture coordinate, fog, point-size and generic attribute clipping (section 2.22.1).
- Final color processing (section 2.23).

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, geometry, or fragment shaders. Also refer to section 3.9 and to section 8.7 of the OpenGL Shading Language Specification,

Texel Fetches

The OpenGL Shading Language texel fetch functions provide the ability to extract a single texel from a specified texture image. The integer coordinates passed to the texel fetch functions are used 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 level of detail is less than the texture's base level ($level_{base}$) or greater than the maximum level ($level_{max}$)
- the computed level of detail is not the texture's base level and the texture's minification filter is `NEAREST` or `LINEAR`
- the layer specified for array textures is negative or greater than the number of layers in the array texture,
- the texel coordinates (i, j, k) refer to a texel outside the defined extents of the specified level of detail, where any of

$$\begin{array}{ll} i < -b_s & i \geq w_s - b_s \\ j < -b_s & j \geq h_s - b_s \\ k < -b_s & k \geq d_s - b_s \end{array}$$

  and the size parameters b_s, w_s, h_s, and d_s refer to the border size, width, height, and depth of the image, as in equation 3.17
- the texture being accessed is not complete, as defined in section 3.9.12.

Multisample Texel Fetches

Multisample buffers do not have mipmaps, and there is no level of detail parameter for multisample texel fetches. Instead, an integer parameter selects the sample number to be fetched from the buffer. The number identifying the sample is the same as the value used to query the sample location using **GetMultisamplefv**. Multisample textures support only `NEAREST` filtering.

Additionally, this fetch may only be performed on a multisample texture sampler. No other sample or fetch commands may be performed on a multisample texture sampler.

Texture Size Query

The OpenGL Shading Language texture size functions provide the ability to query the size of a texture image. The LOD value *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

Shaders have the ability to do a lookup into a texture map. The maximum number of texture image units available to vertex, geometry, or fragment shaders are respectively the values of the implementation-dependent constants `MAX_VERTEX_TEXTURE_IMAGE_UNITS`, `MAX_GEOMETRY_TEXTURE_IMAGE_UNITS`, and `MAX_TEXTURE_IMAGE_UNITS`. The vertex shader, geometry shader, and fragment processing combined cannot use more than the value of `MAX_COMBINED_TEXTURE_IMAGE_UNITS` texture image units. If more than one of the vertex shader, geometry shader, and fragment processing stage access the same texture image unit, each such access counts separately against the `MAX_COMBINED_TEXTURE_IMAGE_UNITS` limit.

When a texture lookup is performed in a vertex or geometry shader, the filtered texture value τ is computed in the manner described in sections 3.9.9 and 3.9.10, and converted to a texture source color C_s according to table 3.25 (section 3.9.15). A four-component vector (R_s, G_s, B_s, A_s) is returned to the shader. Texture lookup functions (see section 8.7 of the OpenGL Shading Language Specification) may return floating-point, signed, or unsigned integer values depending on the function and the internal format of the texture.

In a vertex or geometry shader, it is not possible to perform automatic level-of-detail calculations using partial derivatives of the texture coordinates with respect to window coordinates as described in section 3.9.9. Hence, there is no automatic selection of an image array level. Minification or magnification of a texture map is controlled by a level-of-detail value optionally passed as an argument in the texture lookup functions. If the texture lookup function supplies an explicit level-of-detail value l, then the pre-bias level-of-detail value $\lambda_{base}(x, y) = l$ (replacing equation 3.18). If the texture lookup function does not supply an explicit level-of-detail value, then $\lambda_{base}(x, y) = 0$. The scale factor $\rho(x, y)$ and its approximation function $f(x, y)$ (see equation 3.22) 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 3.9.16. The comparison operation is requested in the shader by using any of the shadow sampler types (`sampler1DShadow`, `sampler2DShadow`, or `sampler2DRectShadow`), and in the texture using the `TEXTURE_COMPARE_MODE` parameter. These requests must be consistent; the results of a texture lookup are undefined if any of the following conditions are true:

- The sampler used in a texture lookup function is not one of the shadow sampler types, the texture object's internal format is `DEPTH_COMPONENT` or `DEPTH_STENCIL`, and the `TEXTURE_COMPARE_MODE` is not `NONE`.
- The sampler used in a texture lookup function is one of the shadow sampler types, the texture object's internal format is `DEPTH_COMPONENT` or `DEPTH_STENCIL`, and the `TEXTURE_COMPARE_MODE` is `NONE`.
- The sampler used in a texture lookup function is one of the shadow sampler types, and the texture object's internal format is not `DEPTH_COMPONENT` or `DEPTH_STENCIL`.

The stencil index texture internal component is ignored if the base internal format is `DEPTH_STENCIL`.

Using a sampler in a vertex or geometry shader will return $(R, G, B, A) = (0, 0, 0, 1)$ if the sampler's associated texture is not complete, as defined in section 3.9.12.

Shader Inputs

Besides having access to vertex attributes and uniform variables, vertex shaders can access the read-only built-in variables `gl_VertexID` and `gl_InstanceID`.

`gl_VertexID` holds the integer index i explicitly passed to **ArrayElement** to specify the vertex, or implicitly passed by **DrawArrays** or one of the other drawing commands defined in section 2.8.1. The value of `gl_VertexID` is defined if and only if:

- the vertex comes from a vertex array command that specifies a complete primitive (a vertex array drawing command other than **ArrayElement**).
- 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.

`gl_InstanceID` holds the integer index of the current primitive in an instanced draw call (see section 2.8.1).

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

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 2.21 and sections 4.3.6, 7.1, and 7.6 of the OpenGL Shading Language Specification 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 variables `gl_ClipVertex` and `gl_ClipDistance` respectively hold the vertex coordinate and clip distance(s) used in the clipping stage, as described in section 2.22. If clipping is enabled, only one of `gl_ClipVertex` and `gl_ClipDistance` should be written.

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

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 `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.

Validation

It is not always possible to determine at link time if a program object actually will execute. Therefore validation is done when the first rendering command is issued, to determine if the currently active program object can be executed. If it cannot be executed then no fragments will be rendered, and the error `INVALID_OPERATION` will be generated.

This error is generated by **Begin**, **RasterPos**, or any command that performs an implicit **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 `INVALID_OPERATION` error reported by these rendering commands may not provide enough information to find out why the currently active program object would not execute. No information at all is available about a program object that would still execute, but is inefficient or suboptimal given the current GL state. As a development aid, use the command

`void` **ValidateProgram**(`uint` *program*);

to validate the program object *program* against the current GL state. Each program object has a boolean status, `VALIDATE_STATUS`, that is modified as a result of validation. This status can be queried with **GetProgramiv** (see section 6.1.16).

If validation succeeded this status will be set to `TRUE`, otherwise it will be set to `FALSE`. If validation succeeded the program object is guaranteed to execute, given the current GL state. If validation failed, the program object is guaranteed to not execute, given the current GL state.

ValidateProgram will check for all the conditions that could lead to an `INVALID_OPERATION` error when rendering commands are issued, and may check for other conditions as well. For example, it could give a hint on how to optimize some piece of shader code. The information log of *program* is overwritten with information on the results of the validation, which could be an empty string. The results written to the information log are typically only useful during application development; an application should not expect different GL implementations to produce identical information.

A shader should not fail to compile, and a program object should not fail to link due to lack of instruction space or lack of temporary variables. Implementations should ensure that all valid shaders and program objects may be successfully compiled, linked and executed.

Undefined Behavior

When using array or matrix variables in a shader, it is possible to access a variable with an index computed at run time that is outside the declared extent of the variable. Such out-of-bounds reads will return undefined values; out-of-bounds writes will have undefined results and could corrupt other variables used by shader or the GL. The level of protection provided against such errors in the shader is implementation-dependent.

2.14.8 Required State

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

The state required per shader object consists of:

- An unsigned integer specifying the shader object name.
- An integer holding the value of `SHADER_TYPE`.
- A boolean holding the delete status, initially `FALSE`.
- A boolean holding the status of the last compile, initially `FALSE`.
- An array of type `char` containing the information log, initially empty.

- An integer holding the length of the information log.
- An array of type `char` containing the concatenated shader string, initially empty.
- An integer holding the length of the concatenated shader string.

The state required per program object consists of:

- An unsigned integer indicating the program object name.
- A boolean holding the delete status, initially `FALSE`.
- A boolean holding the status of the last link attempt, initially `FALSE`.
- A boolean holding the status of the last validation attempt, initally `FALSE`.
- An integer holding the number of attached shader objects.
- A list of unsigned integers to keep track of the names of the shader objects attached.
- An array of type `char` containing the information log, initially empty.
- An integer holding the length of the information log.
- An integer holding the number of active uniforms.
- For each active uniform, three integers, holding its location, size, and type, and an array of type `char` holding its name.
- An array holding the values of each active uniform.
- An integer holding the number of active attributes.
- For each active attribute, three integers holding its location, size, and type, and an array of type `char` holding its name.

Additional state required to support vertex shaders consists of:

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

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

2.15 Geometry Shaders

After vertices are processed, they are arranged into primitives, as described in section 2.6.1. This section describes optional *geometry shaders*, an additional pipeline stage defining operations to further process those primitives. Geometry shaders are defined by source code in the OpenGL Shading Language, in the same manner as vertex shaders. They operate on a single primitive at a time and emit one or more output primitives, all of the same type, which are then processed like an equivalent OpenGL primitive specified by the application. The original primitive is discarded after geometry shader execution. The inputs available to a geometry shader are the transformed attributes of all the vertices that belong to the primitive. Additional *adjacency primitives* are available which also make the transformed attributes of neighboring vertices available to the shader. The results of the shader are a new set of transformed vertices, arranged into primitives by the shader.

The geometry shader pipeline stage is inserted after primitive assembly, prior to transform feedback (section 2.19).

A geometry shader only applies when the GL is in RGBA mode. Its operation in color index mode is undefined.

Geometry shaders are created as described in section 2.14.1 using a *type* of `GEOMETRY_SHADER`. They are attached to and used in program objects as described in section 2.14.2. When the program object currently in use includes a geometry shader, its geometry shader is considered active, and is used to process primitives. If the program object has no geometry shader, or no program object is in use, this stage is bypassed.

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

2.15.1 Geometry Shader Input Primitives

A geometry shader can operate on one of five input primitive types. Depending on the input primitive type, one to six input vertices are available when the shader is executed. Each input primitive type supports a subset of the primitives provided by the GL. If a geometry shader is active, **Begin**, and any command that performs an implicit **Begin**, will generate an `INVALID_OPERATION` error if the primitive *mode* parameter is incompatible with the input primitive type of the currently active program object, as discussed below.

A geometry shader that accesses more input vertices than are available for a given input primitive type can be successfully compiled, because the input primitive type is not part of the shader object. However, a program object containing a shader object that accesses more input vertices than are available for the input primitive type of the program object will not link.

The input primitive type is specified in the geometry shader source code using an input layout qualifier, as described in the OpenGL Shading Language Specification. A program will fail to link if the input primitive type is not specified by any geometry shader object attached to the program, or if it is specified differently by multiple geometry shader objects. The input primitive type may be queried by calling **GetProgramiv** with the symbolic constant `GEOMETRY_INPUT_TYPE`. The supported types and the corresponding OpenGL Shading Language input layout qualifier keywords are:

Points (`points`)
Geometry shaders that operate on points are valid only for the `POINTS` primitive type. There is only a single vertex available for each geometry shader invocation.

Lines (`lines`)
Geometry shaders that operate on line segments are valid only for the `LINES`, `LINE_STRIP`, and `LINE_LOOP` primitive types. There are two vertices available for each geometry shader invocation. The first vertex refers to the vertex at the beginning of the line segment and the second vertex refers to the vertex at the end of the line segment. See also section 2.15.4.

Lines with Adjacency (`lines_adjacency`)
Geometry shaders that operate on line segments with adjacent vertices are valid only for the `LINES_ADJACENCY` and `LINE_STRIP_ADJACENCY` primitive types. There are four vertices available for each program invocation. The second vertex refers to attributes of the vertex at the beginning of the line segment and the third vertex refers to the vertex at the end of the line segment. The first and fourth vertices refer to the vertices adjacent to the beginning and end of the line segment, respectively.

Triangles (`triangles`)
Geometry shaders that operate on triangles are valid for the `TRIANGLES`, `TRIANGLE_STRIP` and `TRIANGLE_FAN` primitive types. There are three vertices available for each program invocation. The first, second and third vertices refer to attributes of the first, second and third vertex of the triangle, respectively.

Triangles with Adjacency (`triangles_adjacency`)
Geometry shaders that operate on triangles with adjacent vertices are valid for the `TRIANGLES_ADJACENCY` and `TRIANGLE_STRIP_ADJACENCY` primitive

types. There are six vertices available for each program invocation. The first, third and fifth vertices refer to attributes of the first, second and third vertex of the triangle, respectively. The second, fourth and sixth vertices refer to attributes of the vertices adjacent to the edges from the first to the second vertex, from the second to the third vertex, and from the third to the first vertex, respectively.

2.15.2 Geometry Shader Output Primitives

A geometry shader can generate primitives of one of three types. The supported output primitive types are points (`POINTS`), line strips (`LINE_STRIP`), and triangle strips (`TRIANGLE_STRIP`). The vertices output by the geometry shader are assembled into points, lines, or triangles based on the output primitive type in the manner described in section 2.6.1. The resulting primitives are then further processed as described in section 2.15.4. If the number of vertices emitted by the geometry shader is not sufficient to produce a single primitive, nothing is drawn. The number of vertices output by the geometry shader is limited to a maximum count specified in the shader.

The output primitive type and maximum output vertex count are specified in the geometry shader source code using an output layout qualifier, as described in section 4.3.8.1 of the OpenGL Shading Language Specification. A program will fail to link if either the output primitive type or maximum output vertex count are not specified by any geometry shader object attached to the program, or if they are specified differently by multiple geometry shader objects. The output primitive type and maximum output vertex count of a linked program may be queried by calling **GetProgramiv** with the symbolic constants `GEOMETRY_OUTPUT_TYPE` and `GEOMETRY_VERTICES_OUT`, respectively.

2.15.3 Geometry Shader Variables

Geometry shaders can access uniforms belonging to the current program object. The amount of storage available for geometry shader uniform variables is specified by the implementation dependent constant `MAX_GEOMETRY_UNIFORM_COMPONENTS`. This value represents the number of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a geometry shader. A link error will be generated if an attempt is made to utilize more than the space available for geometry shader uniform variables. Uniforms are manipulated as described in section 2.14.4. Geometry shaders also have access to samplers to perform texturing operations, as described in sections 2.14.5 and 3.9.

Geometry shaders can access the transformed attributes of all vertices for their input primitive type using input varying variables. A vertex shader writing to output varying variables generates the values of these input varying variables, including values for built-in as well as user-defined varying variables. Values for any varying variables that are not written by a vertex shader are undefined. Additionally, a geometry shader has access to a built-in variable that holds the ID of the current primitive. This ID is generated by the primitive assembly stage that sits in between the vertex and geometry shader.

Additionally, geometry shaders can write to one or more varying variables for each vertex they output. These values are optionally flatshaded (using the OpenGL Shading Language varying qualifier `flat`) and clipped, then the clipped values interpolated across the primitive (if not flatshaded). The results of these interpolations are available to the fragment shader, if one is active. Geometry shaders can also write to a set of built-in varying variables defined in the OpenGL Shading Language, corresponding to the values required for fixed-function processing that occurs after geometry processing.

2.15.4 Geometry Shader Execution Environment

If a successfully linked program object that contains a geometry shader is made current by calling **UseProgram**, the executable version of the geometry shader is used to process primitives resulting from the primitive assembly stage.

The following operations are applied to the primitives that are the result of executing a geometry shader:

- Color clamping or masking (section 2.13.6).
- Perspective division on clip coordinates (section 2.16).
- Viewport mapping, including depth range scaling (section 2.16.1).
- Flatshading (section 2.21).
- Clipping, including client-defined clip planes (section 2.22).
- Front face determination (section 2.13.1).
- Color, texture coordinate, fog, point-size and generic attribute clipping (section 2.22.1).
- Final color processing (section 2.23).

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

Texture Access

The **Shader Only Texturing** subsection of section 2.14.7 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to geometry shaders.

Geometry Shader Inputs

Section 7.1 of the OpenGL Shading Language Specification describes the built-in variable array `gl_in[]` available as input to a geometry shader. `gl_in[]` receives values from equivalent built-in output variables written by the vertex shader, and each array element of `gl_in[]` is a structure holding values for a specific vertex of the input primitive. The length of `gl_in[]` is determined by the geometry shader input type (see section 2.15.1). The members of each element of the `gl_in[]` array are:

- Structure member `gl_ClipDistance[]` holds the per-vertex array of clip distances, as written by the vertex shader to its built-in output variable `gl_ClipDistance[]`.
- Structure member `gl_ClipVertex` holds the per-vertex position in clip coordinates, as written by the vertex shader to its built-in output variable `gl_ClipVertex`.
- Structure members `gl_FrontColor`, `gl_BackColor`, `gl_FrontSecondaryColor` and `gl_BackSecondaryColor` hold the per-vertex front and back colors of the primary and secondary colors, as written by the vertex shader to the corresponding built-in output variables.
- Structure member `gl_FogFragCoord` holds the per-vertex fog coordinate, as written by the vertex shader to its built-in output variable `gl_FogFragCoord`.
- Structure member `gl_TexCoord[]` holds the per-vertex array of texture coordinates written by the vertex shader to its built-in output varying variable `gl_TexCoord[]`.
- Structure member `gl_PointSize` holds the per-vertex point size written by the vertex shader to its built-in output varying variable `gl_PointSize`. If the vertex shader does not write `gl_PointSize`, the value of `gl_PointSize` is undefined, regardless of the value of the enable `PROGRAM_POINT_SIZE`.

- Structure member `gl_Position` holds the per-vertex position, as written by the vertex shader to its built-in output variable `gl_Position`. Note that writing to `gl_Position` from either the vertex or geometry shader is optional (also see section 7.1 of the OpenGL Shading Language Specification)

Geometry shaders also have available the built-in special variable `gl_PrimitiveIDIn`, which is not an array and has no vertex shader equivalent. It is filled with the number of primitives processed 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 triangle primitive is processed. For triangles drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may eventually be drawn. Restarting a primitive topology using the primitive restart index has no effect on the primitive ID counter.

Similarly to the built-in varying variables, each user-defined input varying variable has a value for each vertex and thus needs to be declared as arrays or inside input blocks declared as arrays. Declaring an array size is optional. If no size is specified, it will be inferred by the linker from the input primitive type. If a size is specified, it must match the number of vertices for the input primitive type; otherwise, a link error will occur. The OpenGL Shading Language doesn't support multi-dimensional arrays; therefore, user-defined geometry shader inputs corresponding to vertex shader outputs declared as arrays must be declared as array members of an input block that is itself declared as an array. See sections 4.3.6 and 7.6 of the OpenGL Shading Language Specification for more information.

Similarly to the limit on vertex shader output components (see section 2.14.6), there is a limit on the number of components of built-in and user-defined input varying variables that can be read by the geometry shader, given by the value of the implementation-dependent constant `MAX_GEOMETRY_INPUT_COMPONENTS`.

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

Geometry Shader Outputs

A geometry shader is limited in the number of vertices it may emit per invocation. The maximum number of vertices a geometry shader can possibly emit is specified in the geometry shader source and may be queried after linking by calling

GetProgramiv with the symbolic constant `GEOMETRY_VERTICES_OUT`. If a single invocation of a geometry shader emits more vertices than this value, the emitted vertices may have no effect.

There are two implementation-dependent limits on the value of `GEOMETRY_VERTICES_OUT`; it may not exceed the value of `MAX_GEOMETRY_OUTPUT_VERTICES`, and the product of the total number of vertices and the sum of all components of all active varying variables may not exceed the value of `MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS`. **LinkProgram** will fail if it determines that the total component limit would be violated.

A geometry shader can write to built-in as well as user-defined varying variables. These values are expected to be interpolated across the primitive it outputs, unless they are specified to be flat shaded. To enable seamlessly inserting or removing a geometry shader from a program object, the rules, names and types of the output built-in varying variables and user-defined varying variables are the same as for the vertex shader. Refer to section 2.14.6, and sections 4.3.6, 7.1, and 7.6 of the OpenGL Shading Language Specification for more detail.

After a geometry shader emits a vertex, all built-in and user-defined output variables are undefined, as described in section 8.10 of the OpenGL Shading Language Specification.

The built-in 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, as 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 2.22.

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

The built-in special variable `gl_PrimitiveID` holds the primitive ID counter read by the fragment shader, replacing the value of `gl_PrimitiveID` generated by drawing commands when no geometry shader is active. The geometry shader must write to `gl_PrimitiveID` for the provoking vertex (see section 2.21) of a primitive being generated, or the primitive ID counter read by the fragment shader for that primitive is undefined.

The built-in special variable `gl_Layer` is used in layered rendering, and discussed further in the next section.

Similarly to the limit on vertex shader output components (see section 2.14.6), there is a limit on the number of components of built-in and user-defined output varying variables that can be written by the geometry shader, given by the value of the implementation-dependent constant `MAX_GEOMETRY_OUTPUT_COMPONENTS`.

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

Layered Rendering

Geometry shaders can be used to render to one of several different layers of cube map textures, three-dimensional textures, or one-or two-dimensional texture arrays. This functionality allows an application to bind an entire complex texture to a framebuffer object, and render primitives to arbitrary layers computed at run time. For example, it can be used to project and render a scene onto all six faces of a cubemap texture in one pass. The layer to render to is specified by writing to the built-in output variable `gl_Layer`. Layered rendering requires the use of framebuffer objects (see section 4.4.7).

Primitive Type Mismatches and Drawing Commands

A geometry shader will fail to execute if a mismatch exists between the type of primitive being drawn and the input primitive type of the shader. If it cannot be executed then no fragments will be rendered, and the error `INVALID_OPERATION` will be generated.

This error is generated by **Begin**, **RasterPos**, or any command that performs an implicit **Begin** if a geometry shader is active and:

- the input primitive type of the current geometry shader is `POINTS` and *mode* is not `POINTS`;
- the input primitive type of the current geometry shader is `LINES` and *mode* is not `LINES`, `LINE_STRIP`, or `LINE_LOOP`;
- the input primitive type of the current geometry shader is `TRIANGLES` and *mode* is not `TRIANGLES`, `TRIANGLE_STRIP` or `TRIANGLE_FAN`;
- the input primitive type of the current geometry shader is `LINES_ADJACENCY` and *mode* is not `LINES_ADJACENCY` or `LINE_STRIP_ADJACENCY`; or,

- the input primitive type of the current geometry shader is `TRIANGLES_ADJACENCY` and *mode* is not `TRIANGLES_ADJACENCY` or `TRIANGLE_STRIP_ADJACENCY`.

2.16 Coordinate Transformations

Clip coordinates for a vertex result from fixed-function transformation of the vertex coordinates, or from vertex or, if active, geometry shader execution, which yields a vertex coordinate `gl_Position`. Perspective division on clip coordinates yields *normalized device coordinates*, followed by a *viewport* transformation to convert these coordinates into *window coordinates*.

If a vertex in clip coordinates is given by $\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix}$ then the vertex's normalized device coordinates are

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

2.16.1 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

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

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

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

z_w is represented as either fixed- or floating-point depending on whether the framebuffer's depth buffer uses a fixed- or floating-point representation. If the depth buffer uses fixed-point, we assume that it represents each value $k/(2^m - 1)$, where $k \in \{0, 1, \ldots, 2^m - 1\}$, as k (e.g. 1.0 is represented in binary as a string of all ones). The parameters *n* and *f* are clamped to the range $[0, 1]$, as are all arguments of type `clampd` or `clampf`.

Viewport transformation parameters are specified using

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

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

$$\begin{aligned} o_x &= x + \tfrac{w}{2} \\ o_y &= y + \tfrac{h}{2} \\ p_x &= w \\ p_y &= h. \end{aligned}$$

Viewport width and height are clamped to implementation-dependent maximums when specified. The maximum width and height may be found by issuing an appropriate **Get** command (see chapter 6). The maximum viewport dimensions must be greater than or equal to the larger of the visible dimensions of the display being rendered to (if a display exists), and the largest renderbuffer image which can be successfully created and attached to a framebuffer object (see chapter 4). `INVALID_VALUE` is generated if either *w* or *h* is negative.

The state required to implement the viewport transformation is four integers and two clamped floating-point values. In the initial state, *w* and *h* are set to the width and height, respectively, of the window into which the GL is to do its rendering. If the default framebuffer is bound but no default framebuffer is associated with the GL context (see chapter 4), then *w* and *h* are initially set to zero. o_x, o_y, n, and f are set to $\frac{w}{2}$, $\frac{h}{2}$, 0.0, and 1.0, respectively.

2.17 Asynchronous Queries

Asynchronous queries provide a mechanism to return information about the processing of a sequence of GL commands. There are three query types supported by the GL. Primitive queries with a target of `PRIMITIVES_GENERATED` (see section 2.20) return information on the number of primitives processed by the GL. Primitive queries with a target of `TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN` (see section 2.20) return information on the number of primitives written to one 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 and made active by calling

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

target indicates the type of query to be performed; valid values of *target* are defined in subsequent sections. If *id* is an unused query object name, the name is marked as used and associated with a new query object of the type specified by *target*. Otherwise *id* must be the name of an existing query object of that type.

BeginQuery sets the active query object name for the query type given by *target* to *id*. If **BeginQuery** is called with an *id* of zero, if the active query object name for *target* is non-zero, if *id* is the name of an existing query object whose type does not match *target*, if *id* is the active query object name for any query type, or if *id* is the active query object for condtional rendering (see section 2.18), the error `INVALID_OPERATION` is generated.

The command

`void` **EndQuery**(`enum` *target*);

marks the end of the sequence of commands to be tracked for the query type given by *target*. The active query object for *target* is updated to indicate that query results are not available, and the active query object name for *target* is reset to zero. When the commands issued prior to **EndQuery** have completed and a final query result is available, the query object active when **EndQuery** is called is updated by the GL. The query object is updated to indicate that the query results are available and to contain the query result. If the active query object name for *target* is zero when **EndQuery** is called, the error `INVALID_OPERATION` is generated.

The command

`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

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.

2.18 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

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 (see section 2.8) performing an implicit **Begin** and **End**, **DrawPixels** (see section 3.7.5), **Bitmap** (see section 3.8), **Accum** (see section 4.2.4), **EvalMesh1** and **EvalMesh2** (see section 5.1), and **CopyPixels** (see section 4.3.3), as well as **Clear** and **ClearBuffer*** (see section 4.2.3), have no effect. The effect of commands setting current vertex state, such as **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 subsquent 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.

2.19 Transform Feedback

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

Transform feedback is started and finished by calling

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

and

`void` **EndTransformFeedback**(`void`);

Transform Feedback *primitiveMode*	Allowed render primitive (**Begin**) *modes*
`POINTS`	`POINTS`
`LINES`	`LINES`, `LINE_LOOP`, `LINE_STRIP`
`TRIANGLES`	`TRIANGLES`, `TRIANGLE_STRIP`, `TRIANGLE_FAN`
	`QUADS`, `QUAD_STRIP`, `POLYGON`

Table 2.14: Legal combinations of the transform feedback primitive mode, as passed to **BeginTransformFeedback**, and the current primitive mode.

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 2.14.

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 or geometry shader. The error `INVALID_OPERATION` is generated by **BeginTransformFeedback** if no vertex or geometry 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 2.14. If a geometry shader is active, its output primitive type is used instead of of the *mode* parameter passed to drawing commands.

Regions of buffer objects are bound as the targets of transform feedback by calling one of the commands **BindBufferRange** or **BindBufferBase** (see section 2.9.1) with *target* set to `TRANSFORM_FEEDBACK_BUFFER`. In addition to the general errors described in section 2.9.1, **BindBufferRange** will generate an `INVALID_VALUE` error if *index* is greater than or equal to the value of `MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`, or if *offset* is not a multiple of 4.

When an individual point, line, or triangle primitive reaches the transform feedback stage while transform feedback is active, the values of the specified varying variables of the vertex are appended to the buffer objects bound to the transform feedback binding points. The attributes of the first vertex received after **Begin-**

TransformFeedback are written at the starting offsets of the bound buffer objects set by **BindBufferRange**, and subsequent vertex attributes are appended to the buffer object. When capturing line and triangle primitives, all attributes of the first vertex are written first, followed by attributes of the subsequent vertices. When writing varying variables that are arrays, individual array elements are written in order. For multi-component varying variables or varying array elements, the individual components are written in order. The value for any attribute specified to be streamed to a buffer object but not actually written by a vertex or geometry shader is undefined.

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 2.14.6). In `SEPARATE_ATTRIBS` mode, the first varying variable specified by **TransformFeedbackVaryings** is written to the first transform feedback binding point; subsequent varying variables are written to the subsequent transform feedback binding points. The total number of variables that may be captured in separate mode is given by `MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`.

If recording the vertices of a primitive to the buffer objects being used for transform feedback purposes would result in either exceeding the limits of any buffer object's size, or in exceeding the end position $offset + size - 1$, as set by **BindBufferRange**, then no vertices of that primitive are recorded in any buffer object, and the counter corresponding to the asynchronous query target `TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN` (see section 2.20) is not incremented.

In either separate or interleaved modes, all transform feedback binding points that will be written to must have buffer objects bound when **BeginTransformFeedback** is called. The error `INVALID_OPERATION` is generated by **BeginTransformFeedback** if any binding point used in transform feedback mode does not have a buffer object bound. In interleaved mode, only the first buffer object binding point is ever written to. The error `INVALID_OPERATION` is also generated by **BeginTransformFeedback** if no binding points would be used, either because no program object is active or because the active program object has specified no varying variables to record.

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

Buffers should not be bound or in use for both transform feedback and other purposes in the GL. Specifically, if a buffer object is simultaneously bound to a transform feedback buffer binding point and elsewhere in the GL, any writes to or reads from the buffer generate undefined values. Examples of such bindings include **DrawPixels** and **ReadPixels** to a pixel buffer object binding point and client access to a buffer mapped with **MapBuffer**.

However, if a buffer object is written and read sequentially by transform feedback and other mechanisms, it is the responsibility of the GL to ensure that data are accessed consistently, even if the implementation performs the operations in a pipelined manner. For example, **MapBuffer** may need to block pending the completion of a previous transform feedback operation.

2.20 Primitive Queries

Primitive queries use query objects to track the number of primitives generated by the GL and to track the number of primitives written to transform feedback buffers.

When **BeginQuery** is called with a *target* of `PRIMITIVES_GENERATED`, the primitives-generated count maintained by the GL is set to zero. When the generated primitive query is active, the primitives-generated count is incremented every time a primitive reaches the "Discarding Primitives Before Rasterization" stage (see section 3.1) immediately before rasterization. This counter counts the number of primitives emitted by a geometry shader, if active, possibly further tessellated into separate primitives during the transform-feedback stage, if enabled.

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

These two queries can be used together to determine if all primitives have been written to the bound feedback buffers; if both queries are run simultaneously and the query results are equal, all primitives have been written to the buffer(s). If the number of primitives written is less than the number of primitives generated, the buffer is full.

2.21 Flatshading

For fixed-function vertex processing, *flatshading* a primitive means to assign all vertices of the primitive the same primary and secondary colors (in RGBA mode) or the same color index (in color index mode). If a vertex shader is active, flatshading a varying output means to assign all vertices of the primitive the same value for that output.

The color and/or varying output values assigned are those of the *provoking vertex* of the primitive. The provoking vertex is controlled with the command

`void` **ProvokingVertex**(`enum` *provokeMode*);

provokeMode must be either `FIRST_VERTEX_CONVENTION` or `LAST_VERTEX_CONVENTION`, and controls selection of the vertex whose values are assigned to flatshaded colors and varying outputs, as shown in table 2.15

The provoking vertex behavior of quad primitives is implementation dependent, and may be determined by calling **GetBooleanv** with the symbolic constant `QUADS_FOLLOW_PROVOKING_VERTEX`. A return value of `TRUE` indicates that the provoking vertex mode is respected for quad primitives, while a return value of `FALSE` indicates that the implementation always behave as though the provoking vertex mode were `LAST_VERTEX_CONVENTION`.

Flatshading of colors in fixed-function vertex processing, and of the built-in varying variables `gl_FrontColor`, `gl_BackColor`, `gl_FrontSecondaryColor` and `gl_BackSecondaryColor` when a vertex shader is active, is controlled with the command

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

mode must be `SMOOTH` or `FLAT`. If *mode* is `SMOOTH`, vertex colors are treated individually. If *mode* is `FLAT`, flatshading is enabled and colors are taken from the provoking vertex of the primitive. The colors selected are those derived from current values, generated by lighting, or generated by vertex shading, if lighting is disabled, enabled, or a vertex shader is in use, respectively.

If a vertex or geometry shader is active, user-defined varying outputs may be flatshaded by using the `flat` qualifier when declaring the ouput, as described in section 4.3.6 of the OpenGL Shading Language Specification

The state required for flatshading is one bit for the shade mode, one bit for the provoking vertex mode, and one implementation-dependent bit for the provoking

Primitive type of polygon i	First vertex convention	Last vertex convention
point	i	i
independent line	$2i - 1$	$2i$
line loop	i	$i + 1$, if $i < n$ 1, if $i = n$
line strip	i	$i + 1$
independent triangle	$3i - 2$	$3i$
triangle strip	i	$i + 2$
triangle fan	$i + 1$	$i + 2$
independent quad	$4i - 3$ $4i$	$4i$ [1] $4i$ [2]
quad strip	$2i - 1$ $2i + 2$	$2i + 2$ [1] $2i + 2$ [2]
single polygon ($i = 1$)	1	1
line adjacency	$4i - 2$	$4i - 1$
line strip adjacency	$i + 1$	$i + 2$
triangle adjacency	$6i - 5$	$6i - 1$
triangle strip adjacency	$2i - 1$	$2i + 3$

Table 2.15: Provoking vertex selection. The vertex colors and/or varying values used for flatshading the ith primitive generated by the indicated **Begin/End** type are derived from the corresponding values of the vertex whose index is shown in the table. Vertices are numbered 1 through n, where n is the number of vertices between the **Begin/End** pair.

[1] If the value of `QUADS_FOLLOW_PROVOKING_VERTEX` is `TRUE`.

[2] If the value of `QUADS_FOLLOW_PROVOKING_VERTEX` is `FALSE`.

vertex behavior of quad primitives. The initial value of the shade mode is `SMOOTH` and the initial value of the provoking vertex mode is `LAST_VERTEX_CONVENTION`.

2.22 Primitive Clipping

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

$$\begin{array}{c} -w_c \leq x_c \leq w_c \\ -w_c \leq y_c \leq w_c \\ -w_c \leq z_c \leq w_c. \end{array}$$

This view volume may be further restricted by as many as n client-defined clip planes to generate the clip volume. Each client-defined plane specifies a half-space. (n is an implementation-dependent maximum that must be at least 8.) The clip volume is the intersection of all such half-spaces with the view volume (if no client-defined clip planes are enabled, the clip volume is the view volume).

A client-defined clip plane is specified with

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

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

(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 $\begin{pmatrix} x_e & y_e & z_e & w_e \end{pmatrix}^T$ that satisfy

$$\begin{pmatrix} p'_1 & p'_2 & p'_3 & p'_4 \end{pmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} \geq 0$$

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 $\begin{pmatrix} x_e & y_e & z_e & w_e \end{pmatrix}^T$ is no longer computed. Instead, the value of the `gl_ClipVertex` built-in variable is used in its place. If `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 `gl_ClipVertex` write a single clip distance for each supported clip plane to elements of the `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 **Enable** command and disabled with the **Disable** command. The value of the argument to either command is `CLIP_DISTANCE`i, where i is an integer between 0 and $n - 1$; specifying a value of i enables or disables the plane equation with index i. The constants obey `CLIP_DISTANCE`i $=$ `CLIP_DISTANCE0` $+ i$.

Depth clamping is enabled with the generic **Enable** command and disabled with the **Disable** command. The value of the argument to either command is `DEPTH_CLAMP`. If depth clamping is enabled, the

$$-w_c \leq z_c \leq w_c$$

plane equation is ignored by view volume clipping (effectively, there is no near or far plane clipping).

If the primitive under consideration is a point, then clipping passes it unchanged if it lies within the clip volume; otherwise, it is discarded.

If the primitive is a line segment, then clipping does nothing to it if it lies entirely within the clip volume, and discards it if it lies entirely outside the volume.

If part of the line segment lies in the volume and part lies outside, then the line segment is clipped and new vertex coordinates are computed for one or both vertices. A clipped line segment endpoint lies on both the original line segment and the boundary of the clip volume.

This clipping produces a value, $0 \leq t \leq 1$, for each clipped vertex. If the coordinates of a clipped vertex are $\mathbf{P}$ and the original vertices' coordinates are $\mathbf{P}_1$

and $\mathbf{P}_2$, then t is given by

$$\mathbf{P} = t\mathbf{P}_1 + (1-t)\mathbf{P}_2.$$

The value of t is used to clip color, secondary color, texture coordinate, fog coordinate, and vertex shader varying variables as described in section 2.22.1.

If the primitive is a polygon, then it is passed if every one of its edges lies entirely inside the clip volume and either clipped or discarded otherwise. Polygon clipping may cause polygon edges to be clipped, but because polygon connectivity must be maintained, these clipped edges are connected by new edges that lie along the clip volume's boundary. Thus, clipping may require the introduction of new vertices into a polygon. Edge flags are associated with these vertices so that edges introduced by clipping are flagged as boundary (edge flag `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 user-defined clip planes must satisfy a complementarity criterion. Suppose a single clip plane with coefficients $\begin{pmatrix} p'_1 & p'_2 & p'_3 & p'_4 \end{pmatrix}$ (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 $\begin{pmatrix} -p'_1 & -p'_2 & -p'_3 & -p'_4 \end{pmatrix}$ (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 8 bits indicating which of the client-defined plane equations are enabled, and at least 8 corresponding sets of plane equations (each consisting of four double-precision floating-point coefficients) In the initial state, all plane equations are disabled and all client-defined plane equation coefficients are zero.

2.22.1 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.

Let the colors assigned to the two vertices $\mathbf{P}_1$ and $\mathbf{P}_2$ of an unclipped edge be $\mathbf{c}_1$ and $\mathbf{c}_2$. The value of t (section 2.22) for a clipped point $\mathbf{P}$ is used to obtain the color associated with $\mathbf{P}$ as

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

(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 2.14.6), 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 perspective correction (using the `noperspective` qualifier), the value of t used to obtain the varying value associated with $\mathbf{P}$ will be adjusted to produce results that vary linearly in screen space.

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

2.23 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 may be converted to a signed or unsigned normalized fixed-point value as described in equations 2.4 and 2.6 (depending on the framebuffer format).

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 **Colorub**, **Colorus**, or **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 **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.

2.24 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

`void` **RasterPos**{**234**}{**sifd**}(`T` *coords*);
`void` **RasterPos**{**234**}{**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 `FOG_COORD`, 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.

If depth clamping (see section 2.22) is enabled, then raster position z_w is first clamped to the range $[min(n, f), max(n, f)]$, where n and f are the current near and far depth range values (see section 2.16.1).

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 2.16) 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 2.13 summarizes the behavior of the current raster position.

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

void **WindowPos{23}{sifd}**(T *coords*);
void **WindowPos{23}{sifd}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:

$$x_w = x$$

$$y_w = y$$

$$z_w = \begin{cases} n, & z \le 0 \\ f, & z \ge 1 \\ n + z(f - n), & otherwise \end{cases}$$

$$w_c = 1$$

where n and f are the values passed to **DepthRange** (see section 2.16.1).

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.

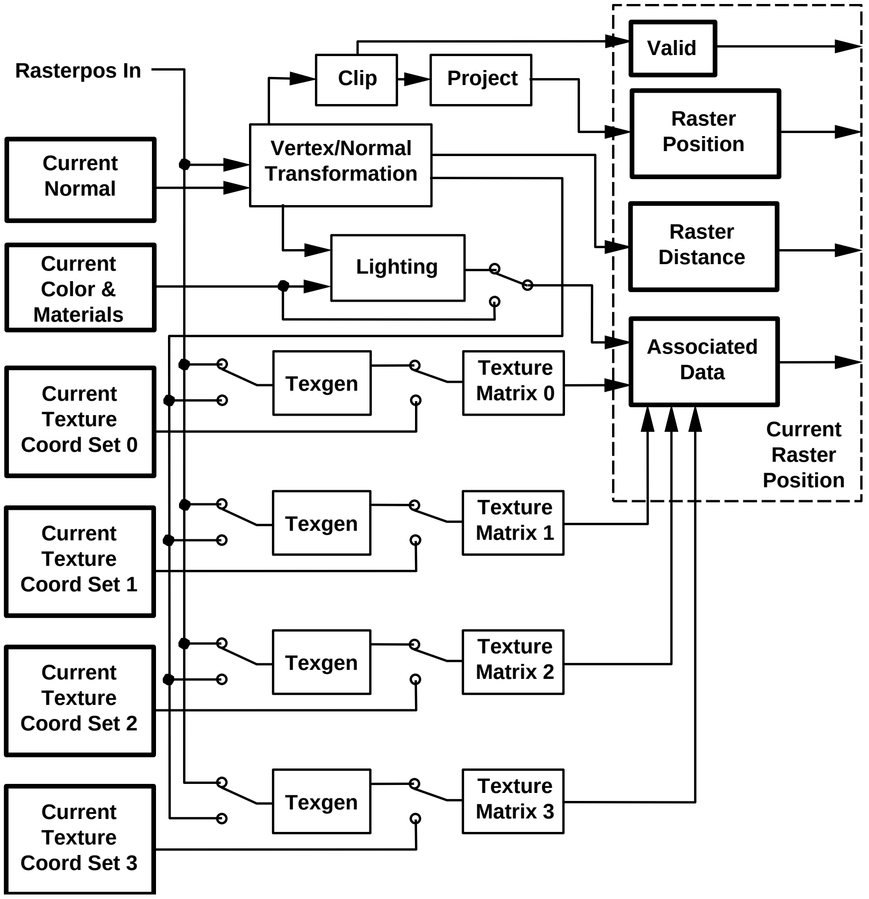


Figure 2.13. The current raster position and how it is set. Four texture units are shown; however, multitexturing may support a different number of units depending on the implementation.

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, and 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 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.

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 z (depth) and assigned colors, fog coordinate, and texture coordinates, or varying shader output parameters is called a *fragment*; the parameters are collectively dubbed the fragment's *associated data*. A fragment is located by its lower left corner, which lies on integer grid coordinates. Rasterization operations also refer to a fragment's *center*, which is offset by $(1/2, 1/2)$ from its lower left corner (and so lies on half-integer coordinates).

Grid squares need not actually be square in the GL. Rasterization rules are not affected by the actual aspect ratio of the grid squares. Display of non-square grids, however, will cause rasterized points and line segments to appear fatter in one direction than the other. We assume that fragments are square, since it simplifies antialiasing and texturing.

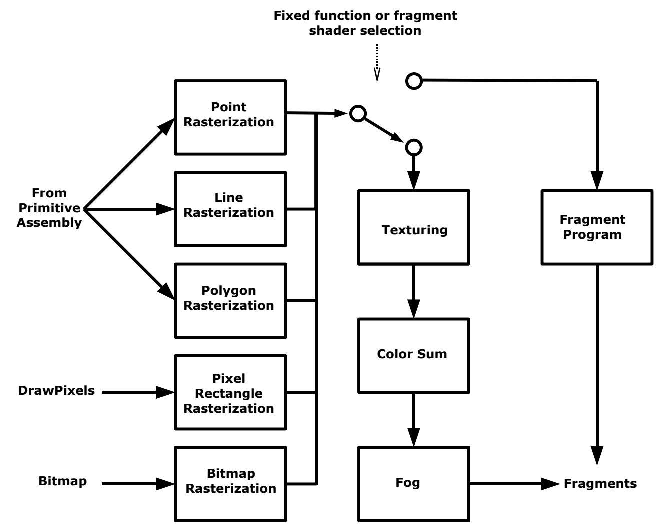


Figure 3.1. Rasterization.

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 2.19). When disabled, primitives are passed through to the rasterization stage to be processed normally. When enabled, `RASTERIZER_DISCARD` also causes the **Accum**, **Bitmap**, **CopyPixels**, **DrawPixels**, **Clear**, and **ClearBuffer*** commands to be ignored.

3.2 Invariance

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

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 into account perceptual issues as well as characteristics of the monitor on which the contents of the framebuffer are displayed. Such details cannot be addressed within the scope of this document. Further, the coverage value computed for a fragment of some primitive may depend on the primitive's relationship to a number of grid squares neighboring the one corresponding to the fragment, and not just on the fragment's grid square. Another consideration is that accurate calculation of coverage values may be computationally expensive; consequently we allow a given GL implementation to approximate true coverage values by using a fast but not entirely accurate coverage computation.

In light of these considerations, we chose to specify the behavior of exact antialiasing in the prototypical case that each displayed pixel is a perfect square of uniform intensity. The square is called a *fragment square* and has lower left corner (x, y) and upper right corner $(x+1, y+1)$. We recognize that this simple box filter may not produce the most favorable antialiasing results, but it provides a simple, well-defined model.

A GL implementation may use other methods to perform antialiasing, subject to the following conditions:

1. If f_1 and f_2 are two fragments, and the portion of f_1 covered by some primitive is a subset of the corresponding portion of f_2 covered by the primitive, then the coverage computed for f_1 must be less than or equal to that computed for f_2.

2. The coverage computation for a fragment f must be local: it may depend only on f's relationship to the boundary of the primitive being rasterized. It may not depend on f's x and y coordinates.

Another property that is desirable, but not required, is:

3. The sum of the coverage values for all fragments produced by rasterizing a particular primitive must be constant, independent of any rigid motions in window coordinates, as long as none of those fragments lies along window edges.

In some implementations, varying degrees of antialiasing quality may be obtained by providing GL hints (section 5.7), 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, polygons, bitmaps, and images. The technique is to sample all primitives multiple times at each pixel. The color sample values are resolved to a single, displayable color each time a pixel is updated, so the antialiasing appears to be automatic at the application level. Because each sample includes color, depth, and stencil information, the color (including texture operation), depth, and stencil functions perform equivalently to the single-sample mode.

An additional buffer, called the multisample buffer, is added to the framebuffer. Pixel sample values, including color, depth, and stencil values, are stored in this buffer. Samples contain separate color values for each fragment color. When the framebuffer includes a multisample buffer, it does not include depth or stencil buffers, even if the multisample buffer does not store depth or stencil values. Color buffers do coexist with the multisample buffer, however.

Multisample antialiasing is most valuable for rendering polygons, because it requires no sorting for hidden surface elimination, and it correctly handles adjacent polygons, object silhouettes, and even intersecting polygons. If only 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`.

The location of a given sample is queried with the command

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

pname must be `SAMPLE_POSITION`, and *index* corresponds to the sample for which the location should be returned. The sample location is returned as two floating point values in *val[0]* and *val[1]*, each between 0 and 1, corresponding to the x and y locations respectively in GL pixel space of that sample. $(0.5, 0.5)$ thus corresponds to the pixel center. The error `INVALID_VALUE` is generated if *index* is greater than or equal to the value of `SAMPLES`. If the multisample mode does not have fixed sample locations, the returned values may only reflect the locations of samples within some pixels.

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

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

If `MULTISAMPLE` is disabled, multisample rasterization of all primitives is equivalent to single-sample (fragment-center) rasterization, except that the fragment coverage value is set to full coverage. The color and depth values and the sets of texture coordinates may all be set to the values that would have been assigned by single-sample rasterization, or they may be assigned as described below for multisample rasterization.

If `MULTISAMPLE` is enabled, multisample rasterization of all primitives differs substantially from single-sample rasterization. It is understood that each pixel in the framebuffer has `SAMPLES` locations associated with it. These locations are exact positions, rather than regions or areas, and each is referred to as a sample point. The sample points associated with a pixel may be located inside or outside of the unit square that is considered to bound the pixel. Furthermore, the relative locations of sample points may be identical for each pixel in the framebuffer, or they may differ.

If the sample locations differ per pixel, they should be aligned to window, not screen, boundaries. Otherwise rendering results will be window-position specific. The invariance requirement described in section 3.2 is relaxed for all multisample rasterization, because the sample locations may be a function of pixel location.

3.4 Points

A point is drawn by generating a set of fragments in the shape of a square or circle centered around the vertex of the point. Each vertex has an associated point size that controls the size of that square or circle.

If no vertex or geometry shader is active, then the rasterization of points is controlled with

```
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:

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

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.

If a vertex or geometry shader is active and point size mode is enabled, then the derived point size is taken from the (potentially clipped) shader built-in `gl_PointSize` written by the geometry shader, or written by the vertex shader if no geometry shader is active, and clamped to the implementation-dependent point size range. If the value written to `gl_PointSize` is less than or equal to zero, results are undefined. If a vertex and/or geometry shader is active and 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. Program point size mode is enabled and disabled by calling **Enable** or **Disable** with the symbolic value `PROGRAM_POINT_SIZE`.

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

$$width = \begin{cases} derived\_size & derived\_size \geq threshold \\ threshold & otherwise \end{cases} \tag{3.1}$$

and the fade factor is computed as follows:

$$fade = \begin{cases} 1 & derived\_size \geq threshold \\ \left(\frac{derived\_size}{threshold}\right)^2 & otherwise \end{cases} \tag{3.2}$$

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

```
void PointParameter{if}( enum pname, T param );
void PointParameter{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 disabled. When point sprites are enabled, the state of the point antialiasing enable is ignored. In a deprecated context, point sprites are always enabled.

The point sprite texture coordinate replacement mode is set with one of the **TexEnv*** commands described in section 3.9.15, 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 coordinate 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 antialiasing 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 clamping 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

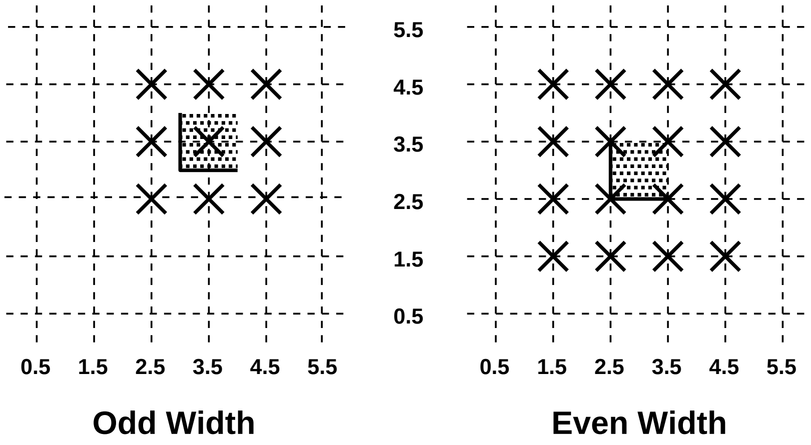


Figure 3.2. Rasterization of non-antialiased wide points. The crosses show fragment centers produced by rasterization for any point that lies within the shaded region. The dotted grid lines lie on half-integer coordinates.

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 centered 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

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

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.

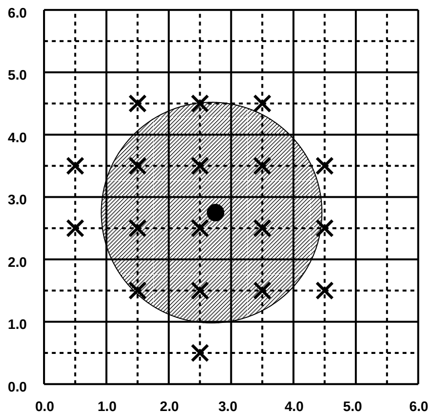


Figure 3.3. Rasterization of antialiased wide points. The black dot indicates the point to be rasterized. The shaded region has the specified width. The X marks indicate those fragment centers produced by rasterization. A fragment's computed coverage value is based on the portion of the shaded region 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 $0.1, 0.2, \ldots, 1.9, 2.0$ 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, the fragment shader builtin `gl_PointCoord` contains point sprite texture coordinates. Additionally, for each texture coordinate set where `COORD_REPLACE` is `TRUE`, these texture coordinates are replaced with point sprite texture coordinates.

The s point sprite texture coordinate varies from 0 to 1 across the point horizontally left-to-right. If `POINT_SPRITE_COORD_ORIGIN` is `LOWER_LEFT`, the t coordinate varies from 0 to 1 vertically bottom-to-top. Otherwise if the point sprite texture coordinate origin is `UPPER_LEFT`, the t coordinate varies from 0 to 1 vertically top-to-bottom. The r and q coordinates are replaced with the constants 0 and 1, respectively.

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

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

$$t = \begin{cases} \frac{1}{2} + \frac{(y_f + \frac{1}{2} - y_w)}{size}, \texttt{POINT\_SPRITE\_COORD\_ORIGIN} = \texttt{LOWER\_LEFT} \\ \frac{1}{2} - \frac{(y_f + \frac{1}{2} - y_w)}{size}, \texttt{POINT\_SPRITE\_COORD\_ORIGIN} = \texttt{UPPER\_LEFT} \end{cases} \quad (3.4)$$

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, two floating-point values specifying the minimum and maximum point size, three floating-point values specifying the distance attenuation coefficients, a bit indicating whether or not antialiasing is enabled, a bit indicating whether or not point sprites are enabled, a bit for the point sprite texture coordinate replacement mode for each texture coordinate set, a bit indicating whether or not vertex program point size mode is enabled, a bit for the point sprite texture coordinate origin, and a floating-point value specifying the point fade threshold size.

3.4.3 Point Multisample Rasterization

If `MULTISAMPLE` is enabled, and the value of `SAMPLE_BUFFERS` is one, then points are rasterized using the following algorithm, regardless of whether point antialiasing (`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 `POINT_SPRITE` is disabled, or a square with side equal to the current point width if `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 `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 `POINT_SPRITE` is disabled. The set of point sizes supported is equivalent to those for point sprites without multisample when `POINT_SPRITE` is enabled.

3.5 Line Segments

A line segment results from a line strip **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

`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 $\mathbf{p}_a$ and ending at $\mathbf{p}_b$ produces those fragments f for which the segment intersects R_f, except if $\mathbf{p}_b$ 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 $\mathbf{p}_a$ and $\mathbf{p}_b$ have window coordinates (x_a, y_a) and (x_b, y_b), respectively. Obtain the perturbed endpoints $\mathbf{p}'_a$ given by $(x_a, y_a) - (\epsilon, \epsilon^2)$ and $\mathbf{p}'_b$ given by $(x_b, y_b) - (\epsilon, \epsilon^2)$. Rasterizing the line segment starting at $\mathbf{p}_a$ and ending at $\mathbf{p}_b$ produces those fragments f for which the segment starting at $\mathbf{p}'_a$ and ending on $\mathbf{p}'_b$ intersects R_f, except if $\mathbf{p}'_b$ is contained in

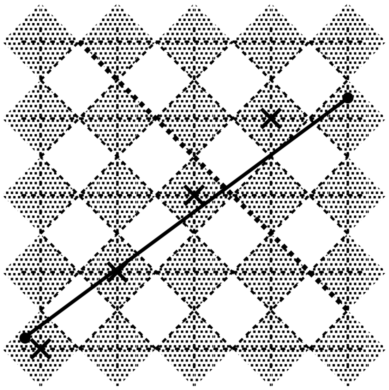


Figure 3.4. Visualization of Bresenham's algorithm. A portion of a line segment is shown. A diamond shaped region of height 1 is placed around each fragment center; those regions that the line segment exits cause rasterization to produce corresponding fragments.

R_f. ϵ is chosen to be so small that rasterizing the line segment produces the same fragments when δ is substituted for ϵ for any $0 < \delta \leq \epsilon$.

When $\mathbf{p}_a$ and $\mathbf{p}_b$ 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 $\mathbf{p}_b$) is not drawn. This means that when rasterizing a series of connected line segments, shared endpoints will be produced only once rather than twice (as would occur with Bresenham's algorithm).

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

1. The coordinates of a fragment produced by the algorithm may not deviate by more than one unit in either x or y window coordinates from a corresponding fragment produced by the diamond-exit rule.

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

3. For an x-major line, no two fragments may be produced that lie in the same

window-coordinate column (for a y-major line, no two fragments may appear in the same row).

4. If two line segments share a common endpoint, and both segments are either x-major (both left-to-right or both right-to-left) or y-major (both bottom-to-top or both top-to-bottom), then rasterizing both segments may not produce duplicate fragments, nor may any fragments be omitted so as to interrupt continuity of the connected segments.

Next we must specify how the data associated with each rasterized fragment are obtained. Let the window coordinates of a produced fragment center be given by $\mathbf{p}_r = (x_d, y_d)$ and let $\mathbf{p}_a = (x_a, y_a)$ and $\mathbf{p}_b = (x_b, y_b)$. Set

$$t = \frac{(\mathbf{p}_r - \mathbf{p}_a) \cdot (\mathbf{p}_b - \mathbf{p}_a)}{\|\mathbf{p}_b - \mathbf{p}_a\|^2}. \tag{3.5}$$

(Note that $t = 0$ at $\mathbf{p}_a$ and $t = 1$ at $\mathbf{p}_b$.) 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

$$f = \frac{(1-t)f_a/w_a + tf_b/w_b}{(1-t)/w_a + t/w_b} \tag{3.6}$$

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

$$z = (1-t)z_a + tz_b \tag{3.7}$$

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

When using a vertex shader, the `noperspective` and `flat` keywords used to declare varying shader outputs affect how they are interpolated. When neither keyword is specified, interpolation is performed as described in equation 3.6. When the `noperspective` keyword is specified, interpolation is performed in the same fashion as for depth values, as described in equation 3.7. When the `flat` keyword is specified, no interpolation is performed, and varying outputs are taken from the corresponding varying value of the provoking vertex corresponding to that primitive (see section 2.21).

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 $FFFF_{16}$. We now describe the rasterization of line segments for general values of the line segment rasterization parameters.

Line Stipple

The command

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

defines a *line stipple*. *pattern* is an unsigned short integer. The *line stipple* is taken from the lowest order 16 bits of *pattern*. It determines those fragments that are to be drawn when the line is rasterized. *factor* is a count that is used to modify the effective line stipple by causing each bit in *pattern* to be used *factor* times. *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

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

Then a fragment is produced if the bth 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,

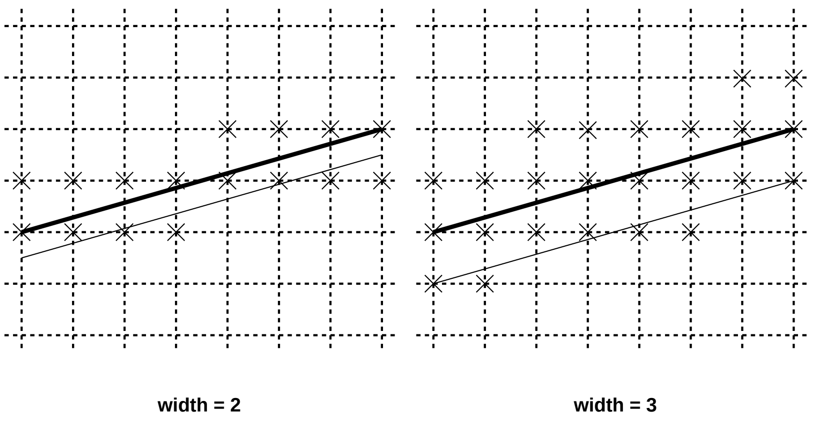


Figure 3.5. Rasterization of non-antialiased wide lines. x-major line segments are shown. The heavy line segment is the one specified to be rasterized; the light segment is the offset segment used for rasterization. x marks indicate the fragment centers produced by rasterization.

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.

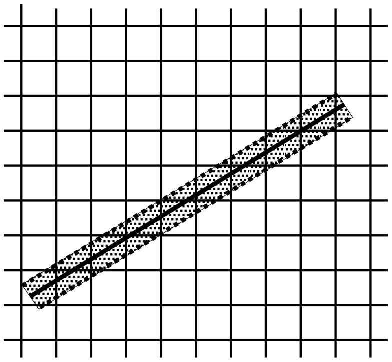


Figure 3.6. The region used in rasterizing and finding corresponding coverage values for an antialiased line segment (an x-major line segment is shown).

Antialiasing

Rasterized antialiased line segments produce fragments whose fragment squares intersect a rectangle centered on the line segment. Two of the edges are parallel to the specified line segment; each is at a distance of one-half the current width from that segment: one above the segment and one below it. The other two edges pass through the line endpoints and are perpendicular to the direction of the specified line segment. Coverage values are computed for each fragment by computing the area of the intersection of the rectangle with the fragment square (see figure 3.6; 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 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 bit indicating whether line antialiasing is on or off, a 16-bit line stipple, the line stipple repeat count, and a bit indicating whether stippling is enabled or disabled. 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 state of line segment antialiasing is disabled. 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.

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 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 depth value and set of associated data is produced by substituting the corresponding sample location into equation 3.5, then using the result to evaluate equation 3.7. An implementation may choose to assign the associated data to more than one sample by evaluating equation 3.5 at any location within the pixel including the fragment center or any one of the sample locations, then substituting into equation 3.6. The different associated data values need not be evaluated at the same location.

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

3.6 Polygons

A polygon results from a triangle arising from a triangle strip, triangle fan, or series of separate triangles, a polygon **Begin/End** object, or a quadrilateral arising from a quadrilateral strip, series of separate quadrilaterals, or **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 based on the sign of the (clipped or unclipped) polygon's area computed in window coordinates. One way to compute this area is

$$a = \frac{1}{2}\sum_{i=0}^{n-1} x_w^i y_w^{i\oplus 1} - x_w^{i\oplus 1} y_w^i \tag{3.8}$$

where x_w^i and y_w^i are the x and y window coordinates of the ith vertex of the n-vertex polygon (vertices are numbered starting at zero for purposes of this computation) and $i \oplus 1$ is $(i+1) \bmod n$. 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) uses a as computed above. Setting *dir* to `CW` (corresponding to clockwise orientation) indicates that the sign of a should be reversed prior to use. Front face determination requires one bit of state, and is initially set to `CCW`.

If the sign of a (including the possible reversal of this sign as determined by **FrontFace**) is positive, the polygon is front-facing; otherwise, it is back-facing. This determination is used in conjunction with the **CullFace** enable bit and mode value to decide whether or not a particular polygon is rasterized. The **CullFace** mode is set by calling

`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

$$a = \frac{\mathrm{A}(pp_bp_c)}{\mathrm{A}(p_ap_bp_c)}, \quad b = \frac{\mathrm{A}(pp_ap_c)}{\mathrm{A}(p_ap_bp_c)}, \quad c = \frac{\mathrm{A}(pp_ap_b)}{\mathrm{A}(p_ap_bp_c)},$$

where $\mathrm{A}(lmn)$ 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

$$f = \frac{af_a/w_a + bf_b/w_b + cf_c/w_c}{a/w_a + b/w_b + c/w_c} \tag{3.9}$$

where w_a, w_b and w_c are the clip w coordinates of p_a, p_b, and p_c, respectively. a, b, and c are the barycentric coordinates of the fragment for which the data are produced. a, b, and c must correspond precisely to the exact coordinates of the center of the fragment. Another way of saying this is that the data associated with a fragment must be sampled at the fragment's center. However, depth values for polygons must be interpolated by

$$z = az_a + bz_b + cz_c \tag{3.10}$$

where z_a, z_b, and z_c are the depth values of p_a, p_b, and p_c, respectively.

When using a vertex shader, the `noperspective` and `flat` keywords used to declare varying shader outputs affect how they are interpolated. When neither keyword is specified, interpolation is performed as described in equation 3.9. When the `noperspective` keyword is specified, interpolation is performed in the same fashion as for depth values, as described in equation 3.10. When the `flat` keyword is specified, no interpolation is performed, and varying outputs are taken from the corresponding varying value of the provoking vertex corresponding to that primitive (see section 2.21).

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 3.9 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

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

pattern is a pointer to memory into which a 32 × 32 pattern is packed. The pattern is unpacked from memory according to the procedure given in section 3.7.5 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 $(x_w \bmod 32, y_w \bmod 32)$ 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

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

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

3.6.5 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

`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

$$m = \sqrt{\left(\frac{\partial z_w}{\partial x_w}\right)^2 + \left(\frac{\partial z_w}{\partial y_w}\right)^2} \tag{3.11}$$

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

$$m = \max\left\{\left|\frac{\partial z_w}{\partial x_w}\right|, \left|\frac{\partial z_w}{\partial y_w}\right|\right\}. \tag{3.12}$$

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 $[min, max]$, where min and max are the smallest and largest values obtained by evaluating equation 3.11 or equation 3.12 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

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

The offset value o for a polygon is

$$o = m \times factor + r \times units. \tag{3.13}$$

m is computed as described above. If the depth buffer uses a fixed-point representation, m is a function of depth values in the range $[0, 1]$, and o is applied to depth values in the same range.

Boolean state values `POLYGON_OFFSET_POINT`, `POLYGON_OFFSET_LINE`, and `POLYGON_OFFSET_FILL` determine whether o is applied during the rasterization of polygons in `POINT`, `LINE`, and `FILL` modes. These boolean state values are enabled and disabled as argument values to the commands **Enable** and **Disable**. If `POLYGON_OFFSET_POINT` is enabled, o is added to the depth value of each fragment produced by the rasterization of a polygon in `POINT` mode. Likewise, if `POLYGON_OFFSET_LINE` or `POLYGON_OFFSET_FILL` is enabled, o is added to the depth value of each fragment produced by the rasterization of a polygon in `LINE` or `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 `MULTISAMPLE` is enabled and the value of `SAMPLE_BUFFERS` is one, then polygons are rasterized using the following algorithm, regardless of whether polygon antialiasing (`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 **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 associated datum is produced as described in section 3.6.1, but using the corresponding sample location instead of the fragment center. An implementation may choose to assign the same associated data values to more than one sample by barycentric evaluation using any location within the pixel including the fragment center or one of the sample locations. The color value and the set of texture coordinates need not be evaluated at the same location.

When using a vertex shader, the `noperspective` and `flat` qualifiers affect how varying shader outputs are interpolated in the same fashion as described for for basic polygon rasterization in section 3.6.1.

The rasterization described above applies only to the `FILL` state of **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 specified to the GL using **TexImage*D** (see section 3.9.1) or converted to fragments using the **DrawPixels** command (described in section 3.7.5) Some of the parameters and operations governing the operation of these commands are shared by **CopyPixels** (used to copy pixels from one framebuffer location to another) and **ReadPixels** (used to obtain pixel values from the framebuffer); the discussion of **CopyPixels** and **ReadPixels**, however, is deferred until chapter 4 after the framebuffer has been discussed in detail. Nevertheless, we note in this section when parameters and state pertaining to these commands also pertain to **CopyPixels** or **ReadPixels**.

A number of parameters control the encoding of pixels in buffer object or client memory (for reading and writing) and how pixels are processed before being placed in or after being read from the framebuffer (for reading, writing, and copying). These parameters are set with three commands: **PixelStore**, **PixelTransfer**, and

Parameter Name	Type	Initial Value	Valid Range
`UNPACK_SWAP_BYTES`	boolean	`FALSE`	`TRUE`/`FALSE`
`UNPACK_LSB_FIRST`	boolean	`FALSE`	`TRUE`/`FALSE`
`UNPACK_ROW_LENGTH`	integer	0	$[0, \infty)$
`UNPACK_SKIP_ROWS`	integer	0	$[0, \infty)$
`UNPACK_SKIP_PIXELS`	integer	0	$[0, \infty)$
`UNPACK_ALIGNMENT`	integer	4	1,2,4,8
`UNPACK_IMAGE_HEIGHT`	integer	0	$[0, \infty)$
`UNPACK_SKIP_IMAGES`	integer	0	$[0, \infty)$

Table 3.1: **PixelStore** parameters pertaining to one or more of **DrawPixels**, **ColorTable**, **ColorSubTable**, **ConvolutionFilter1D**, **ConvolutionFilter2D**, **SeparableFilter2D**, **PolygonStipple**, **TexImage1D**, **TexImage2D**, **TexImage3D**, **TexSubImage1D**, **TexSubImage2D**, and **TexSubImage3D**.

PixelMap.

3.7.1 Pixel Storage Modes and Pixel Buffer Objects

Pixel storage modes affect the operation of **TexImage*D**, **TexSubImage*D**, **DrawPixels**, and **ReadPixels** (as well as other commands; see sections 3.6.2 and 3.8) 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 *pname*, T *param*);

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

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.

Parameter Name	Type	Initial Value	Valid Range
`MAP_COLOR`	boolean	`FALSE`	`TRUE`/`FALSE`
`MAP_STENCIL`	boolean	`FALSE`	`TRUE`/`FALSE`
`INDEX_SHIFT`	integer	0	$(-\infty, \infty)$
`INDEX_OFFSET`	integer	0	$(-\infty, \infty)$
x`_SCALE`	float	1.0	$(-\infty, \infty)$
`DEPTH_SCALE`	float	1.0	$(-\infty, \infty)$
x`_BIAS`	float	0.0	$(-\infty, \infty)$
`DEPTH_BIAS`	float	0.0	$(-\infty, \infty)$
`POST_CONVOLUTION_`x`_SCALE`	float	1.0	$(-\infty, \infty)$
`POST_CONVOLUTION_`x`_BIAS`	float	0.0	$(-\infty, \infty)$
`POST_COLOR_MATRIX_`x`_SCALE`	float	1.0	$(-\infty, \infty)$
`POST_COLOR_MATRIX_`x`_BIAS`	float	0.0	$(-\infty, \infty)$

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

The imaging subset is supported only if the `EXTENSIONS` string includes the substring "`GL_ARB_imaging`" Querying `EXTENSIONS` is described in section 6.1.4.

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 3.7.5), **ReadPixels** (section 4.3.2), and **CopyPixels** (section 4.3.3) at the time when one of these commands is executed (which may differ from the time the command is issued). Some pixel transfer modes are set with

`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

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

Map Name	Address	Value	Init. Size	Init. Value
`PIXEL_MAP_I_TO_I`	color idx	color idx	1	0.0
`PIXEL_MAP_S_TO_S`	stencil idx	stencil idx	1	0
`PIXEL_MAP_I_TO_R`	color idx	R	1	0.0
`PIXEL_MAP_I_TO_G`	color idx	G	1	0.0
`PIXEL_MAP_I_TO_B`	color idx	B	1	0.0
`PIXEL_MAP_I_TO_A`	color idx	A	1	0.0
`PIXEL_MAP_R_TO_R`	R	R	1	0.0
`PIXEL_MAP_G_TO_G`	G	G	1	0.0
`PIXEL_MAP_B_TO_B`	B	B	1	0.0
`PIXEL_MAP_A_TO_A`	A	A	1	0.0

Table 3.3: **PixelMap** parameters.

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 as described in equation 2.1 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 $size = 2^n$ or the error `INVALID_VALUE` results. The maximum allowable *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 *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 *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

Table Name	Type

`COLOR_TABLE`
`POST_CONVOLUTION_COLOR_TABLE`
`POST_COLOR_MATRIX_COLOR_TABLE`
	regular

`PROXY_COLOR_TABLE`
`PROXY_POST_CONVOLUTION_COLOR_TABLE`
`PROXY_POST_COLOR_MATRIX_COLOR_TABLE`
	proxy

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.

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

`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 3.7.5), with *height* taken to be 1. The maximum allowable *width* of a table is implementation-dependent, but must be at least 32. The *format*s `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 3.16 or tables 3.17- 3.19, 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 ith 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

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 framebuffer, and portions of existing tables may be respecified.

The command

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,

```
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 **ColorTable** is executed with *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 **ColorTable** called with *target* set to the corresponding regular table name (`COLOR_TABLE` is the regular name corresponding to `PROXY_COLOR_TABLE`, for example), the proxy state values are set exactly as though the regular table were being specified. Calling **ColorTable** with a proxy *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 **GetColorTable**. The error `INVALID_ENUM` is generated if this is attempted.

Convolution Filter Specification

A two-dimensional convolution filter image is specified by calling

`void` **ConvolutionFilter2D**(`enum` *target*, `enum` *internalformat*,
`sizei` *width*, `sizei` *height*, `enum` *format*, `enum` *type*,
`void` **data*);

target must be `CONVOLUTION_2D`. *width*, *height*, *format*, *type*, and *data* specify an image in memory with the same meaning and allowed values as the corresponding parameters to **DrawPixels**. The *format*s `COLOR_INDEX`, `DEPTH_COMPONENT`, `DEPTH_STENCIL`, and `STENCIL_INDEX` and the *type* `BITMAP` are not allowed.

The specified image is extracted 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 two-dimensional `CONVOLUTION_FILTER_SCALE` parameters and biased by the four two-dimensional `CONVOLUTION_FILTER_BIAS` parameters. These parameters are set by calling **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 internal format* specified by (or derived from) *internalformat*, in the same manner as for textures (section 3.9.1). *internalformat* accepts the same values as the corresponding argument of **ColorTable**.

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, starting at zero, and j increases from bottom to top, also starting at zero. Image location i, j is specified by the Nth pixel, counting from zero, where

$$N = i + j * width$$

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

`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

`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 ith 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 **ConvolutionFilter1D**, except that image data are taken from the framebuffer, rather than from client memory. *target* must be `CONVOLUTION_1D`. *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 **ConvolutionFilter1D**, beginning with scaling by `CONVOLUTION_FILTER_SCALE`. Parameters *target*, *internalformat*, and *width* are specified using the same values, with the same meanings, as the equivalent arguments of **ConvolutionFilter2D**. *format* is taken to be `RGBA`.

Calling **CopyConvolutionFilter1D** or **CopyConvolutionFilter2D** 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).

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 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 `COLOR` causes the matrix operations described in section 2.12.1 to apply to the top matrix on the color matrix stack. All matrix operations have the same effect on the color matrix as they do on the other matrices.

Histogram Table Specification

The histogram table is specified with

> `void` **Histogram**(`enum` *target*`,` `sizei` *width*`,`
>   `enum` *internalformat*`,` `boolean` *sink* `);`

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 accomodated 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

`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 Transfer of Pixel Rectangles

The process of transferring 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.

Commands accepting or returning pixel rectangles take the following arguments (as well as additional arguments specific to their function):

format is a symbolic constant indicating what the values in memory represent.

width and *height* are the width and height, respectively, of the pixel rectangle to be transferred.

data refers to the data to be drawn. These data are represented with one of several GL data types, specified by *type*. The correspondence between the *type* token values and the GL data types they indicate is given in table 3.5.

Not all combinations of *format* and *type* are valid. 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. Additional restrictions may be imposed by specific commands.

Unpacking

Data are taken from the currently bound pixel unpack buffer or client memory as a sequence of signed or unsigned bytes (GL data types `byte` and `ubyte`), signed or unsigned short integers (GL data types `short` and `ushort`), signed or unsigned integers (GL data types `int` and `uint`), or floating point values (GL data types `half` and `float`). These elements are grouped into sets of one, two, three, or four values, depending on the *format*, to form a group. Table 3.6 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

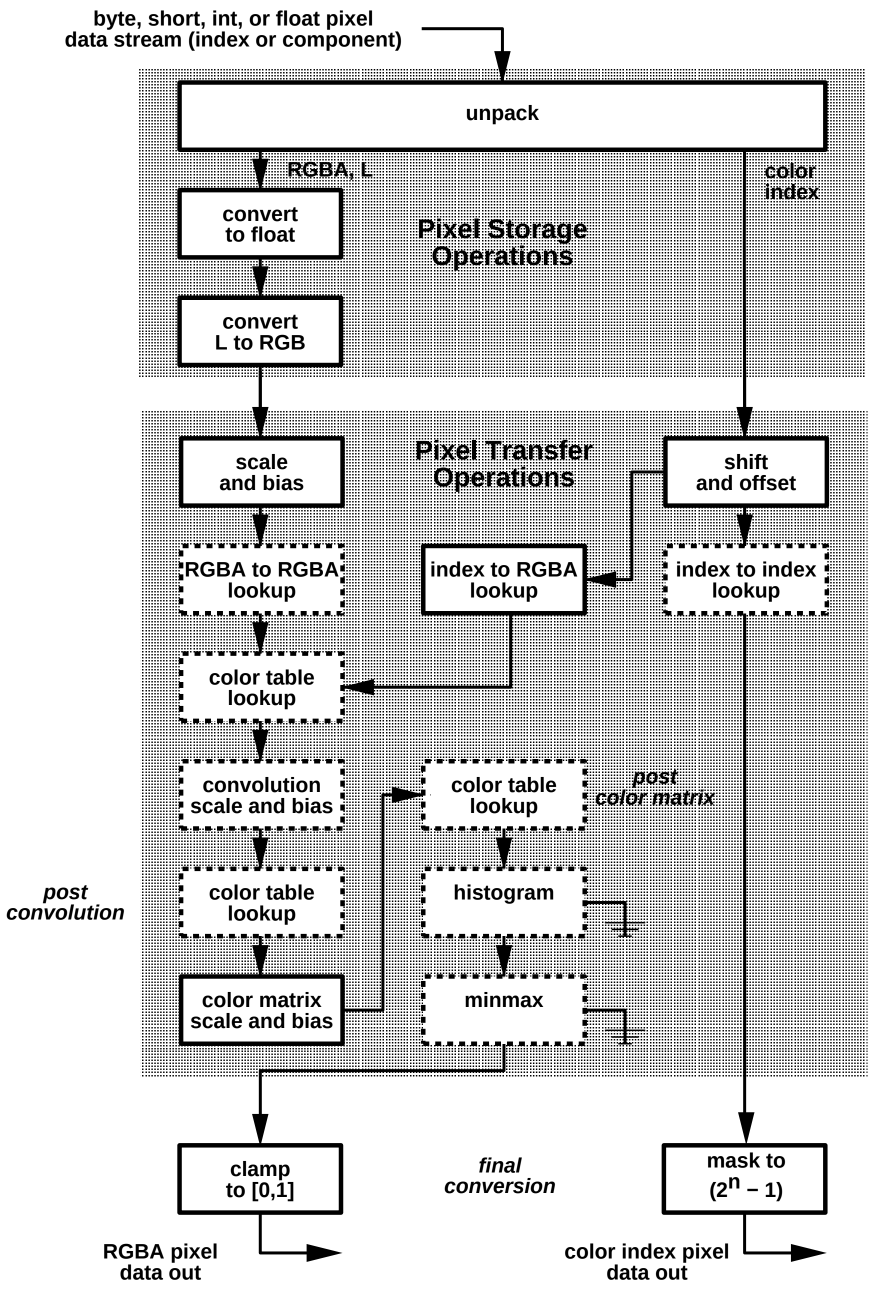


Figure 3.7. Transfer of pixel rectangles to the GL. Output is RGBA pixels if the GL is in RGBA mode, color index pixels otherwise. Operations in dashed boxes may be enabled or disabled. RGBA and color index pixel paths are shown; depth and stencil pixel paths are not shown.

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

Table 3.5: Pixel data *type* parameter values and the corresponding GL data types. Refer to table 2.2 for definitions of GL data types. Special interpretations are described near the end of section 3.8.

Format Name	Element Meaning and Order	Target Buffer
`COLOR_INDEX`	Color Index	Color Index
`STENCIL_INDEX`	Stencil Index	Stencil
`DEPTH_COMPONENT`	Depth	Depth
`DEPTH_STENCIL`	Depth and Stencil Index	Depth and Stencil
`RED`	R	Color
`GREEN`	G	Color
`BLUE`	B	Color
`ALPHA`	A	Color
`RG`	R, G	Color
`RGB`	R, G, B	Color
`RGBA`	R, G, B, A	Color
`BGR`	B, G, R	Color
`BGRA`	B, G, R, A	Color
`LUMINANCE`	Luminance	Color
`LUMINANCE_ALPHA`	Luminance, A	Color
`RED_INTEGER`	iR	Color
`GREEN_INTEGER`	iG	Color
`BLUE_INTEGER`	iB	Color
`ALPHA_INTEGER`	iA	Color
`RG_INTEGER`	iR, iG	Color
`RGB_INTEGER`	iR, iG, iB	Color
`RGBA_INTEGER`	iR, iG, iB, iA	Color
`BGR_INTEGER`	iB, iG, iR	Color
`BGRA_INTEGER`	iB, iG, iR, iA	Color

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

Element Size	Default Bit Ordering	Modified Bit Ordering
8 bit	[7..0]	[7..0]
16 bit	[15..0]	[7..0][15..8]
32 bit	[31..0]	[7..0][15..8][23..16][31..24]

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.

of basic machine units needed to store in memory the corresponding GL data type from table 3.5 for the *type* parameter (or not evenly divisible by 4 for *type* `FLOAT_32_UNSIGNED_INT_24_8_REV`, which does not have a corresponding GL data type), an `INVALID_OPERATION` error results.

By default the values of each GL data type are interpreted as they would be specified in the language of the client's GL binding. If `UNPACK_SWAP_BYTES` is enabled, however, then the values are interpreted with the bit orderings modified as per table 3.7. The modified bit orderings are defined only if the GL data type `ubyte` has eight bits, and then for each specific GL data type only if that type is represented with 8, 16, or 32 bits.

The groups in memory are treated as being arranged in a rectangle. This rectangle consists of a series of *rows*, with the first element of the first group of the first row pointed to by *data*. If the value of `UNPACK_ROW_LENGTH` is not positive, then the number of groups in a row is *width*; otherwise the number of groups is `UNPACK_ROW_LENGTH`. If p indicates the location in memory of the first element of the first row, then the first element of the Nth row is indicated by

$$p + Nk \tag{3.14}$$

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

$$k = \begin{cases} nl & s \geq a, \\ a/s \lceil snl/a \rceil & s < a \end{cases} \tag{3.15}$$

where n is the number of elements in a group, l is the number of groups in the row, a is the value of `UNPACK_ALIGNMENT`, and s is the size, in units of GL `ubyte`s, 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:

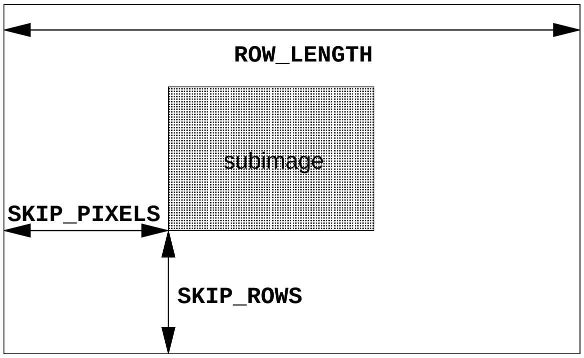


Figure 3.8. Selecting a subimage from an image. The indicated parameter names are prefixed by `UNPACK_` for **DrawPixels** and by `PACK_` for **ReadPixels**.

`UNPACK_ROW_LENGTH`, `UNPACK_SKIP_ROWS`, and `UNPACK_SKIP_PIXELS`. Before obtaining the first group from memory, the *data* pointer is advanced by $(\texttt{UNPACK\_SKIP\_PIXELS})n + (\texttt{UNPACK\_SKIP\_ROWS})k$ 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.

Special Interpretations

A *type* matching one of the types in table 3.8 is a special case in which all the components of each group are packed into a single unsigned byte, unsigned short, or unsigned int, depending on the type. If *type* is `FLOAT_32_UNSIGNED_INT_24_8_REV`, the components of each group are contained within two 32-bit words; the first word contains the float component, and the second word contains a packed 24-bit unused field, followed by an 8-bit component. The number of components per packed pixel is fixed by the type, and must match the number of components per group indicated by the *format* parameter, as listed in table 3.8. The error `INVALID_OPERATION` is generated by any command processing pixel rectangles if a mismatch occurs.

Bitfield locations of the first, second, third, and fourth components of each packed pixel type are illustrated in tables 3.9- 3.12. 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

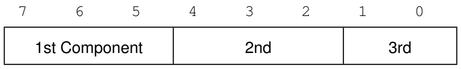
type Parameter Token Name	GL Data Type	Number of Components	Matching Pixel Formats
`UNSIGNED_BYTE_3_3_2`	`ubyte`	3	`RGB`
`UNSIGNED_BYTE_2_3_3_REV`	`ubyte`	3	`RGB`
`UNSIGNED_SHORT_5_6_5`	`ushort`	3	`RGB`
`UNSIGNED_SHORT_5_6_5_REV`	`ushort`	3	`RGB`
`UNSIGNED_SHORT_4_4_4_4`	`ushort`	4	`RGBA`,`BGRA`
`UNSIGNED_SHORT_4_4_4_4_REV`	`ushort`	4	`RGBA`,`BGRA`
`UNSIGNED_SHORT_5_5_5_1`	`ushort`	4	`RGBA`,`BGRA`
`UNSIGNED_SHORT_1_5_5_5_REV`	`ushort`	4	`RGBA`,`BGRA`
`UNSIGNED_INT_8_8_8_8`	`uint`	4	`RGBA`,`BGRA`
`UNSIGNED_INT_8_8_8_8_REV`	`uint`	4	`RGBA`,`BGRA`
`UNSIGNED_INT_10_10_10_2`	`uint`	4	`RGBA`,`BGRA`
`UNSIGNED_INT_2_10_10_10_REV`	`uint`	4	`RGBA`,`BGRA`
`UNSIGNED_INT_24_8`	`uint`	2	`DEPTH_STENCIL`
`UNSIGNED_INT_10F_11F_11F_REV`	`uint`	3	`RGB`
`UNSIGNED_INT_5_9_9_9_REV`	`uint`	4	`RGB`
`FLOAT_32_UNSIGNED_INT_24_8_REV`	n/a	2	`DEPTH_STENCIL`

Table 3.8: Packed pixel formats.

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.

`UNSIGNED_BYTE_3_3_2`:



`UNSIGNED_BYTE_2_3_3_REV`:

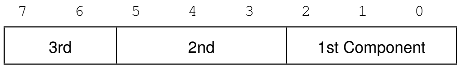


Table 3.9: `UNSIGNED_BYTE` formats. Bit numbers are indicated for each component.

`UNSIGNED_SHORT_5_6_5`:

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">1st Component</td><td colspan="6">2nd</td><td colspan="5">3rd</td></tr>
</table>

`UNSIGNED_SHORT_5_6_5_REV`:

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">3rd</td><td colspan="6">2nd</td><td colspan="5">1st Component</td></tr>
</table>

`UNSIGNED_SHORT_4_4_4_4`:

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="4">1st Component</td><td colspan="4">2nd</td><td colspan="4">3rd</td><td colspan="4">4th</td></tr>
</table>

`UNSIGNED_SHORT_4_4_4_4_REV`:

<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="4">4th</td><td colspan="4">3rd</td><td colspan="4">2nd</td><td colspan="4">1st Component</td></tr>
</table>

`UNSIGNED_SHORT_5_5_5_1`:

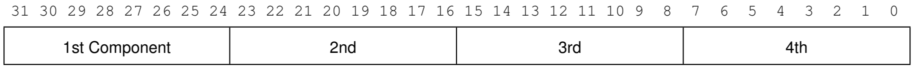
<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">1st Component</td><td colspan="5">2nd</td><td colspan="5">3rd</td><td colspan="1">4th</td></tr>
</table>

`UNSIGNED_SHORT_1_5_5_5_REV`:

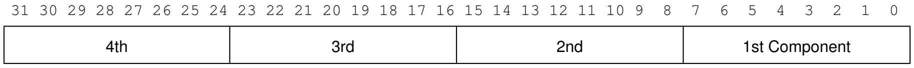
<table>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="1">4th</td><td colspan="5">3rd</td><td colspan="5">2nd</td><td colspan="5">1st Component</td></tr>
</table>

Table 3.10: `UNSIGNED_SHORT` formats

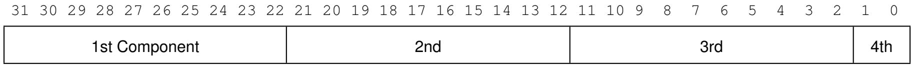
`UNSIGNED_INT_8_8_8_8`:



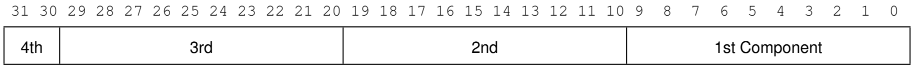
`UNSIGNED_INT_8_8_8_8_REV`:



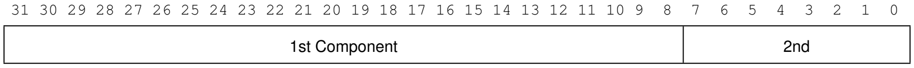
`UNSIGNED_INT_10_10_10_2`:



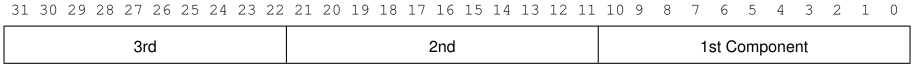
`UNSIGNED_INT_2_10_10_10_REV`:



`UNSIGNED_INT_24_8`:



`UNSIGNED_INT_10F_11F_11F_REV`:



`UNSIGNED_INT_5_9_9_9_REV`:

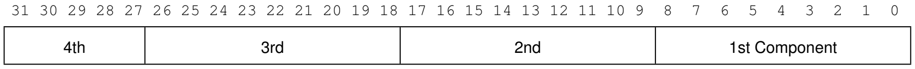


Table 3.11: `UNSIGNED_INT` formats

`FLOAT_32_UNSIGNED_INT_24_8_REV`:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1st Component																															

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Unused																								2nd							

Table 3.12: `FLOAT_UNSIGNED_INT` formats

Format	First Component	Second Component	Third Component	Fourth Component
`RGB`	red	green	blue	
`RGBA`	red	green	blue	alpha
`BGRA`	blue	green	red	alpha
`DEPTH_STENCIL`	depth	stencil		

Table 3.13: Packed pixel field assignments.

The assignment of component to fields in the packed pixel is as described in table 3.13.

Byte swapping, if enabled, is performed before the components are extracted from each pixel. The above discussions of row length and image extraction are valid for packed pixels, if “group” is substituted for “component” and the number of components per group is understood to be one.

A *type* of `UNSIGNED_INT_10F_11F_11F_REV` and *format* of `RGB` is a special case in which the data are a series of GL `uint` values. Each uint value specifies 3 packed components as shown in table 3.11. 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.

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:

$$\begin{aligned} red &= p_{red}2^{p_{exp}-B-N} \\ green &= p_{green}2^{p_{exp}-B-N} \\ blue &= p_{blue}2^{p_{exp}-B-N} \end{aligned}$$

where $B = 15$ (the exponent bias) and $N = 9$ (the number of mantissa bits).

Conversion to floating-point

This step applies only to groups of floating-point components. It is not performed on indices or integer components. For groups containing both components and indices, such as `DEPTH_STENCIL`, the indices are not converted.

Each element in a group is converted to a floating-point value. For unsigned integer elements, equation 2.1 is used. For signed integer elements, equation 2.2 is used unless the final destination of the transferred element is a texture or framebuffer component in one of the `SNORM` formats described in table 3.17, in which case equation 2.3 is used instead.

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 3.7.6. After the processing described in that section is completed, groups are processed as described in the following sections.

3.7.5 Rasterization of Pixel Rectangles

Pixels are drawn using

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

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.

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).

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

$$k = a \left\lceil \frac{l}{8a} \right\rceil \tag{3.16}$$

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` $* k$ `ubyte`s. 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 `ubyte`s. *height* sets of *width* elements are obtained this way.

Once pixels are transferred, **DrawPixels** performs final conversion on pixel values, then converts them to fragments as described below. Fragments generated by **DrawPixels** are then processed in the same fashion as fragments generated by rasterization of a primitive.

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 equation 2.4. If fragment color clamping is disabled, RGBA components are unmodified. Fragment color clamping is controlled by calling

void **ClampColor**(enum *target*, enum *clamp*);

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

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 2.16.1). 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`.

Conversion to Fragments

The conversion of a group to fragments is controlled with

void **PixelZoom**(float z_x, float z_y);

Let (x_{rp}, y_{rp}) be the current raster position (section 2.24). (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 nth in a row and belongs to the mth row, consider the region in window coordinates bounded by the rectangle with corners

$$(x_{rp} + z_x n, y_{rp} + z_y m) \qquad \text{and} \qquad (x_{rp} + z_x(n + 1), y_{rp} + z_y(m + 1))$$

(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.

3.7.6 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.
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.

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: `RED_SCALE` for an R component, `GREEN_SCALE` for a G component, `BLUE_SCALE` for a B component, and `ALPHA_SCALE` for an A component, or `DEPTH_SCALE` for a depth component. Then the result is added to the appropriate signed bias: `RED_BIAS`, `GREEN_BIAS`, `BLUE_BIAS`, `ALPHA_BIAS`, or `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 $\lceil \log_2(\texttt{MAX\_PIXEL\_MAP\_TABLE}) \rceil$ 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 $|\texttt{INDEX\_SHIFT}|$ bits, left if $\texttt{INDEX\_SHIFT} > 0$ and right otherwise. In either case the shift is zero-filled. Then, the signed integer offset `INDEX_OFFSET` is added to the index.

RGBA to RGBA Lookup

This step applies only to RGBA component groups, and is skipped if `MAP_COLOR` is `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: `PIXEL_MAP_R_TO_R` for R, `PIXEL_MAP_G_TO_G` for G, `PIXEL_MAP_B_TO_B` for B, and `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 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 integer index 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.

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 3.14). 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-

Base Internal Format	R	G	B	A
`ALPHA`				A_t
`LUMINANCE`	L_t	L_t	L_t	
`LUMINANCE_ALPHA`	L_t	L_t	L_t	A_t
`INTENSITY`	I_t	I_t	I_t	I_t
`RGB`	R_t	G_t	B_t	
`RGBA`	R_t	G_t	B_t	A_t

Table 3.14: Color table lookup. R_t, G_t, B_t, A_t, L_t, and I_t are color table values that are assigned to pixel components R, G, B, and A depending on the table format. When there is no assignment, the component value is left unchanged by lookup.

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, 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 3.15.

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_{row} and C_{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

Base Filter Format	R	G	B	A
`ALPHA`	R_s	G_s	B_s	$A_s * A_f$
`LUMINANCE`	$R_s * L_f$	$G_s * L_f$	$B_s * L_f$	A_s
`LUMINANCE_ALPHA`	$R_s * L_f$	$G_s * L_f$	$B_s * L_f$	$A_s * A_f$
`INTENSITY`	$R_s * I_f$	$G_s * I_f$	$B_s * I_f$	$A_s * I_f$
`RGB`	$R_s * R_f$	$G_s * G_f$	$B_s * B_f$	A_s
`RGBA`	$R_s * R_f$	$G_s * G_f$	$B_s * B_f$	$A_s * A_f$

Table 3.15: Computation of filtered color components depending on filter image format. $C * F$ indicates the convolution of image component C with filter F.

nomenclature is decribed in the **Convolution Filter Specification** subsection of section 3.7.3.

One-dimensional filter:

$$C[i'] = \sum_{n=0}^{W_f-1} C'_s[i'+n] * C_f[n]$$

Two-dimensional filter:

$$C[i',j'] = \sum_{n=0}^{W_f-1} \sum_{m=0}^{H_f-1} C'_s[i'+n, j'+m] * C_f[n,m]$$

Two-dimensional separable filter:

$$C[i',j'] = \sum_{n=0}^{W_f-1} \sum_{m=0}^{H_f-1} C'_s[i'+n, j'+m] * C_{row}[n] * C_{column}[m]$$

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

`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 $C'_s[i']$:

$$C'_s[i'] = \begin{cases} C_s[i'], & 0 \le i' < W_s \\ C_c, & \text{otherwise} \end{cases}$$

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 $C'_s[i', j']$:

$$C'_s[i',j'] = \begin{cases} C_s[i',j'], & 0 \le i' < W_s, 0 \le j' < H_s \\ C_c, & otherwise \end{cases}$$

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 $C'_s[i']$:

$$C'_s[i'] = C_s[\mathrm{clamp}(i', W_s)]$$

and the clamping function $\mathrm{clamp}(val, max)$ is defined as

$$\mathrm{clamp}(val, max) = \begin{cases} 0, & val < 0 \\ val, & 0 \le val < max \\ max - 1, & val \ge max \end{cases}$$

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 $(0, 0, 0, 0)$.

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 3.7.6.

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

$$\begin{pmatrix} R' \\ G' \\ B' \\ A' \end{pmatrix} = \begin{pmatrix} R_s & 0 & 0 & 0 \\ 0 & G_s & 0 & 0 \\ 0 & 0 & B_s & 0 \\ 0 & 0 & 0 & A_s \end{pmatrix} M_c \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix} + \begin{pmatrix} R_b \\ G_b \\ B_b \\ A_b \end{pmatrix}.$$

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 3.7.6.

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, and 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 or 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 luminance 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.

3.7.7 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 nth in a row and belongs to the mth 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 3.7.5. 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.

A single pixel rectangle will generate multiple, perhaps very many fragments for the same framebuffer pixel, depending on the pixel zoom factors.

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

`void` **Bitmap**`(` `sizei` *w*`,` `sizei` *h*`,` `float` x_{bo}`,` `float` y_{bo}`,`
`float` x_{bi}`,` `float` y_{bi}`,` `ubyte` **data* `);`

w and *h* comprise the integer width and height of the rectangular bitmap, respectively. (x_{bo}, y_{bo}) gives the floating-point x and y values of the bitmap's origin. (x_{bi}, y_{bi}) 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.

Like a polygon pattern, a bitmap is unpacked from memory according to the procedure given in section 3.7.5 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

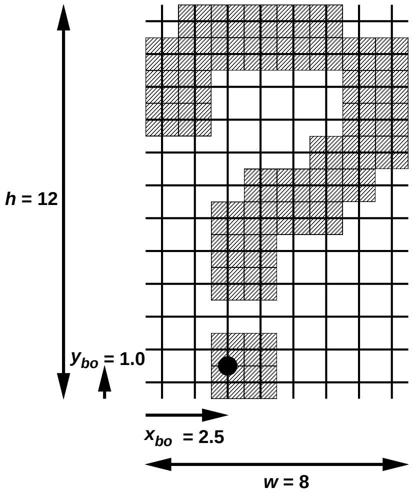


Figure 3.9. A bitmap and its associated parameters. x_{bi} and y_{bi} are not shown.

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).

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 in shaders by *sampling* the color of an image at the location indicated by specified (s, t, r) *texture coordinates*. It is accomplished in fixed-function processing by using the color of an image at the location indicated by a texture coordinate set's (s, t, r, q) values. Texture lookups are typically used to modify a fragment's RGBA color but may be used for any purpose in a shader.

The internal data type of a texture may be signed or unsigned normalized fixed-point, signed or unsigned integer, or floating-point, depending on the internal format of the texture. The correspondence between the internal format and the internal data type is given in tables 3.17-3.19. 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 either signed normalized fixed-point or integer textures in this case are undefined.

Each of the supported types of texture is a collection of images built from one-, two-, or three-dimensional arrays of image elements referred to as *texels*. One-, two-, and three-dimensional textures consist respectively of one-, two-, or three-dimensional texel arrays. One- and two-dimensional array textures are arrays of one- or two-dimensional images, consisting of one or more layers. Two-dimensional multisample and two-dimensional multisample array textures are special two-dimensional and two-dimensional array textures, respectively, containing multiple samples in each texel. Cube maps are special two-dimensional array textures with six layers that represent the faces of a cube. When accessing a cube map, the texture coordinates are projected onto one of the six faces of the cube. Rectangular textures are special two-dimensional textures consisting of only a single image and accessed using unnormalized coordinates. Buffer textures are special one-dimensional textures whose texel arrays are stored in separate buffer objects.

Implementations must support texturing using multiple images. 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 3.9.9) specify the GL operation with a single texture. Section 3.9.19 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.

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.

The command

`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 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**, **TexImage**, **BindTexture**, **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.

The active texture unit selector also specifies the texture coordinate set accessed by commands involving texture coordinate processing (see section 2.12.1).

ActiveTexture generates the error `INVALID_ENUM` if an invalid *texture* is specified. *texture* is a symbolic constant of the form `TEXTURE`i, indicating that texture unit i is to be modified. The constants obey `TEXTURE`i $=$ `TEXTURE0` $+ i$ (i is in the range 0 to $k-1$, where k is the larger of the values 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`.

The state required for the active texture image unit selector is a single integer. The initial value is `TEXTURE0`.

3.9.1 Texture Image Specification

The command

void **TexImage3D**(enum *target*, int *level*, int *internalformat*,
sizei *width*, sizei *height*, sizei *depth*, int *border*,
enum *format*, enum *type*, void **data*);

is used to specify a three-dimensional texture image. *target* must be one of `TEXTURE_3D` for a three-dimensional texture or `TEXTURE_2D_ARRAY` for an two-dimensional array texture. Additionally, *target* may be either `PROXY_TEXTURE_3D` for a three-dimensional proxy texture, or `PROXY_TEXTURE_2D_ARRAY` for a two-dimensional proxy array texture, as discussed in section 3.9.13. *format*, *type*, and *data* specify the format of the image data, the type of those data, and a reference to the image data in the currently bound pixel unpack buffer or client memory, as described in section 3.7.4. The *format* `STENCIL_INDEX` is not allowed.

The groups in memory are treated as being arranged in a sequence of adjacent rectangles. Each rectangle is a two-dimensional image, whose size and organization are specified by the *width* and *height* parameters to **TexImage3D**. The values of `UNPACK_ROW_LENGTH` and `UNPACK_ALIGNMENT` control the row-to-row spacing in these images as described in section 3.7.4. If the value of the integer parameter `UNPACK_IMAGE_HEIGHT` is not positive, then the number of rows in each two-dimensional image is *height*; otherwise the number of rows is `UNPACK_IMAGE_HEIGHT`. Each two-dimensional image comprises an integral number of rows, and is exactly adjacent to its neighbor images.

The mechanism for selecting a sub-volume of a three-dimensional image relies on the integer parameter `UNPACK_SKIP_IMAGES`. If `UNPACK_SKIP_IMAGES` is positive, the pointer is advanced by `UNPACK_SKIP_IMAGES` times the number of elements in one two-dimensional image before obtaining the first group from memory. Then *depth* two-dimensional images are processed, each having a subimage extracted as described in section 3.7.4.

The selected groups are transferred to the GL as described in section 3.7.4 and then clamped to the representable range of the internal format. If the *internalformat* of the texture is signed or unsigned integer, components are clamped to $[-2^{n-1}, 2^{n-1} - 1]$ or $[0, 2^n - 1]$, respectively, where n is the number of bits per component. For color component groups, if the *internalformat* of the texture is signed or unsigned normalized fixed-point, components are clamped to $[-1, 1]$ or $[0, 1]$, respectively. For depth component groups, the depth value is clamped to $[0, 1]$. Otherwise, values are not modified. Stencil index values are masked by $2^n - 1$, where n is the number of stencil bits in the internal format resolution (see below). If the base internal format is `DEPTH_STENCIL` and *format* is not `DEPTH_STENCIL`, then the values of the stencil index texture components are undefined.

Components are then selected from the resulting R, G, B, A, depth, or stencil values to obtain a texture with the *base internal format* specified by (or derived from) *internalformat*. Table 3.16 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 3.16, as one of the *sized internal format* symbolic constants listed in tables 3.17- 3.19, as one of the generic compressed internal format symbolic constants listed in table 3.20, or as one of the specific compressed internal format symbolic constants (if listed in table 3.20). *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`.

Base Internal Format	RGBA, Depth, and Stencil Values	Internal Components
`ALPHA`	A	A
`DEPTH_COMPONENT`	Depth	D
`DEPTH_STENCIL`	Depth,Stencil	D,S
`LUMINANCE`	R	L
`LUMINANCE_ALPHA`	R,A	L,A
`INTENSITY`	R	I
`RED`	R	R
`RG`	R,G	R,G
`RGB`	R,G,B	R,G,B
`RGBA`	R,G,B,A	R,G,B,A

Table 3.16: Conversion from RGBA, depth, and stencil pixel components to internal texture, table, or filter components. See section 3.9.15 for a description of the texture components R, G, B, A, L, I, D, and S.

Textures with a base internal format of `DEPTH_COMPONENT` or `DEPTH_STENCIL` are supported by texture image specification commands only if *target* is `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`, `TEXTURE_RECTANGLE`, `TEXTURE_CUBE_MAP`, `PROXY_TEXTURE_1D`, `PROXY_TEXTURE_2D`, `PROXY_TEXTURE_1D_ARRAY`, `PROXY_TEXTURE_2D_ARRAY`, `PROXY_TEXTURE_RECTANGLE`, or `PROXY_TEXTURE_CUBE_MAP`. Using these formats in conjunction with any other *target* will result in an `INVALID_OPERATION` error.

Textures with a base internal format of `DEPTH_COMPONENT` or `DEPTH_STENCIL` require either depth component data or depth/stencil component data. Textures with other base internal formats require RGBA component data. The error `INVALID_OPERATION` is generated if one of the base internal format and *format* is `DEPTH_COMPONENT` or `DEPTH_STENCIL`, and the other is neither of these values.

Textures with integer internal formats (see tables 3.17- 3.18) 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 *type* is `FLOAT`, `HALF_FLOAT`, `UNSIGNED_INT_10F_11F_11F_REV`, or `UNSIGNED_INT_5_9_9_9_REV`.

In addition to the specific compressed internal formats listed in table 3.20, the GL provides a mechanism to obtain token values for all such formats provided by extensions. The number of specific compressed internal formats supported by the renderer can be obtained by querying the value of `NUM_COMPRESSED_TEXTURE_FORMATS`. The set of specific compressed internal formats supported by the renderer can be obtained by querying the value of `COMPRESSED_TEXTURE_FORMATS`. The only values returned by this query are those corresponding to formats suitable for general-purpose usage. The renderer will not enumerate formats with restrictions that need to be specifically understood prior to use.

Generic compressed internal formats are never used directly as the internal formats of texture images. If *internalformat* is one of the six generic compressed internal formats, its value is replaced by the symbolic constant for a specific compressed internal format of the GL's choosing with the same base internal format. If no specific compressed format is available, *internalformat* is instead replaced by the corresponding base internal format. If *internalformat* is given as or mapped to a specific compressed internal format, but the GL can not support images compressed in the chosen internal format for any reason (e.g., the compression format might not support 3D textures or borders), *internalformat* is replaced by the corresponding base internal format and the texture image will not be compressed by the GL.

The *internal component resolution* is the number of bits allocated to each value in a texture image. If *internalformat* is specified as a base internal format, the GL stores the resulting texture with internal component resolutions of its own choosing. If a sized internal format is specified, the mapping of the R, G, B, A, depth, and stencil values to texture components is equivalent to the mapping of the corresponding base internal format's components, as specified in table 3.16; 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 3.17- 3.19 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 3.17- 3.19 is non-zero, and zero bits must be allocated for all other components).

Required Texture Formats

Implementations are required to support at least one allocation of internal component resolution for each type (unsigned int, float, etc.) for each base internal format.

In addition, implementations are required to support the following sized and compressed internal formats. Requesting one of these sized internal formats for any texture type will allocate at least the internal component sizes, and exactly the component types shown for that format in tables 3.17- 3.19:

- Texture and renderbuffer color formats (see section 4.4.2).
  - `RGBA32F`, `RGBA32I`, `RGBA32UI`, `RGBA16`, `RGBA16F`, `RGBA16I`, `RGBA16UI`, `RGBA8`, `RGBA8I`, `RGBA8UI`, `SRGB8_ALPHA8`, and `RGB10_A2`.
  - `R11F_G11F_B10F`.
  - `RG32F`, `RG32I`, `RG32UI`, `RG16`, `RG16F`, `RG16I`, `RG16UI`, `RG8`, `RG8I`, and `RG8UI`.
  - `R32F`, `R32I`, `R32UI`, `R16F`, `R16I`, `R16UI`, `R16`, `R8`, `R8I`, and `R8UI`.
  - `ALPHA8`.
- Texture-only color formats:
  - `RGBA16_SNORM` and `RGBA8_SNORM`.
  - `RGB32F`, `RGB32I`, and `RGB32UI`.
  - `RGB16_SNORM`, `RGB16F`, `RGB16I`, `RGB16UI`, and `RGB16`.
  - `RGB8_SNORM`, `RGB8`, `RGB8I`, `RGB8UI`, and `SRGB8`.
  - `RGB9_E5`.
  - `RG16_SNORM`, `RG8_SNORM`, `COMPRESSED_RG_RGTC2` and `COMPRESSED_SIGNED_RG_RGTC2`.
  - `R16_SNORM`, `R8_SNORM`, `COMPRESSED_RED_RGTC1` and `COMPRESSED_SIGNED_RED_RGTC1`.
- Depth formats: `DEPTH_COMPONENT32F`, `DEPTH_COMPONENT24`, and `DEPTH_COMPONENT16`.
- Combined depth+stencil formats: `DEPTH32F_STENCIL8` and `DEPTH24_STENCIL8`.

Encoding of Special Internal Formats

If *internalformat* is `R11F_G11F_B10F`, the red, green, and blue bits are converted to unsigned 11-bit, unsigned 11-bit, and unsigned 10-bit floating-point values as described in sections 2.1.3 and 2.1.4.

If *internalformat* is `RGB9_E5`, the red, green, and blue bits are converted to a shared exponent format according to the following procedure:

Components red, $green$, and $blue$ are first clamped (in the process, mapping NaN to zero) as follows:

$$
\begin{aligned}
red_c &= max(0, min(sharedexp_{max}, red)) \\
green_c &= max(0, min(sharedexp_{max}, green)) \\
blue_c &= max(0, min(sharedexp_{max}, blue))
\end{aligned}
$$

where

$$sharedexp_{max} = \frac{(2^N - 1)}{2^N} 2^{E_{max} - B}.$$

N is the number of mantissa bits per component (9), B is the exponent bias (15), and E_{max} is the maximum allowed biased exponent value (31).

The largest clamped component, max_c, is determined:

$$max_c = max(red_c, green_c, blue_c)$$

A preliminary shared exponent exp_p is computed:

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

A refined shared exponent exp_s is computed:

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

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

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

$$
\begin{aligned}
red_s &= \left\lfloor \frac{red_c}{2^{exp_s - B - N}} + 0.5 \right\rfloor \\
green_s &= \left\lfloor \frac{green_c}{2^{exp_s - B - N}} + 0.5 \right\rfloor \\
blue_s &= \left\lfloor \frac{blue_c}{2^{exp_s - B - N}} + 0.5 \right\rfloor
\end{aligned}
$$

The resulting red_s, $green_s$, $blue_s$, and exp_s are stored in the red, green, blue, and shared bits respectively of the texture image.

An implementation accepting pixel data of *type* `UNSIGNED_INT_5_9_9_9_REV` with *format* `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.

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	Shared bits
`ALPHA4`	`ALPHA`				4	
`ALPHA8`	`ALPHA`				8	
`ALPHA12`	`ALPHA`				12	
`ALPHA16`	`ALPHA`				16	
`R8`	`RED`	8				
`R8_SNORM`	`RED`	s8				
`R16`	`RED`	16				
`R16_SNORM`	`RED`	s16				
`RG8`	`RG`	8	8			
`RG8_SNORM`	`RG`	s8	s8			
`RG16`	`RG`	16	16			
`RG16_SNORM`	`RG`	s16	s16			
`R3_G3_B2`	`RGB`	3	3	2		
`RGB4`	`RGB`	4	4	4		
`RGB5`	`RGB`	5	5	5		
`RGB8`	`RGB`	8	8	8		
`RGB8_SNORM`	`RGB`	s8	s8	s8		
`RGB10`	`RGB`	10	10	10		
`RGB12`	`RGB`	12	12	12		
`RGB16`	`RGB`	16	16	16		
`RGB16_SNORM`	`RGB`	s16	s16	s16		
`RGBA2`	`RGBA`	2	2	2	2	
`RGBA4`	`RGBA`	4	4	4	4	
`RGB5_A1`	`RGBA`	5	5	5	1	
`RGBA8`	`RGBA`	8	8	8	8	
`RGBA8_SNORM`	`RGBA`	s8	s8	s8	s8	
`RGB10_A2`	`RGBA`	10	10	10	2	
`RGBA12`	`RGBA`	12	12	12	12	
`RGBA16`	`RGBA`	16	16	16	16	
`RGBA16_SNORM`	`RGBA`	s16	s16	s16	s16	
`SRGB8`	`RGB`	8	8	8		
`SRGB8_ALPHA8`	`RGBA`	8	8	8	8	
`R16F`	`RED`	f16				
`RG16F`	`RG`	f16	f16			
`RGB16F`	`RGB`	f16	f16	f16		

Sized internal color formats continued on next page

Sized internal color formats continued from previous page						
Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	Shared bits
`RGBA16F`	`RGBA`	f16	f16	f16	f16	
`R32F`	`RED`	f32				
`RG32F`	`RG`	f32	f32			
`RGB32F`	`RGB`	f32	f32	f32		
`RGBA32F`	`RGBA`	f32	f32	f32	f32	
`R11F_G11F_B10F`	`RGB`	f11	f11	f10		
`RGB9_E5`	`RGB`	9	9	9		5
`R8I`	`RED`	i8				
`R8UI`	`RED`	ui8				
`R16I`	`RED`	i16				
`R16UI`	`RED`	ui16				
`R32I`	`RED`	i32				
`R32UI`	`RED`	ui32				
`RG8I`	`RG`	i8	i8			
`RG8UI`	`RG`	ui8	ui8			
`RG16I`	`RG`	i16	i16			
`RG16UI`	`RG`	ui16	ui16			
`RG32I`	`RG`	i32	i32			
`RG32UI`	`RG`	ui32	ui32			
`RGB8I`	`RGB`	i8	i8	i8		
`RGB8UI`	`RGB`	ui8	ui8	ui8		
`RGB16I`	`RGB`	i16	i16	i16		
`RGB16UI`	`RGB`	ui16	ui16	ui16		
`RGB32I`	`RGB`	i32	i32	i32		
`RGB32UI`	`RGB`	ui32	ui32	ui32		
`RGBA8I`	`RGBA`	i8	i8	i8	i8	
`RGBA8UI`	`RGBA`	ui8	ui8	ui8	ui8	
`RGBA16I`	`RGBA`	i16	i16	i16	i16	
`RGBA16UI`	`RGBA`	ui16	ui16	ui16	ui16	
`RGBA32I`	`RGBA`	i32	i32	i32	i32	
`RGBA32UI`	`RGBA`	ui32	ui32	ui32	ui32	

Table 3.17: Correspondence of sized internal color formats to base internal formats, internal data type, and *desired* component resolutions for each sized internal format. The component resolution prefix indicates the internal data type: *f* is floating point, *i* is signed integer, *ui* is unsigned integer, *s* is signed normalized fixed-point, and no prefix is unsigned normalized fixed-point.

Sized Internal Format	Base Internal Format	*A* bits	*L* bits	*I* bits
`LUMINANCE4`	`LUMINANCE`		4	
`LUMINANCE8`	`LUMINANCE`		8	
`LUMINANCE12`	`LUMINANCE`		12	
`LUMINANCE16`	`LUMINANCE`		16	
`LUMINANCE4_ALPHA4`	`LUMINANCE_ALPHA`	4	4	
`LUMINANCE6_ALPHA2`	`LUMINANCE_ALPHA`	2	6	
`LUMINANCE8_ALPHA8`	`LUMINANCE_ALPHA`	8	8	
`LUMINANCE12_ALPHA4`	`LUMINANCE_ALPHA`	4	12	
`LUMINANCE12_ALPHA12`	`LUMINANCE_ALPHA`	12	12	
`LUMINANCE16_ALPHA16`	`LUMINANCE_ALPHA`	16	16	
`INTENSITY4`	`INTENSITY`			4
`INTENSITY8`	`INTENSITY`			8
`INTENSITY12`	`INTENSITY`			12
`INTENSITY16`	`INTENSITY`			16
`SLUMINANCE`	`LUMINANCE`		8	
`SLUMINANCE8_ALPHA8`	`LUMINANCE_ALPHA`	8	8	

Table 3.18: 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 3.16. 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

Sized Internal Format	Base Internal Format	D bits	S bits
DEPTH_COMPONENT16	DEPTH_COMPONENT	16	
DEPTH_COMPONENT24	DEPTH_COMPONENT	24	
DEPTH_COMPONENT32	DEPTH_COMPONENT	32	
DEPTH_COMPONENT32F	DEPTH_COMPONENT	f32	
DEPTH24_STENCIL8	DEPTH_STENCIL	24	8
DEPTH32F_STENCIL8	DEPTH_STENCIL	f32	8

Table 3.19: 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.

Compressed Internal Format	Base Internal Format	Type
COMPRESSED_ALPHA	ALPHA	Generic
COMPRESSED_LUMINANCE	LUMINANCE	Generic
COMPRESSED_LUMINANCE_ALPHA	LUMINANCE_ALPHA	Generic
COMPRESSED_INTENSITY	INTENSITY	Generic
COMPRESSED_RED	RED	Generic
COMPRESSED_RG	RG	Generic
COMPRESSED_RGB	RGB	Generic
COMPRESSED_RGBA	RGBA	Generic
COMPRESSED_SRGB	RGB	Generic
COMPRESSED_SRGB_ALPHA	RGBA	Generic
COMPRESSED_SLUMINANCE	LUMINANCE	Generic
COMPRESSED_SLUMINANCE_ALPHA	LUMINANCE_ALPHA	Generic
COMPRESSED_RED_RGTC1	RED	Specific
COMPRESSED_SIGNED_RED_RGTC1	RED	Specific
COMPRESSED_RG_RGTC2	RG	Specific
COMPRESSED_SIGNED_RG_RGTC2	RG	Specific

Table 3.20: Generic and specific compressed internal formats. 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 3.9.13.

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 3.16. Counting from zero, each resulting Nth texel is assigned internal integer coordinates (i, j, k), where

$$\begin{aligned} i &= (N \bmod width) - w_b \\ j &= (\lfloor \frac{N}{width} \rfloor \bmod height) - h_b \\ k &= (\lfloor \frac{N}{width \times height} \rfloor \bmod depth) - d_b \end{aligned}$$

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.

If the internal data type of the image array is signed or unsigned normalized fixed-point, each color component is converted using equation 2.6 or 2.4, respectively. If the internal type is floating-point or integer, components are clamped to the representable range of the corresponding internal component, but are not converted.

The *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

$$\begin{aligned} w_s &= w_t + 2w_b \\ h_s &= h_t + 2h_b \\ d_s &= d_t + 2d_b \end{aligned} \tag{3.17}$$

where w_s, h_s, and d_s are the specified image *width*, *height*, and *depth*, and w_t, h_t, and d_t are the dimensions of the texture image internal to the border. If w_t, h_t, or d_t are less than zero, then the error `INVALID_VALUE` is generated.

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 `TEXTURE_BASE_LEVEL` (see section 3.9.6), it is as if texturing were disabled.

The maximum border width b_t is 1. If *border* is less than zero, or greater than b_t, then the error `INVALID_VALUE` is generated.

The maximum allowable width, height, or depth of a texel array for a three-dimensional texture is an implementation-dependent function of the level-of-detail and internal format of the resulting image array. It must be at least $2^{k-lod} + 2b_t$ for image arrays of level-of-detail 0 through k, where k is the log base 2 of `MAX_3D_TEXTURE_SIZE`, lod is the level-of-detail of the image array, and b_t is the maximum border width. It may be zero for image arrays of any level-of-detail greater than k. The error `INVALID_VALUE` is generated if the specified image is too large to be stored under any conditions.

If a pixel unpack buffer object is bound and storing texture data would access memory beyond the end of the pixel unpack buffer, an `INVALID_OPERATION` error results.

In a similar fashion, the maximum allowable width of a texel array for a one- or two-dimensional, one- or two-dimensional array, two-dimensional multisample, or two-dimensional multisample array texture, and the maximum allowable height of a two-dimensional, two-dimensional array, two-dimensional multisample, or two-dimensional multisample array texture, must be at least $2^{k-lod} + 2b_t$ for image arrays of level 0 through k, where k is the log base 2 of `MAX_TEXTURE_SIZE`. The maximum allowable width and height of a cube map texture must be the same, and must be at least $2^{k-lod} + 2b_t$ for image arrays level 0 through k, where k is the log base 2 of `MAX_CUBE_MAP_TEXTURE_SIZE`. The maximum number of layers for one- and two-dimensional array textures (height or depth, respectively) must be at least `MAX_ARRAY_TEXTURE_LAYERS` for all levels.

The maximum allowable width and height of a rectangular texture image must each be at least the value of the implementation-dependent constant `MAX_RECTANGLE_TEXTURE_SIZE`.

An implementation may allow an image array of level 0 to be created only if that single image array can be supported. Additional constraints on the creation of image arrays of level 1 or greater are described in more detail in section 3.9.12.

The command

`void` **TexImage2D**(`enum` *target*, `int` *level*,
`int` *internalformat*, `sizei` *width*, `sizei` *height*,

`int` *border*, `enum` *format*, `enum` *type*, `void` **data*);

is used to specify a two-dimensional texture image. *target* must be one of `TEXTURE_2D` for a two-dimensional texture, `TEXTURE_1D_ARRAY` for a one-dimensional array texture, `TEXTURE_RECTANGLE` for a rectangle texture, or one of `TEXTURE_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, or `TEXTURE_CUBE_MAP_NEGATIVE_Z` for a cube map texture. Additionally, *target* may be either `PROXY_TEXTURE_2D` for a two-dimensional proxy texture, `PROXY_TEXTURE_1D_ARRAY` for a one-dimensional proxy array texture, `PROXY_TEXTURE_RECTANGLE` for a rectangle proxy texture, or `PROXY_TEXTURE_CUBE_MAP` for a cube map proxy texture in the special case discussed in section 3.9.13. The other parameters match the corresponding parameters of **TexImage3D**.

For the purposes of decoding the texture image, **TexImage2D** is equivalent to calling **TexImage3D** with corresponding arguments and *depth* of 1, except that

- 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 or rectangle texture consists of a single two-dimensional texture image. A cube map texture is a set of six two-dimensional texture images. The six cube map texture targets form a single cube map texture though each target names a distinct face of the cube map. The `TEXTURE_CUBE_MAP_*` targets listed above update their appropriate cube map face 2D texture image. Note that the six cube map two-dimensional image tokens such as `TEXTURE_CUBE_MAP_POSITIVE_X` are used when specifying, updating, or querying one of a cube map's six two-dimensional images, but when 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.

When *target* is `TEXTURE_RECTANGLE`, an `INVALID_VALUE` error is generated if *level* is non-zero.

When *target* is `TEXTURE_RECTANGLE`, an `INVALID_VALUE` error is generated if *border* is non-zero.

Finally, the command

```
void TexImage1D( enum target, int level,
    int internalformat, sizei width, int border,
    enum format, enum type, void *data );
```

is used to specify a one-dimensional texture image. *target* must be either `TEXTURE_1D`, or `PROXY_TEXTURE_1D` in the special case discussed in section 3.9.13.

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 3.17. A two-dimensional texel array has depth $d_s = 1$, with height h_s and width w_s as above. A rectangular texel array must have zero border width, so w_s and h_s equal the specified *width* and *height*, respectively, while $d_s = 1$. 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

---

[1] Figure 3.10 needs to show a three-dimensional texture image.

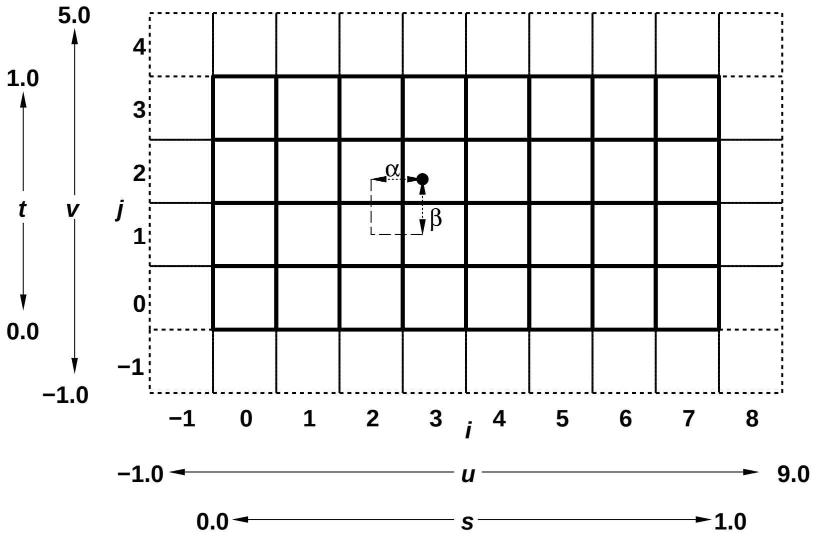


Figure 3.10. A texture image and the coordinates used to access it. This is a two-dimensional texture with $n = 3$ and $m = 2$. A one-dimensional texture would consist of a single horizontal strip. α and β, values used in blending adjacent texels to obtain a texture value, are also shown.

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 in fixed-function fragment shading, or by sampling the texture in a shader, 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.

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

`void` **CopyTexImage2D**(`enum` *target*, `int` *level*,
`enum` *internalformat*, `int` *x*, `int` *y*, `sizei` *width*,
`sizei` *height*, `int` *border*);

defines a two-dimensional texel array in exactly the manner of **TexImage2D**, except that the image data are taken from the framebuffer rather than from client memory. Currently, *target* must be one of `TEXTURE_2D`, `TEXTURE_1D_ARRAY`, `TEXTURE_RECTANGLE`, `TEXTURE_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, or `TEXTURE_CUBE_MAP_NEGATIVE_Z`. *x*, *y*, *width*, and *height* correspond precisely to the corresponding arguments to **ReadPixels** (refer to section 4.3.2); they specify the image's *width* and *height*, and the lower left (x, y) coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to **ReadPixels** with argument *type* set to `COLOR`, `DEPTH`, or `DEPTH_STENCIL`, depending on *internalformat*, stopping after 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.

The command

`void` **CopyTexImage1D**(`enum` *target*, `int` *level*,
`enum` *internalformat*, `int` *x*, `int` *y*, `sizei` *width*,
`int` *border*);

defines a one-dimensional texel array in exactly the manner of **TexImage1D**, except that the image data are taken from the framebuffer, rather than from client memory. Currently, *target* must be `TEXTURE_1D`. For the purposes of decoding the texture image, **CopyTexImage1D** is equivalent to calling **CopyTexImage2D** with corresponding arguments and *height* of 1, except that the *height* of the image is always 1, regardless of the value of *border*. *level*, *internalformat*, and *border* are specified using the same values, with the same meanings, as the equivalent arguments of **TexImage1D**, 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,

`void` **TexSubImage3D**(`enum` *target*, `int` *level*, `int` *xoffset*,
`int` *yoffset*, `int` *zoffset*, `sizei` *width*, `sizei` *height*,
`sizei` *depth*, `enum` *format*, `enum` *type*, `void` **data*);
`void` **TexSubImage2D**(`enum` *target*, `int` *level*, `int` *xoffset*,
`int` *yoffset*, `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** and **CopyTexSubImage2D** must be one of `TEXTURE_2D`, `TEXTURE_1D_ARRAY`, `TEXTURE_RECTANGLE`, `TEXTURE_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, or `TEXTURE_CUBE_MAP_NEGATIVE_Z`, and the *target* arguments of **TexSubImage3D** and **CopyTexSubImage3D** must be `TEXTURE_3D` or `TEXTURE_2D_ARRAY`. The *level* parameter of each command specifies the level of the texel array that is modified. If *level* is less than zero or greater than the base 2 logarithm of the maximum texture width, height, or depth, the error `INVALID_VALUE` is generated. If *target* is `TEXTURE_RECTANGLE` and *level* is not zero, the error `INVALID_VALUE` is generated. **TexSubImage3D** arguments *width*, *height*, *depth*, *format*, *type*, and *data* match the corresponding arguments to **TexImage3D**, meaning that they are specified using the same values, and have the same meanings. Likewise, **TexSubImage2D** arguments *width*, *height*, *format*, *type*, and *data* match the corresponding arguments to **TexImage2D**, and **TexSubImage1D** arguments *width*, *format*, *type*, and *data* match the corresponding arguments to **TexImage1D**.

CopyTexSubImage3D and **CopyTexSubImage2D** arguments *x*, *y*, *width*, and *height* match the corresponding arguments to **CopyTexImage2D**[2]. **CopyTexSubImage1D** arguments *x*, *y*, and *width* match the corresponding arguments to **CopyTexImage1D**. Each of the **TexSubImage** commands interprets and processes pixel groups in exactly the manner of its **TexImage** counterpart, except that the assignment of R, G, B, A, depth, and stencil index pixel group values to the texture components is controlled by the *internalformat* of the texel array, not by an argument to the command. The same constraints and errors apply to the **TexSubImage** commands' argument *format* and the *internalformat* of the texel array being respecified as apply to the *format* and *internalformat* arguments of its **TexImage** counterparts.

Arguments *xoffset*, *yoffset*, and *zoffset* of **TexSubImage3D** and **CopyTexSubImage3D** specify the lower left texel coordinates of a *width*-wide by *height*-high by *depth*-deep rectangular subregion of the texel array. The *depth* argument associated with **CopyTexSubImage3D** is always 1, because framebuffer memory is two-dimensional - only a portion of a single s, t slice of a three-dimensional texture is replaced by **CopyTexSubImage3D**.

Negative values of *xoffset*, *yoffset*, and *zoffset* correspond to the coordinates of border texels, addressed as in figure 3.10. Taking w_s, h_s, d_s, w_b, h_b, and d_b to be the specified width, height, depth, and border width, border height, and border depth of the texel array, and taking x, y, z, w, h, and d to be the *xoffset*, *yoffset*,

---

[2] Because the framebuffer is inherently two-dimensional, there is no **CopyTexImage3D** command.

zoffset, *width*, *height*, and *depth* argument values, any of the following relationships generates the error `INVALID_VALUE`:

$$x < -w_b$$

$$x + w > w_s - w_b$$

$$y < -h_b$$

$$y + h > h_s - h_b$$

$$z < -d_b$$

$$z + d > d_s - d_b$$

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

$$i = x + (n \bmod w)$$

$$j = y + (\lfloor \frac{n}{w} \rfloor \bmod h)$$

$$k = z + (\lfloor \frac{n}{width * height} \rfloor \bmod d$$

Arguments *xoffset* and *yoffset* of **TexSubImage2D** and **CopyTexSubImage2D** specify the lower left texel coordinates of a *width*-wide by *height*-high rectangular subregion of the texel array. Negative values of *xoffset* and *yoffset* correspond to the coordinates of border texels, addressed as in figure 3.10. Taking w_s, h_s, and b_s to be the specified width, height, and border width of the texel array, and taking x, y, w, and h to be the *xoffset*, *yoffset*, *width*, and *height* argument values, any of the following relationships generates the error `INVALID_VALUE`:

$$x < -b_s$$

$$x + w > w_s - b_s$$

$$y < -b_s$$

$$y + h > h_s - b_s$$

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

$$i = x + (n \bmod w)$$

$$j = y + (\lfloor \frac{n}{w} \rfloor \bmod h)$$

The *xoffset* argument of **TexSubImage1D** and **CopyTexSubImage1D** specifies the left texel coordinate of a *width*-wide subregion of the texel array. Negative values of *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 *xoffset* and *width* argument values, either of the following relationships generates the error `INVALID_VALUE`:

$$x < -b_s$$

$$x + w > w_s - b_s$$

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

$$i = x + (n \bmod w)$$

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 **TexSubImage3D**, **CopyTexSubImage3D**, **TexSubImage2D**, **CopyTexSubImage2D**, **TexSubImage1D**, or **CopyTexSubImage1D** will result in an `INVALID_OPERATION` error if *xoffset*, *yoffset*, or *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 `RGTC` formats described in table 3.20, the texture is stored using one of the RGTC texture image encodings (see appendix C.1). Since RGTC images are easily edited along 4×4 texel boundaries, the limitations on subimage location and size are relaxed for **TexSubImage2D**, **TexSubImage3D**, **CopyTexSubImage2D**, and **CopyTexSubImage3D**. These commands will generate an `INVALID_OPERATION` error if one of the following conditions occurs:

- *width* is not a multiple of four, $width + xoffset$ is not equal to the value of `TEXTURE_WIDTH`, and either *xoffset* or *yoffset* is non-zero.

- *height* is not a multiple of four, $height + yoffset$ is not equal to the value of `TEXTURE_HEIGHT`, and either *xoffset* or *yoffset* is non-zero.

- *xoffset* or *yoffset* is not a multiple of four.

The contents of any 4×4 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).

Texture Copying Feedback Loops

Calling **CopyTexSubImage3D**, **CopyTexImage2D**, **CopyTexSubImage2D**, **CopyTexImage1D**, or **CopyTexSubImage1D** will result in undefined behavior if the destination texture image level is also bound to to the selected read buffer (see section 4.3.2) of the read framebuffer. This situation is discussed in more detail in the description of feedback loops in section 4.4.3.

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

`void` **CompressedTexImage1D**(`enum` *target*, `int` *level*,
`enum` *internalformat*, `sizei` *width*, `int` *border*,
`sizei` *imageSize*, `void` **data*);
`void` **CompressedTexImage2D**(`enum` *target*, `int` *level*,
`enum` *internalformat*, `sizei` *width*, `sizei` *height*,
`int` *border*, `sizei` *imageSize*, `void` **data*);
`void` **CompressedTexImage3D**(`enum` *target*, `int` *level*,
`enum` *internalformat*, `sizei` *width*, `sizei` *height*,
`sizei` *depth*, `int` *border*, `sizei` *imageSize*, `void` **data*);

define one-, two-, and three-dimensional texture images, respectively, with incoming data stored in a specific compressed image format. The *target*, *level*, *internalformat*, *width*, *height*, *depth*, and *border* parameters have the same meaning as in **TexImage1D**, **TexImage2D**, and **TexImage3D**, except that compressed rectangular texture formats are not supported. *data* refers to compressed image data stored in the specific compressed image format corresponding to *internalformat*.

If a pixel unpack buffer is bound (as indicated by a non-zero value of `PIXEL_UNPACK_BUFFER_BINDING`), *data* is an offset into the pixel unpack buffer and the compressed data is read from the buffer relative to this offset; otherwise, *data* is a pointer to client memory and the compressed data is read from client memory relative to the pointer.

If the *target* parameter to any of the **CompressedTexImage$n$D** commands is `TEXTURE_RECTANGLE` or `PROXY_TEXTURE_RECTANGLE`, the error `INVALID_ENUM` is generated.

internalformat must be a supported specific compressed internal format. An `INVALID_ENUM` error will be generated if any other values, including any of the generic compressed internal formats, is specified.

For all other compressed internal formats, the compressed image will be decoded according to the specification defining the *internalformat* token. Compressed texture images are treated as an array of *imageSize* `ubytes` relative to *data*. If a pixel unpack buffer object is bound and $data + imageSize$ is greater than the size of the pixel buffer, an `INVALID_OPERATION` error results. All pixel storage 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` formats described in table 3.20, the compressed image data is stored using one of the RGTC compressed texture image encodings (see appendix C.1) The RGTC texture compression algorithm supports only two-dimensional images without borders. If *internalformat* is an RGTC format, **CompressedTexImage1D** will generate an `INVALID_ENUM` error; **CompressedTexImage2D** will generate an `INVALID_OPERATION` error if *border* is non-zero or *target* is `TEXTURE_RECTANGLE`; and **CompressedTexImage3D** will generate an `INVALID_OPERATION` error if *border* is non-zero or *target* is not `TEXTURE_2D_ARRAY`.

If the *data* argument of **CompressedTexImage1D**, **CompressedTexImage2D**, or **CompressedTexImage3D** is a null pointer (a zero-valued pointer in the C implementation), and the pixel unpack buffer object is zero, a texel array with unspecified image contents is created, just as when a null pointer is passed to **TexImage1D**, **TexImage2D**, or **TexImage3D**.

The commands

```
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*. Using any of the generic compressed internal formats as *format* will result in an `INVALID_ENUM` error.

If the *target* parameter to any of the **CompressedTexSubImage$n$D** commands is `TEXTURE_RECTANGLE` or `PROXY_TEXTURE_RECTANGLE`, the error `INVALID_ENUM` is generated.

The image pointed to by *data* and the *imageSize* parameter are interpreted as though they were provided to **CompressedTexImage1D**, **CompressedTexImage2D**, and **CompressedTexImage3D**. These commands do not provide for 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:

- *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_SIZE` 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 $-b_s$ (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` formats described in table 3.20, 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 4×4 texel boundaries, the limitations on subimage location and size are relaxed for **CompressedTexSubImage2D** and **CompressedTexSubImage3D**. These commands will result in an `INVALID_OPERATION` error if one of the following conditions occurs:

- *width* is not a multiple of four, and $width + xoffset$ is not equal to the value of `TEXTURE_WIDTH`.
- *height* is not a multiple of four, and $height + yoffset$ is not equal to the value of `TEXTURE_HEIGHT`.
- *xoffset* or *yoffset* is not a multiple of four.

The contents of any 4×4 block of texels of an RGTC compressed texture image that does not intersect the area being modified are preserved during valid **TexSubImage*** and **CopyTexSubImage*** calls.

3.9.4 Multisample Textures

In addition to the texture types described in previous sections, two additional types of textures are supported. A multisample texture is similar to a two-dimensional or two-dimensional array texture, except it contains multiple samples per texel. Multisample textures do not have multiple image levels.

The commands

`void` **TexImage2DMultisample**(`enum` *target*, `sizei` *samples*,
  `int` *internalformat*, `sizei` *width*, `sizei` *height*,
  `boolean` *fixedsamplelocations*);

`void` **TexImage3DMultisample**(`enum` *target*`,` `sizei` *samples*`,`
`int` *internalformat*`,` `sizei` *width*`,` `sizei` *height*`,`
`sizei` *depth*`,` `boolean` *fixedsamplelocations* `);`

establish the data storage, format, dimensions, and number of samples of a multisample texture's image. For **TexImage2DMultisample**, *target* must be `TEXTURE_2D_MULTISAMPLE` or `PROXY_TEXTURE_2D_MULTISAMPLE` and for **TexImage3DMultisample** *target* must be `TEXTURE_2D_MULTISAMPLE_ARRAY` or `PROXY_TEXTURE_2D_MULTISAMPLE_ARRAY`. *width* and *height* are the dimensions in texels of the texture.

internalformat must be color-renderable, depth-renderable, or stencil- renderable (as defined in section 4.4.4). The error `INVALID_OPERATION` may be generated if any of the following are true:

- *internalformat* is a depth/stencil-renderable format and *samples* is greater than the value of `MAX_DEPTH_TEXTURE_SAMPLES`
- *internalformat* is a color-renderable format and *samples* is greater than the value of `MAX_COLOR_TEXTURE_SAMPLES`
- *internalformat* is a signed or unsigned integer format and *samples* is greater than the value of `MAX_INTEGER_SAMPLES`.

If *fixedsamplelocations* is `TRUE`, the image will use identical sample locations and the same number of samples for all texels in the image, and the sample locations will not depend on the internalformat or size of the image. If either *width* or *height* is greater than `MAX_TEXTURE_SIZE`, or if *samples* is greater than `MAX_SAMPLES`, then the error `INVALID_VALUE` is generated. If the GL is unable to create a texture level of the requested size, the error `OUT_OF_MEMORY` is generated.

When a multisample texture is accessed in a shader, the access takes one vector of integers describing which texel to fetch and an integer corresponding to the sample numbers described in section 3.3.1 describing which sample within the texel to fetch. No standard sampling instructions are allowed on the multisample texture targets.

3.9.5 Buffer Textures

In addition to one-, two-, and three-dimensional, one- and two-dimensional array, and cube map textures described in previous sections, one additional type of texture is supported. A buffer texture is similar to a one-dimensional texture. However, unlike other texture types, the texel array is not stored as part of the texture. Instead, a buffer object is attached to a buffer texture and the texel array is taken from that buffer object's data store. When the contents of a buffer object's data store are modified, those changes are reflected in the contents of any buffer texture to which the buffer object is attached. Buffer textures do not have multiple image levels; only a single data store is available.

The command

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

attaches the storage for the buffer object named *buffer* to the active buffer texture, and specifies an internal format for the texel array found in the attached buffer object. If *buffer* is zero, any buffer object attached to the buffer texture is detached, and no new buffer object is attached. If *buffer* is non-zero, but is not the name of an existing buffer object, the error `INVALID_OPERATION` is generated. *target* must be `TEXTURE_BUFFER`. *internalformat* specifies the storage format, and must be one of the sized internal formats found in table 3.21.

When a buffer object is attached to a buffer texture, the buffer object's data store is taken as the texture's texel array. The number of texels in the buffer texture's texel array is given by

$$\left\lfloor \frac{buffer\_size}{components \times sizeof(base\_type)} \right\rfloor .$$

where *buffer_size* is the size of the buffer object, in basic machine units and *components* and *base_type* are the element count and base data type for elements, as specified in table 3.21. The number of texels in the texel array is then clamped to the implementation-dependent limit `MAX_TEXTURE_BUFFER_SIZE`. When a buffer texture is accessed in a shader, the results of a texel fetch are undefined if the specified texel coordinate is negative, or greater than or equal to the clamped number of texels in the texel array.

When a buffer texture is accessed in a shader, an integer is provided to indicate the texel coordinate being accessed. If no buffer object is bound to the buffer texture, the results of the texel access are undefined. Otherwise, the attached buffer object's data store is interpreted as an array of elements of the GL data type corresponding to *internalformat*. Each texel consists of one to four elements that are mapped to texture components (R, G, B, and A). Element m of the texel numbered n is taken from element $n \times components + m$ of the attached buffer object's data store. Elements and texels are both numbered starting with zero. For texture formats with signed or unsigned normalized fixed-point components, the extracted values are converted to floating-point using equations 2.1 or 2.3, respectively. The components of the texture are then converted to an (R,G,B,A) vector according to table 3.21, and returned to the shader as a four-component result vector with components of the appropriate data type for the texture's internal format. The base data type, component count, normalized component information, and mapping of data store elements to texture components is specified in table 3.21.

In addition to attaching buffer objects to textures, buffer objects can be bound to the buffer object target named `TEXTURE_BUFFER`, in order to specify, modify, or read the buffer object's data store. The buffer object bound to `TEXTURE_BUFFER` has no effect on rendering. A buffer object is bound to `TEXTURE_BUFFER` by calling **BindBuffer** with *target* set to `TEXTURE_BUFFER`, as described in section 2.9.

3.9.6 Texture Parameters

Various parameters control how the texel array is treated when specified or changed, and when applied to a fragment. Each parameter is set by calling

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

target is the target, either `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`. `TEXTURE_RECTANGLE`, or `TEXTURE_CUBE_MAP`. *params* is a symbolic constant indicating the parameter to be set; the possible constants and corresponding parameters are summarized in table 3.22. 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 with **TexParameteri** or **TexParameteriv**, it is converted to floating-point using equation 2.2, 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**, they are converted to floating-point using equation 2.2. Otherwise the values are unmodified and stored as floating-point.

In the remainder of section 3.9, denote by lod_{min}, lod_{max}, $level_{base}$, and $level_{max}$ the values of the texture parameters `TEXTURE_MIN_LOD`, `TEXTURE_MAX_LOD`, `TEXTURE_BASE_LEVEL`, and `TEXTURE_MAX_LEVEL` respectively.

Sized Internal Format	Base Type	Components	Norm	Component			
				0	1	2	3
`R8`	`ubyte`	1	Yes	R	0	0	1
`R16`	`ushort`	1	Yes	R	0	0	1
`R16F`	`half`	1	No	R	0	0	1
`R32F`	`float`	1	No	R	0	0	1
`R8I`	`byte`	1	No	R	0	0	1
`R16I`	`short`	1	No	R	0	0	1
`R32I`	`int`	1	No	R	0	0	1
`R8UI`	`ubyte`	1	No	R	0	0	1
`R16UI`	`ushort`	1	No	R	0	0	1
`R32UI`	`uint`	1	No	R	0	0	1
`RG8`	`ubyte`	2	Yes	R	G	0	1
`RG16`	`ushort`	2	Yes	R	G	0	1
`RG16F`	`half`	2	No	R	G	0	1
`RG32F`	`float`	2	No	R	G	0	1
`RG8I`	`byte`	2	No	R	G	0	1
`RG16I`	`short`	2	No	R	G	0	1
`RG32I`	`int`	2	No	R	G	0	1
`RG8UI`	`ubyte`	2	No	R	G	0	1
`RG16UI`	`ushort`	2	No	R	G	0	1
`RG32UI`	`uint`	2	No	R	G	0	1
`RGBA8`	`ubyte`	4	Yes	R	G	B	A
`RGBA16`	`ushort`	4	Yes	R	G	B	A
`RGBA16F`	`half`	4	No	R	G	B	A
`RGBA32F`	`float`	4	No	R	G	B	A
`RGBA8I`	`byte`	4	No	R	G	B	A
`RGBA16I`	`short`	4	No	R	G	B	A
`RGBA32I`	`int`	4	No	R	G	B	A
`RGBA8UI`	`ubyte`	4	No	R	G	B	A
`RGBA16UI`	`ushort`	4	No	R	G	B	A
`RGBA32UI`	`uint`	4	No	R	G	B	A

Table 3.21: Internal formats for buffer textures. For each format, the data type of each element is indicated in the "Base Type" column and the element count is in the "Components" column. The "Norm" column indicates whether components should be treated as normalized floating-point values. The "Component 0, 1, 2, and 3" columns indicate the mapping of each element of a texel to texture components.

Name	Type	Legal Values
`TEXTURE_WRAP_S`	enum	`CLAMP`, `CLAMP_TO_EDGE`, `REPEAT`, `CLAMP_TO_BORDER`, `MIRRORED_REPEAT`
`TEXTURE_WRAP_T`	enum	`CLAMP`, `CLAMP_TO_EDGE`, `REPEAT`, `CLAMP_TO_BORDER`, `MIRRORED_REPEAT`
`TEXTURE_WRAP_R`	enum	`CLAMP`, `CLAMP_TO_EDGE`, `REPEAT`, `CLAMP_TO_BORDER`, `MIRRORED_REPEAT`
`TEXTURE_MIN_FILTER`	enum	`NEAREST`, `LINEAR`, `NEAREST_MIPMAP_NEAREST`, `NEAREST_MIPMAP_LINEAR`, `LINEAR_MIPMAP_NEAREST`, `LINEAR_MIPMAP_LINEAR`,
`TEXTURE_MAG_FILTER`	enum	`NEAREST`, `LINEAR`
`TEXTURE_BORDER_COLOR`	4 floats, integers, or unsigned integers	any 4 values
`TEXTURE_PRIORITY`	float	any value in $[0, 1]$
`TEXTURE_MIN_LOD`	float	any value
`TEXTURE_MAX_LOD`	float	any value
`TEXTURE_BASE_LEVEL`	integer	any non-negative integer
`TEXTURE_MAX_LEVEL`	integer	any non-negative integer
`TEXTURE_LOD_BIAS`	float	any value
`DEPTH_TEXTURE_MODE`	enum	`RED`, `LUMINANCE`, `INTENSITY`, `ALPHA`
`TEXTURE_COMPARE_MODE`	enum	`NONE`, `COMPARE_REF_TO_TEXTURE`

`TEXTURE_COMPARE_FUNC`	enum	`LEQUAL`, `GEQUAL`
		`LESS`, `GREATER`,
		`EQUAL`, `NOTEQUAL`,
		`ALWAYS`, `NEVER`

`GENERATE_MIPMAP`	boolean	`TRUE` or `FALSE`

Table 3.22: Texture parameters and their values.

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.

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 3.9.9.

When *target* is `TEXTURE_RECTANGLE`, certain texture parameter values may not be specified. In this case, the error `INVALID_ENUM` is generated if the `TEXTURE_WRAP_S`, `TEXTURE_WRAP_T`, or `TEXTURE_WRAP_R` parameter is set to `REPEAT` or `MIRRORED_REPEAT`. The error `INVALID_ENUM` is generated if `TEXTURE_MIN_FILTER` is set to a value other than `NEAREST` or `LINEAR` (no mipmap filtering is permitted). The error `INVALID_ENUM` is generated if `TEXTURE_BASE_LEVEL` is set to any value other than zero.

3.9.7 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 3.9.16). 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.

3.9.8 Cube Map Texture Selection

When cube map texturing is enabled, the $\begin{pmatrix} s & t & r \end{pmatrix}$ texture coordinates are treated as a direction vector $\begin{pmatrix} r_x & r_y & r_z \end{pmatrix}$ 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 $\begin{pmatrix} r_x & r_y & r_z \end{pmatrix}$. The target column in table 3.23 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 3.23, an updated $\begin{pmatrix} s & t \end{pmatrix}$ is calculated as follows:

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

Major Axis Direction	Target	s_c	t_c	m_a
$+r_x$	`TEXTURE_CUBE_MAP_POSITIVE_X`	$-r_z$	$-r_y$	r_x
$-r_x$	`TEXTURE_CUBE_MAP_NEGATIVE_X`	r_z	$-r_y$	r_x
$+r_y$	`TEXTURE_CUBE_MAP_POSITIVE_Y`	r_x	r_z	r_y
$-r_y$	`TEXTURE_CUBE_MAP_NEGATIVE_Y`	r_x	$-r_z$	r_y
$+r_z$	`TEXTURE_CUBE_MAP_POSITIVE_Z`	r_x	$-r_y$	r_z
$-r_z$	`TEXTURE_CUBE_MAP_NEGATIVE_Z`	$-r_x$	$-r_y$	r_z

Table 3.23: Selection of cube map images based on major axis direction of texture coordinates.

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

Seamless Cube Map Filtering

Seamless cube map filtering is enabled or disabled by calling **Enable** or **Disable**, respectively, with the symbolic constant `TEXTURE_CUBE_MAP_SEAMLESS`.

When seamless cube map filtering is disabled, the new $\begin{pmatrix} s & t \end{pmatrix}$ is used to find a texture value in the determined face's two-dimensional image using the rules given in sections 3.9.9 through 3.9.10.

When seamless cube map filtering is enabled, the rules for texel selection in sections 3.9.9 through 3.9.10 are modified so that texture wrap modes are ignored. Instead,

- If `NEAREST` filtering is done within a miplevel, always apply wrap mode `CLAMP_TO_EDGE`.
- If `LINEAR` filtering is done within a miplevel, always apply wrap mode `CLAMP_TO_BORDER`. Then,
  - If a texture sample location would lie in the texture border in either u or v, instead select the corresponding texel from the appropriate neighboring face.
  - If a texture sample location would lie in the texture border in *both* u and v (in one of the corners of the cube), there is no unique neighboring face from which to extract one texel. The recommended method to generate this texel is to average the values of the three available samples. However, implementations are free to construct this fourth texel

> in another way, so long as, when the three available samples have the same value, this texel also has that value.

The required state is one bit indicating whether seamless cube map filtering is enabled or disabled. Initially, it is disabled.

3.9.9 Texture Minification

Applying a texture to a primitive implies a mapping from texture image space to framebuffer image space. In general, this mapping involves a reconstruction of the sampled texture image, followed by a homogeneous warping implied by the mapping to framebuffer space, then a filtering, followed finally by a resampling of the filtered, warped, reconstructed image before applying it to a fragment. In the GL this mapping is approximated by one of two simple filtering schemes. One of these schemes is selected based on whether the mapping from texture space to framebuffer space is deemed to *magnify* or *minify* the texture image.

Scale Factor and Level of Detail

The choice is governed by a scale factor $\rho(x, y)$ and the *level-of-detail* parameter $\lambda(x, y)$, defined as

$$\lambda_{base}(x, y) = \log_2[\rho(x, y)] \tag{3.18}$$

$$\lambda'(x, y) = \lambda_{base}(x, y) + clamp(bias_{texobj} + bias_{texunit} + bias_{shader}) \tag{3.19}$$

$$\lambda = \begin{cases} lod_{max}, & \lambda' > lod_{max} \\ \lambda', & lod_{min} \le \lambda' \le lod_{max} \\ lod_{min}, & \lambda' < lod_{min} \\ undefined, & lod_{min} > lod_{max} \end{cases} \tag{3.20}$$

$bias_{texobj}$ is the value of `TEXTURE_LOD_BIAS` for the bound texture object (as described in section 3.9.6). $bias_{texunit}$ is the value of `TEXTURE_LOD_BIAS` for the current texture unit (as described in section 3.9.15). $bias_{shader}$ is the value of the optional bias parameter in the texture lookup functions available to fragment shaders. If the texture access is performed in a fragment shader without a provided bias, or outside a fragment shader, then $bias_{shader}$ is zero. The sum of these values is clamped to the range $[-bias_{max}, bias_{max}]$ where $bias_{max}$ is the value of the implementation defined constant `MAX_TEXTURE_LOD_BIAS`.

If $\lambda(x, y)$ is less than or equal to the constant c (see section 3.9.10) the texture is said to be magnified; if it is greater, the texture is minified. Sampling of minified textures is described in the remainder of this section, while sampling of magnified textures is described in section 3.9.10.

The initial values of lod_{min} and lod_{max} are chosen so as to never clamp the normal range of λ. They may be respecified for a specific texture by calling **TexParameter[if]** with *pname* set to `TEXTURE_MIN_LOD` or `TEXTURE_MAX_LOD` respectively.

Let $s(x, y)$ be the function that associates an s texture coordinate with each set of window coordinates (x, y) that lie within a primitive; define $t(x, y)$ and $r(x, y)$ analogously. Let

$$\begin{aligned} u(x,y) &= \begin{cases} s(x,y) + \delta_u, & \text{rectangular texture} \\ w_t \times s(x,y) + \delta_u, & \text{otherwise} \end{cases} \\ v(x,y) &= \begin{cases} t(x,y) + \delta_v, & \text{rectangular texture} \\ h_t \times t(x,y) + \delta_v, & \text{otherwise} \end{cases} \\ w(x,y) &= d_t \times r(x,y) + \delta_w \end{aligned} \tag{3.21}$$

where w_t, h_t, and d_t are as defined by equation 3.17 with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is $level_{base}$. For a one-dimensional or one-dimensional array texture, define $v(x, y) = 0$ and $w(x, y) = 0$; for a two-dimensional, two-dimensional array, rectangular, or cube map texture, define $w(x, y) = 0$.

$(\delta_u, \delta_v, \delta_w)$ are the texel offsets specified in the OpenGL Shading Language texture lookup functions that support offsets. If the texture function used does not support offsets, 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 `MIN_PROGRAM_TEXEL_OFFSET` and `MAX_PROGRAM_TEXEL_OFFSET`, results of the texture lookup are undefined.

For a polygon, or for a point sprite with texture coordinate replacement enabled, ρ is given at a fragment with window coordinates (x, y) by

$$\rho = \max\left\{ \sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial w}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial y}\right)^2} \right\} \tag{3.22}$$

where $\partial u / \partial x$ indicates the derivative of u with respect to window x, and similarly for the other derivatives.

For a line, the formula is

$$\rho = \sqrt{\left(\frac{\partial u}{\partial x}\Delta x + \frac{\partial u}{\partial y}\Delta y\right)^2 + \left(\frac{\partial v}{\partial x}\Delta x + \frac{\partial v}{\partial y}\Delta y\right)^2 + \left(\frac{\partial w}{\partial x}\Delta x + \frac{\partial w}{\partial y}\Delta y\right)^2} \Big/ l, \tag{3.23}$$

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, point sprite without texture coordinate replacement enabled, pixel rectangle, or bitmap, $\rho = 1$.

While it is generally agreed that equations 3.22 and 3.23 give the best results when texturing, they are often impractical to implement. Therefore, an implementation may approximate the ideal ρ 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 $\max\{m_u, m_v, m_w\} \le f(x, y) \le m_u + m_v + m_w$.

Coordinate Wrapping and Texel Selection

After generating $u(x, y)$, $v(x, y)$, and $w(x, y)$, they may be clamped and wrapped before sampling the texture, depending on the corresponding texture wrap modes.

Let

$$u'(x,y) = \begin{cases} clamp(u(x,y), 0, w_t), & \texttt{TEXTURE\_WRAP\_S} \text{ is } \texttt{CLAMP} \\ u(x,y), & \text{otherwise} \end{cases}$$

$$v'(x,y) = \begin{cases} clamp(v(x,y), 0, h_t), & \texttt{TEXTURE\_WRAP\_T} \text{ is } \texttt{CLAMP} \\ v(x,y), & \text{otherwise} \end{cases}$$

$$w'(x,y) = \begin{cases} clamp(w(x,y), 0, h_t), & \texttt{TEXTURE\_WRAP\_R} \text{ is } \texttt{CLAMP} \\ w(x,y), & \text{otherwise} \end{cases}$$

where $clamp(a, b, c)$ 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 $level_{base}$ that is nearest (in Manhattan distance) to (u', v', w') is obtained. Let (i, j, k) be integers such that

$$\begin{aligned} i &= wrap(\lfloor u'(x,y) \rfloor) \\ j &= wrap(\lfloor v'(x,y) \rfloor) \\ k &= wrap(\lfloor w'(x,y) \rfloor) \end{aligned}$$

and the value returned by $wrap()$ is defined in table 3.24. For a three-dimensional texture, the texel at location (i, j, k) becomes the texture value. For two-dimensional, two-dimensional array, rectangular, or cube map textures, k is irrelevant, and the texel at location (i, j) becomes the texture value. For one-dimensional texture or one-dimensional array textures, j and k are irrelevant, and the texel at location i becomes the texture value.

For one- and two-dimensional array textures, the texel is obtained from image layer l, where

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

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

$$\begin{aligned} i &< -b_s & i &\geq w_t + b_s \\ j &< -b_s & j &\geq h_t + b_s \\ k &< -b_s & k &\geq d_t + b_s \end{aligned}$$

Wrap mode	Result of $wrap(coord)$
`CLAMP`	$\begin{cases} clamp(coord, 0, size - 1), & \text{for NEAREST filtering} \\ clamp(coord, -1, size), & \text{for LINEAR filtering} \end{cases}$
`CLAMP_TO_EDGE`	$clamp(coord, 0, size - 1)$
`CLAMP_TO_BORDER`	$clamp(coord, -1, size)$
`REPEAT`	$fmod(coord, size)$
`MIRRORED_REPEAT`	$(size - 1) - mirror(fmod(coord, 2 \times size) - size)$

Table 3.24: Texel location wrap mode application. $fmod(a, b)$ returns $a - b \times \lfloor \frac{a}{b} \rfloor$. $mirror(a)$ returns a if $a \geq 0$, and $-(1 + a)$ otherwise. The values of $mode$ and $size$ are `TEXTURE_WRAP_S` and w_t, `TEXTURE_WRAP_T` and h_t, and `TEXTURE_WRAP_R` and d_t when wrapping i, j, or k coordinates, respectively.

then the border values defined by `TEXTURE_BORDER_COLOR` are used in place of the non-existent texel. If the texture contains color components, the values of `TEXTURE_BORDER_COLOR` are interpreted as an RGBA color to match the texture's internal format in a manner consistent with table 3.16. 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. Border values are clamped before they are used, according to the format in which texture components are stored. For signed and unsigned normalized fixed-point formats, border values are clamped to $[-1, 1]$ and $[0, 1]$, respectively. For floating-point and integer formats, border values are clamped to the representable range of the format. If the texture contains depth components, the first component of `TEXTURE_BORDER_COLOR` is interpreted as a depth value.

When the value of `TEXTURE_MIN_FILTER` is `LINEAR`, a $2 \times 2 \times 2$ cube of texels in the image array of level $level_{base}$ is selected. Let

$$\begin{aligned}
i_0 &= wrap(\lfloor u' - 0.5 \rfloor) \\
j_0 &= wrap(\lfloor v' - 0.5 \rfloor) \\
k_0 &= wrap(\lfloor w' - 0.5 \rfloor) \\
i_1 &= wrap(\lfloor u' - 0.5 \rfloor + 1) \\
j_1 &= wrap(\lfloor v' - 0.5 \rfloor + 1) \\
k_1 &= wrap(\lfloor w' - 0.5 \rfloor + 1) \\
\alpha &= frac(u' - 0.5) \\
\beta &= frac(v' - 0.5) \\
\gamma &= frac(w' - 0.5)
\end{aligned}$$

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

For a three-dimensional texture, the texture value τ is found as

$$\begin{aligned}
\tau = {} & (1-\alpha)(1-\beta)(1-\gamma)\tau_{i_0 j_0 k_0} + \alpha(1-\beta)(1-\gamma)\tau_{i_1 j_0 k_0} \\
& + (1-\alpha)\beta(1-\gamma)\tau_{i_0 j_1 k_0} + \alpha\beta(1-\gamma)\tau_{i_1 j_1 k_0} \\
& + (1-\alpha)(1-\beta)\gamma\tau_{i_0 j_0 k_1} + \alpha(1-\beta)\gamma\tau_{i_1 j_0 k_1} \\
& + (1-\alpha)\beta\gamma\tau_{i_0 j_1 k_1} + \alpha\beta\gamma\tau_{i_1 j_1 k_1}
\end{aligned} \tag{3.24}$$

where τ_{ijk} is the texel at location (i, j, k) in the three-dimensional texture image.

For a two-dimensional, two-dimensional array, rectangular, or cube map texture,

$$\begin{aligned}
\tau = {} & (1-\alpha)(1-\beta)\tau_{i_0 j_0} + \alpha(1-\beta)\tau_{i_1 j_0} \\
& + (1-\alpha)\beta\tau_{i_0 j_1} + \alpha\beta\tau_{i_1 j_1}
\end{aligned}$$

where τ_{ij} is the texel at location (i, j) in the two-dimensional texture image. For two-dimensional array textures, all texels are obtained from layer l, where

$$l = clamp(\lfloor r + 0.5 \rfloor, 0, d_t - 1).$$

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

$$\tau = (1-\alpha)\tau_{i_0} + \alpha\tau_{i_1}$$

where τ_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

$$l = clamp(\lfloor t + 0.5 \rfloor, 0, h_t - 1).$$

For any texel in the equation above that refers to a border texel outside the defined range of the image, the texel value is taken from the texture border color as with `NEAREST` filtering.

Rendering Feedback Loops

If all of the following conditions are satisfied, then the value of the selected τ_{ijk}, τ_{ij}, or τ_i in the above equations is undefined instead of referring to the value of the texel at location (i, j, k), (i, j), or (i) respectively. This situation is discussed in more detail in the description of feedback loops in section 4.4.3.

- The current `DRAW_FRAMEBUFFER_BINDING` names a framebuffer object *F*.
- The texture is attached to one of the attachment points, *A*, of framebuffer object *F*.
- The value of `TEXTURE_MIN_FILTER` is `NEAREST` or `LINEAR`, and the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL` for attachment point *A* is equal to the value of `TEXTURE_BASE_LEVEL`

  -or-

  The value of `TEXTURE_MIN_FILTER` is `NEAREST_MIPMAP_NEAREST`, `NEAREST_MIPMAP_LINEAR`, `LINEAR_MIPMAP_NEAREST`, or `LINEAR_MIPMAP_LINEAR`, and the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL` for attachment point *A* is within the the inclusive range from `TEXTURE_BASE_LEVEL` to q.

Mipmapping

`TEXTURE_MIN_FILTER` values `NEAREST_MIPMAP_NEAREST`, `NEAREST_MIPMAP_LINEAR`, `LINEAR_MIPMAP_NEAREST`, and `LINEAR_MIPMAP_LINEAR` each require the use of a *mipmap*. Rectangular textures do not support mipmapping (it is an error to specify a minification filter that requires mipmapping). A mipmap is an ordered set of arrays representing the same image; each array has a resolution lower than the previous one. If the image array of level $level_{base}$ (excluding its border) has dimensions $w_t \times h_t \times d_t$, then there are $\lfloor \log_2(maxsize) \rfloor + 1$ levels in the mipmap. where

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

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

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

where

$$w_d = 2^i$$
$$h_d = \begin{cases} 1, & \text{for 1D and 1D array textures} \\ 2^i, & \text{otherwise} \end{cases}$$
$$d_d = \begin{cases} 2^i, & \text{for 3D textures} \\ 1, & \text{otherwise} \end{cases}$$

until the last array is reached with dimension $1 \times 1 \times 1$.

Each array in a mipmap is defined using **TexImage3D**, **TexImage2D**, **CopyTexImage2D**, **TexImage1D**, or **CopyTexImage1D**; the array being set is indicated with the level-of-detail argument *level*. Level-of-detail numbers proceed from $level_{base}$ for the original texel array through $p = \lfloor \log_2(maxsize) \rfloor + level_{base}$ with each unit increase indicating an array of half the dimensions of the previous one (rounded down to the next integer if fractional) as already described. All arrays from $level_{base}$ through $q = \min\{p, level_{max}\}$ must be defined, as discussed in section 3.9.12.

The values of $level_{base}$ and $level_{max}$ may be respecified for a specific texture by calling **TexParameter[if]** with *pname* set to `TEXTURE_BASE_LEVEL` or `TEXTURE_MAX_LEVEL` respectively.

The error `INVALID_VALUE` is generated if either value is negative.

The mipmap is used in conjunction with the level of detail to approximate the application of an appropriately filtered texture to a fragment. Let c be the value of λ at which the transition from minification to magnification occurs (since this discussion pertains to minification, we are concerned only with values of λ where $\lambda > c$).

For mipmap filters `NEAREST_MIPMAP_NEAREST` and `LINEAR_MIPMAP_NEAREST`, the dth mipmap array is selected, where

$$d = \begin{cases} level_{base}, & \lambda \le \frac{1}{2} \\ \lceil level_{base} + \lambda + \frac{1}{2} \rceil - 1, & \lambda > \frac{1}{2}, level_{base} + \lambda \le q + \frac{1}{2} \\ q, & \lambda > \frac{1}{2}, level_{base} + \lambda > q + \frac{1}{2} \end{cases} \qquad (3.25)$$

The rules for `NEAREST` or `LINEAR` filtering are then applied to the selected array. Specifically, the coordinate (u, v, w) is computed as in equation 3.21, with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is d.

For mipmap filters `NEAREST_MIPMAP_LINEAR` and `LINEAR_MIPMAP_LINEAR`, the level d_1 and d_2 mipmap arrays are selected, where

$$d_1 = \begin{cases} q, & level_{base} + \lambda \ge q \\ \lfloor level_{base} + \lambda \rfloor, & \text{otherwise} \end{cases} \qquad (3.26)$$

$$d_2 = \begin{cases} q, & level_{base} + \lambda \ge q \\ d_1 + 1, & \text{otherwise} \end{cases} \qquad (3.27)$$

The rules for `NEAREST` or `LINEAR` filtering are then applied to each of the selected arrays, yielding two corresponding texture values τ_1 and τ_2. Specifically, for level d_1, the coordinate (u, v, w) is computed as in equation 3.21, 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 3.21, with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is d_2.

The final texture value is then found as

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

Manual Mipmap Generation

Mipmaps can be generated manually with the command

`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 3.9.12.

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

The internal formats and border widths of the derived mipmap arrays all match those of the $level_{base}$ array, and the dimensions of the derived arrays follow the requirements described in section 3.9.12.

The contents of the derived arrays are computed by repeated, filtered reduction of the $level_{base}$ array. For one- and two-dimensional array textures, each layer is filtered independently. No particular filter algorithm is required, though a box filter is recommended as the default filter. In some implementations, filter quality may be affected by hints (section 5.7).

Automatic Mipmap Generation

If the value of texture parameter `GENERATE_MIPMAP` is `TRUE`, and a change is made to the interior or border texels of the $level_{base}$ 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 3.9.12) will be computed. Array levels $level_{base} + 1$ through p are replaced with arrays derived from the modified $level_{base}$ array, as described above for **Manual Mipmap Generation**. All other mipmap arrays, including the $level_{base}$ array, are left unchanged by this computation. For arrays in the range $level_{base} + 1$ through q, inclusive, automatic and manual mipmap generation generate the same derived arrays, given identical $level_{base}$ arrays.

Automatic mipmap generation is available only for non-proxy texture image targets.

3.9.10 Texture Magnification

When λ 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

---

[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.

`TEXTURE_MIN_FILTER` as described in section 3.9.9, including the texture coordinate wrap modes specified in table 3.24. The level-of-detail $level_{base}$ texel array is always used for magnification.

Implementations may either unconditionally assume $c = 0$ for the minification vs. magnification switch-over point, or may choose to make c depend on the combination of minification and magnification modes as follows: if the magnification filter is given by `LINEAR` and the minification filter is given by `NEAREST_MIPMAP_NEAREST` or `NEAREST_MIPMAP_LINEAR`, then $c = 0.5$. This is done to ensure that a minified texture does not appear "sharper" than a magnified texture. Otherwise $c = 0$.

3.9.11 Combined Depth/Stencil Textures

If the texture image has a base internal format of `DEPTH_STENCIL`, then the stencil index texture component is ignored. The texture value τ does not include a stencil index component, but includes only the depth component.

3.9.12 Texture Completeness

A texture is said to be *complete* if all the image arrays and texture parameters required to utilize the texture for texture application are consistently defined. The definition of completeness varies depending on texture dimensionality and type.

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

- The set of mipmap arrays $level_{base}$ through q (where q is defined in the **Mipmapping** discussion of section 3.9.9) 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 3.9.9.
- $level_{base} \leq level_{max}$

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

A cube map texture is mipmap complete if each of the six texture images, considered individually, is mipmap complete. Additionally, a cube map texture is *cube complete* if the following conditions all hold true:

- The $level_{base}$ 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.

Using the preceding definitions, a texture is complete unless any of the following conditions hold true:

- Any dimension of the $level_{base}$ array is not positive. For a rectangular or multisample texture, $level_{base}$ is always zero.
- The texture is a cube map texture, and is not cube complete.
- The minification filter requires a mipmap (is neither `NEAREST` nor `LINEAR`), and the texture is not mipmap complete.
- The internal format of the texture arrays is integer (see tables 3.17- 3.18), and either the magnification filter is not `NEAREST`, or the minification filter is neither `NEAREST` nor `NEAREST_MIPMAP_NEAREST`.

Effects of Completeness on Texture Application

Texture lookup and texture fetch operations performed in vertex, geometry, and fragment shaders are affected by completeness of the texture being sampled as described in sections 2.14.4 and 3.12.2.

For fixed-function texture access, if texturing is enabled for a texture unit at the time a primitive is rasterized, 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.

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 with $level_{base} = 0$ and $level_{max} = 1000$.

3.9.13 Texture State and Proxy State

The state necessary for texture can be divided into two categories. First, there are the multiple sets of texel arrays (a single array for the rectangular texture target; one set of mipmap arrays each for the one-, two-, and three-dimensional and one- and two-dimensional array texture targets; and six sets of mipmap arrays for the cube map texture targets) and their number. Each array has associated with it a width, height (two- and three-dimensional, rectangular, one-dimensional array, and cube map only), and depth (three-dimensional and two-dimensional array only), a border width, an integer describing the internal format of the image, integer values describing the resolutions of each of the red, green, blue, alpha, luminance, intensity, depth, and stencil components of the image, integer values describing the type (unsigned normalized, integer, floating-point, etc.) of each of the components, a boolean describing whether the image is compressed or not, and an integer size of a compressed image. Each initial texel array is null (zero width, height, and depth, zero border width, internal format 1, component sizes set to zero and component types set to `NONE`, the compressed flag set to `FALSE`, and a zero compressed size). Multisample textures contain an integer identifying the number of samples in each texel, and a boolean indicating whether identical sample locations and the same number of samples will be used for all texels in the image. The buffer texture target has associated an integer containing the name of the buffer object that provided the data store for the texture, initially zero, and an integer identifying the internal format of the texture, initially `LUMINANCE8`.

Next, there are the four sets of texture properties, corresponding to the one-, two-, three-dimensional, and cube map texture targets. Each set consists of the selected minification and magnification filters, the wrap modes for s, t (two- and three-dimensional and cube map only), and r (three-dimensional only), the `TEXTURE_BORDER_COLOR`, two floating-point numbers describing the minimum and maximum level of detail, two integers describing the base and maximum mipmap array, a boolean flag indicating whether the texture is resident, a boolean indicating whether automatic mipmap generation should be performed, the priority associated with each set of properties, and three integers describing the depth texture mode, compare mode, and compare function. 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 3.9.14). In the initial state, the value assigned to `TEXTURE_MIN_FILTER` is `NEAREST_MIPMAP_LINEAR` (except for rectangular textures, where the initial value is `LINEAR`), and the value for `TEXTURE_MAG_FILTER` is `LINEAR`. s, t, and r wrap modes are all set to `REPEAT` (except for rectangular textures, where the initial value is `CLAMP_TO_EDGE`). The values of `TEXTURE_MIN_LOD` and `TEXTURE_MAX_LOD` are -1000 and 1000 respectively. The values of `TEXTURE_BASE_LEVEL` and `TEXTURE_MAX_LEVEL` are 0 and 1000 respectively. The value of `TEXTURE_PRIORITY` is 1.0. The value of `TEXTURE_BORDER_COLOR` is (0,0,0,0). The value of `GENERATE_MIPMAP` is false. The values of `DEPTH_TEXTURE_MODE`, `TEXTURE_COMPARE_MODE`, and `TEXTURE_COMPARE_FUNC` are `LUMINANCE`, `NONE`, and `LEQUAL` respectively. The initial value of `TEXTURE_RESIDENT` is determined by the GL.

In addition to image arrays for the non-proxy texture targets described above, partially instantiated image arrays are maintained for one-, two-, and three-dimensional, rectangular, and one- and two-dimensional array textures. Additionally, a single proxy image array is maintained for the cube map texture. Each proxy image array includes width, height, depth, 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 **TexImage3D** is executed with *target* specified as `PROXY_TEXTURE_3D`, the three-dimensional proxy state values of the specified level-of-detail are recomputed and updated. If the image array would not be supported by **TexImage3D** called with *target* set to `TEXTURE_3D`, no error is generated, but the proxy width, height, depth, border width, and component resolutions are set to zero, and the component types are set to `NONE`. If the image array would be supported by such a call to **TexImage3D**, the proxy state values are set exactly as though the actual image array were being specified. No pixel data are transferred or processed in either case.

Proxy arrays for one- and two-dimensional textures and one- and two-dimensional array textures are operated on in the same way when **TexImage1D** is executed with *target* specified as `PROXY_TEXTURE_1D`, **TexImage2D** is executed with *target* specified as `PROXY_TEXTURE_2D`, `PROXY_TEXTURE_1D_ARRAY`, or `PROXY_TEXTURE_RECTANGLE`, or **TexImage3D** is executed with *target* specified as `PROXY_TEXTURE_2D_ARRAY`.

Proxy arrays for two-dimensional multisample and two-dimensional multisample array textures are operated on in the same way when **TexImage2DMultisample** is called with *target* specified as `PROXY_TEXTURE_2D_MULTISAMPLE`, or **TexImage3DMultisample** is called with *target* specified as `PROXY_TEXTURE_2D_MULTISAMPLE_ARRAY`.

The cube map proxy arrays are operated on in the same manner when **TexImage2D** is executed with the *target* field specified as `PROXY_TEXTURE_CUBE_MAP`, with the addition that determining that a given cube map texture is supported with `PROXY_TEXTURE_CUBE_MAP` indicates that all six of the cube map 2D images are supported. Likewise, if the specified `PROXY_TEXTURE_CUBE_MAP` is not supported, none of the six cube map 2D images are supported.

There is no image or non-level-related state associated with proxy textures. Therefore they may not be used as textures, and calling **BindTexture**, **GetTexImage**, **GetTexParameteriv**, or **GetTexParameterfv** with a proxy texture *target* generates an `INVALID_ENUM` error.

3.9.14 Texture Objects

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

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

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

with *target* set to the desired texture target and *texture* set to the unused name. The resulting texture object is a new state vector, comprising all the state values listed in section 3.9.13, set to the same initial values. The new texture object bound to *target* is, and remains a texture of the dimensionality and type specified by *target* until it is deleted.

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

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

Texture objects are deleted by calling

`void` **DeleteTextures**(`sizei` *n*, `uint` **textures*);

textures contains *n* names of texture objects to be deleted. After a texture object is deleted, it has no contents or dimensionality, and its name is again unused. If a texture that is currently bound to any of the *target* bindings of **BindTexture** is deleted, it is as though **BindTexture** had been executed with the same *target* and *texture* zero. Additionally, special care must be taken when deleting a texture if any of the images of the texture are attached to a framebuffer object. See section 4.4.2 for details.

Unused names in *textures* are silently ignored, as is the value zero.

The command

`void` **GenTextures**(`sizei` *n*, `uint` **textures*);

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

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

`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

`void` **PrioritizeTextures**(`sizei` *n*, `uint` **textures*,
`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, rectangular, buffer, cube map, two-dimensional multisample, and two-dimensional multisample array texture objects, is shared among all texture units. A texture object may be bound to more than one texture unit simultaneously. After a texture object is bound, any GL operations on that target object affect any other texture units to which the same texture object is bound.

Texture binding is affected by the setting of the state `ACTIVE_TEXTURE`.

If a texture object is deleted, it as if all texture units which are bound to that texture object are rebound to texture object zero.

3.9.15 Texture Environments and Texture Functions

The command

`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 `TEXTURE_FILTER_CONTROL`, `POINT_SPRITE`, or `TEXTURE_ENV`. *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 `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 λ as described in section 3.9.9.

When *target* is `POINT_SPRITE`, point sprite rasterization behavior is affected as described in section 3.4.

When *target* is `TEXTURE_ENV`, the possible environment parameters are `TEXTURE_ENV_MODE`, `TEXTURE_ENV_COLOR`, `COMBINE_RGB`, `COMBINE_ALPHA`, `RGB_SCALE`, `ALPHA_SCALE`, `SRC`n`_RGB`, `SRC`n`_ALPHA`, `OPERAND`n`_RGB`, and `OPERAND`n`_ALPHA`, where n = 0, 1, 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 described in equation 2.2.

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[4] 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 3.25; C_c and A_c are the components of the texture environment color; C_p and A_p are the 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 3.26, 3.27, and 3.28.

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 3.28. 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

---

[4] In the remainder of section 3.9.15, 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.

Texture Base Internal Format	Texture source color C_s	A_s
ALPHA	$(0, 0, 0)$	A_t
LUMINANCE	(L_t, L_t, L_t)	1
LUMINANCE_ALPHA	(L_t, L_t, L_t)	A_t
INTENSITY	(I_t, I_t, I_t)	I_t
RED	$(R_t, 0, 0)$	1
RG	$(R_t, G_t, 0)$	1
RGB	(R_t, G_t, B_t)	1
RGBA	(R_t, G_t, B_t)	A_t

Table 3.25: Correspondence of filtered texture components to texture source components.

Texture Base Internal Format	REPLACE Function	MODULATE Function	DECAL Function
ALPHA	$C_v = C_p$ $A_v = A_s$	$C_v = C_p$ $A_v = A_pA_s$	*undefined*
LUMINANCE (or 1)	$C_v = C_s$ $A_v = A_p$	$C_v = C_pC_s$ $A_v = A_p$	*undefined*
LUMINANCE_ALPHA (or 2)	$C_v = C_s$ $A_v = A_s$	$C_v = C_pC_s$ $A_v = A_pA_s$	*undefined*
INTENSITY	$C_v = C_s$ $A_v = A_s$	$C_v = C_pC_s$ $A_v = A_pA_s$	*undefined*
RGB, RG, RED, or 3	$C_v = C_s$ $A_v = A_p$	$C_v = C_pC_s$ $A_v = A_p$	$C_v = C_s$ $A_v = A_p$
RGBA or 4	$C_v = C_s$ $A_v = A_s$	$C_v = C_pC_s$ $A_v = A_pA_s$	$C_v = C_p(1 - A_s) + C_sA_s$ $A_v = A_p$

Table 3.26: Texture functions REPLACE, MODULATE, and DECAL.

Texture Base Internal Format	`BLEND` Function	`ADD` Function
`ALPHA`	$C_v = C_p$ $A_v = A_pA_s$	$C_v = C_p$ $A_v = A_pA_s$
`LUMINANCE` (or 1)	$C_v = C_p(1 - C_s) + C_cC_s$ $A_v = A_p$	$C_v = C_p + C_s$ $A_v = A_p$
`LUMINANCE_ALPHA` (or 2)	$C_v = C_p(1 - C_s) + C_cC_s$ $A_v = A_pA_s$	$C_v = C_p + C_s$ $A_v = A_pA_s$
`INTENSITY`	$C_v = C_p(1 - C_s) + C_cC_s$ $A_v = A_p(1 - A_s) + A_cA_s$	$C_v = C_p + C_s$ $A_v = A_p + A_s$
`RGB`, `RG`, `RED`, or 3	$C_v = C_p(1 - C_s) + C_cC_s$ $A_v = A_p$	$C_v = C_p + C_s$ $A_v = A_p$
`RGBA` or 4	$C_v = C_p(1 - C_s) + C_cC_s$ $A_v = A_pA_s$	$C_v = C_p + C_s$ $A_v = A_pA_s$

Table 3.27: Texture functions `BLEND` and `ADD`.

clamping is enabled, the arguments and results used in table 3.28 are clamped to $[0, 1]$. Otherwise, the results are unmodified.

The arguments $Arg0$, $Arg1$, and $Arg2$ are determined by the values of `SRC`n`_RGB`, `SRC`n`_ALPHA`, `OPERAND`n`_RGB` and `OPERAND`n`_ALPHA`, where n = 0, 1, or 2, as shown in tables 3.29 and 3.30. ${C_s}^n$ 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 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.

COMBINE_RGB	Texture Function
REPLACE	$Arg0$
MODULATE	$Arg0 * Arg1$
ADD	$Arg0 + Arg1$
ADD_SIGNED	$Arg0 + Arg1 - 0.5$
INTERPOLATE	$Arg0 * Arg2 + Arg1 * (1 - Arg2)$
SUBTRACT	$Arg0 - Arg1$
DOT3_RGB	$4 \times ((Arg0_r - 0.5) * (Arg1_r - 0.5) + (Arg0_g - 0.5) * (Arg1_g - 0.5) + (Arg0_b - 0.5) * (Arg1_b - 0.5))$
DOT3_RGBA	$4 \times ((Arg0_r - 0.5) * (Arg1_r - 0.5) + (Arg0_g - 0.5) * (Arg1_g - 0.5) + (Arg0_b - 0.5) * (Arg1_b - 0.5))$

COMBINE_ALPHA	Texture Function
REPLACE	$Arg0$
MODULATE	$Arg0 * Arg1$
ADD	$Arg0 + Arg1$
ADD_SIGNED	$Arg0 + Arg1 - 0.5$
INTERPOLATE	$Arg0 * Arg2 + Arg1 * (1 - Arg2)$
SUBTRACT	$Arg0 - Arg1$

Table 3.28: 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.

SRCn_RGB	OPERANDn_RGB	Argument
TEXTURE	SRC_COLOR	C_s
	ONE_MINUS_SRC_COLOR	$1 - C_s$
	SRC_ALPHA	A_s
	ONE_MINUS_SRC_ALPHA	$1 - A_s$
TEXTUREn	SRC_COLOR	${C_s}^n$
	ONE_MINUS_SRC_COLOR	$1 - {C_s}^n$
	SRC_ALPHA	${A_s}^n$
	ONE_MINUS_SRC_ALPHA	$1 - {A_s}^n$
CONSTANT	SRC_COLOR	C_c
	ONE_MINUS_SRC_COLOR	$1 - C_c$
	SRC_ALPHA	Ac
	ONE_MINUS_SRC_ALPHA	$1 - A_c$
PRIMARY_COLOR	SRC_COLOR	C_f
	ONE_MINUS_SRC_COLOR	$1 - C_f$
	SRC_ALPHA	A_f
	ONE_MINUS_SRC_ALPHA	$1 - A_f$
PREVIOUS	SRC_COLOR	C_p
	ONE_MINUS_SRC_COLOR	$1 - C_p$
	SRC_ALPHA	A_p
	ONE_MINUS_SRC_ALPHA	$1 - A_p$

Table 3.29: Arguments for `COMBINE_RGB` functions.

SRCn_ALPHA	OPERANDn_ALPHA	Argument
TEXTURE	SRC_ALPHA	A_s
	ONE_MINUS_SRC_ALPHA	$1 - A_s$
TEXTUREn	SRC_ALPHA	${A_s}^n$
	ONE_MINUS_SRC_ALPHA	$1 - {A_s}^n$
CONSTANT	SRC_ALPHA	A_c
	ONE_MINUS_SRC_ALPHA	$1 - A_c$
PRIMARY_COLOR	SRC_ALPHA	A_f
	ONE_MINUS_SRC_ALPHA	$1 - A_f$
PREVIOUS	SRC_ALPHA	A_p
	ONE_MINUS_SRC_ALPHA	$1 - A_p$

Table 3.30: Arguments for `COMBINE_ALPHA` functions.

3.9.16 Texture Comparison Modes

Texture values can also be computed according to a specified comparison function. Texture parameter `TEXTURE_COMPARE_MODE` specifies the comparison operands, and parameter `TEXTURE_COMPARE_FUNC` specifies the comparison function. 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 `DEPTH_TEXTURE_MODE`, `TEXTURE_COMPARE_MODE` and `TEXTURE_COMPARE_FUNC` control the output of the texture unit as described below. Otherwise, the texture unit operates in the normal manner and texture comparison is bypassed.

Let D_t be the depth texture value and D_{ref} be the reference value, 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 texture 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

$$r = D_t$$

If the value of `TEXTURE_COMPARE_MODE` is `COMPARE_REF_TO_TEXTURE`, then r depends on the texture comparison function as shown in table 3.31.

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

Texture Comparison Function	Computed result r
`LEQUAL`	$r = \begin{cases} 1.0, & D_{ref} \leq D_t \\ 0.0, & D_{ref} > D_t \end{cases}$
`GEQUAL`	$r = \begin{cases} 1.0, & D_{ref} \geq D_t \\ 0.0, & D_{ref} < D_t \end{cases}$
`LESS`	$r = \begin{cases} 1.0, & D_{ref} < D_t \\ 0.0, & D_{ref} \geq D_t \end{cases}$
`GREATER`	$r = \begin{cases} 1.0, & D_{ref} > D_t \\ 0.0, & D_{ref} \leq D_t \end{cases}$
`EQUAL`	$r = \begin{cases} 1.0, & D_{ref} = D_t \\ 0.0, & D_{ref} \neq D_t \end{cases}$
`NOTEQUAL`	$r = \begin{cases} 1.0, & D_{ref} \neq D_t \\ 0.0, & D_{ref} = D_t \end{cases}$
`ALWAYS`	$r = 1.0$
`NEVER`	$r = 0.0$

Table 3.31: Depth texture comparison functions.

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.

3.9.17 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`, `SLUMINANCE`, `SLUMINANCE8`, `COMPRESSED_SLUMINANCE`, `COMPRESSED_SLUMINANCE_ALPHA`, `COMPRESSED_SRGB`, or `COMPRESSED_SRGB_ALPHA`, the red, green, and blue components are converted from an sRGB color space to a linear color space as part of filtering described in sections 3.9.9 and 3.9.10. Any alpha component is left unchanged. Ideally, implementations should perform this color conversion on each sample prior to filtering but implementations are allowed to perform this conversion after filtering (though this post-filtering approach is inferior to converting from sRGB prior to filtering).

The conversion from an sRGB encoded component, c_s, to a linear component,

c_l, is as follows.

$$c_l = \begin{cases} \frac{c_s}{12.92}, & c_s \leq 0.04045 \\ \left(\frac{c_s+0.055}{1.055}\right)^{2.4}, & c_s > 0.04045 \end{cases} \qquad (3.28)$$

Assume c_s is the sRGB component in the range $[0, 1]$.

3.9.18 Shared Exponent Texture Color Conversion

If the currently bound texture's internal format is `RGB9_E5`, the red, green, blue, and shared bits are converted to color components (prior to filtering) using shared exponent decoding. The component red_s, $green_s$, $blue_s$, and exp_{shared} values (see section 3.9.1) are treated as unsigned integers and are converted to red, $green$, and $blue$ as follows:

$$red = red_s 2^{exp_{shared}-B}$$
$$green = green_s 2^{exp_{shared}-B}$$
$$blue = blue_s 2^{exp_{shared}-B}$$

3.9.19 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 3.9.8 through 3.9.10. 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 `SRC`n`_RGB`, `SRC`n`_ALPHA`, `OPERAND`n`_RGB` and `OPERAND`n`_ALPHA`. If `TEXTURE`n is specified as `SRC`n`_RGB` or `SRC`n`_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 3.9.12) 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 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 intial 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 $\mathbf{c}_{pri}$ (which texturing, if enabled, may have modified) and a secondary color $\mathbf{c}_{sec}$.

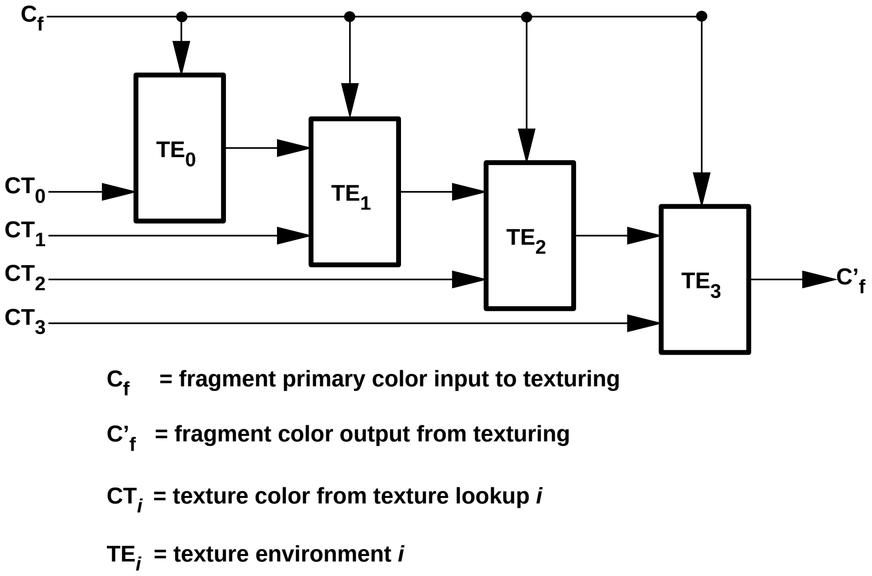

C_f
TE_0
TE_1
TE_2
TE_3
CT_0
CT_1
CT_2
CT_3
C'_f
C_f = fragment primary color input to texturing
C'_f = fragment color output from texturing
CT_i = texture color from texture lookup i
TE_i = texture environment i

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.

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 $\mathbf{c}$. The A component of $\mathbf{c}$ is taken from the A component of $\mathbf{c}_{pri}$; the A component of $\mathbf{c}_{sec}$ is unused. If color sum is disabled, then $\mathbf{c}_{pri}$ is assigned to $\mathbf{c}$. If fragment color clamping is enabled, the components of $\mathbf{c}$ are then clamped to the range $[0, 1]$.

Color sum is enabled or disabled using the generic **Enable** and **Disable** commands, respectively, with the symbolic constant `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 `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 **Enable** and **Disable** commands using the symbolic constant `FOG`.

This factor f is computed according to one of three equations:

$$f = \exp(-d \cdot c), \tag{3.29}$$

$$f = \exp(-(d \cdot c)^2), \text{ or} \tag{3.30}$$

$$f = \frac{e - c}{e - s} \tag{3.31}$$

If a vertex or geometry shader is active, or if the fog source, as defined below, is `FOG_COORD`, then c is the interpolated value of the fog coordinate for this fragment. Otherwise, if the fog source is `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

`void` **Fog**{**if**}`(` `enum` *pname*`,` `T` *param* `);`
`void` **Fog**{**if**}**v**`(` `enum` *pname*`,` `T` *params* `);`

If *pname* is `FOG_MODE`, then *param* must be, or *params* must point to an integer that is one of the symbolic constants `EXP`, `EXP2`, or `LINEAR`, in which case equation 3.29, 3.30, or 3.31, respectively, is selected for the fog calculation (if, when 3.31 is selected, $e = s$, results are undefined). If *pname* is `FOG_COORD_SRC`, then *param* must be, or *params* must point to an integer that is one of the symbolic constants `FRAGMENT_DEPTH` or `FOG_COORD`. If *pname* is `FOG_DENSITY`, `FOG_START`, or `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 `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 **Fog** with *pname* equal to `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 as described in equation 2.2. 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 a color in the color index buffer (buffers are discussed in chapter 4). The value of i_f is set by calling **Fog** with *pname* set to `FOG_INDEX` and *param* being or *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, `FOG_COORD_SRC` is `FRAGMENT_DEPTH`, `FOG_MODE` is `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 *fragment shader*.

A fragment shader is an array of strings containing source code for the operations that are meant to occur on each fragment that results from rasterization. The language used for fragment shaders is described in the OpenGL Shading Language Specification.

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 2.14.1 using a *type* parameter of `FRAGMENT_SHADER`. They are attached to and used in program objects as described in section 2.14.2.

When the program object currently in use includes a fragment shader, its fragment shader is considered *active*, and is used to process fragments. If the program object has no fragment shader, or no program object is currently in use, the fixed-function fragment processing operations described in previous sections are used instead.

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 in the default uniform block is specified by the value of the implementation-dependent constant `MAX_FRAGMENT_UNIFORM_COMPONENTS`. The total amount of combined storage available for fragment shader uniform variables in all uniform blocks (including the default uniform block) is specified by the value of the implementation-dependent constant `MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS`. These values represent the numbers of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a fragment shader. A uniform matrix will consume no more than $4 \times min(r, c)$ such values, where r and c are the number of rows and columns in the matrix. A link error will be generated if an attempt is made to utilize more than the space available for fragment shader uniform variables.

Fragment shaders can read varying variables that correspond to the attributes of the fragments produced by rasterization. The OpenGL Shading Language Specification defines a set of built-in varying variables that can be be accessed by a fragment shader. These built-in varying variables include data associated with a fragment that are used for fixed-function fragment processing, such as the fragment's color, secondary color, texture coordinates, fog coordinate, eye z coordinate, and position.

Additionally, when a vertex shader is active, it may define one or more *varying* variables (see section 2.14.6 and the OpenGL Shading Language Specification). These values are, if not flat shaded, interpolated across the primitive being rendered. The results of these interpolations are available when varying variables of the same name are defined in the fragment shader.

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, 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 **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 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 3.9.15 are not applied.
- Texture application as described in section 3.9.19 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 2.14.7 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to fragment shaders.

When a texture lookup is performed in a fragment shader, the GL computes the filtered texture value τ in the manner described in sections 3.9.9 and 3.9.10, and converts it to a texture source color C_s according to table 3.25 (section 3.9.15). The GL returns a four-component vector (R_s, G_s, B_s, A_s) to the fragment shader. For the purposes of level-of-detail calculations, the derivatives $\frac{du}{dx}$, $\frac{du}{dy}$, $\frac{dv}{dx}$, $\frac{dv}{dy}$, $\frac{dw}{dx}$ and $\frac{dw}{dy}$ may be approximated by a differencing algorithm as detailed in section 8.8 of the OpenGL Shading Language Specification.

Texture lookups involving textures with depth component data can either return the depth data directly or return the results of a comparison with the D_{ref} value (see section 3.9.16) used to perform the lookup. The comparison operation is requested in the shader by using any of the shadow sampler types (`sampler1DShadow`, `sampler2DShadow`, or `sampler2DRectShadow`), and in the texture using the `TEXTURE_COMPARE_MODE` parameter. These requests must be consistent; the results of a texture lookup are undefined if:

- 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`.

Using a sampler in a fragment shader will return $(R, G, B, A) = (0, 0, 0, 1)$ if the sampler's associated texture is not complete, as defined in section 3.9.12.

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 fragment coordinate $(x_f \quad y_f \quad z_f \quad w_f)$ for the fragment. Computing the fragment coordinate depends on the fragment processing pixel-center and origin conventions (discussed below) as follows:

$$
\begin{aligned}
x_f &= \begin{cases} x_w - \frac{1}{2}, & \text{pixel-center convention is integer} \\ x_w, & \text{otherwise} \end{cases} \\
y'_f &= \begin{cases} H - y_w, & \text{origin convention is upper-left} \\ y_w, & \text{otherwise} \end{cases} \\
y_f &= \begin{cases} y'_f - \frac{1}{2}, & \text{pixel-center convention is integer} \\ y'_f & \text{otherwise} \end{cases} \\
z_f &= z_w \\
w_f &= \frac{1}{w_c}
\end{aligned}
\tag{3.32}
$$

where $\begin{pmatrix} x_w & y_w & z_w \end{pmatrix}$ is the fragment's window-space position, w_c is the w component of the fragment's clip-space position, and H is the window's height in pixels. Note that z_w already has a polygon offset added in, if enabled (see section 3.6.5). z_f must be precisely 0 or 1 in the case where z_w is either 0 or 1, respectively. The $\frac{1}{w}$ value is computed from the w_c coordinate (see section 2.16), which is the result of the product of the projection matrix and the vertex's eye coordinates.

Unless otherwise specified by layout qualifiers in the fragment shader (see section 4.3.8.1 of the OpenGL Shading Language Specification), the fragment processing pixel-center convention is half-integer and the fragment processing origin convention is lower-left.

The built-in 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 quadrilateral, polygon, or triangle primitives (including ones resulting from primitives rendered as points or lines), the determination is made by examining the sign of the area computed by equation 3.8 of section 3.6.1 (including the possible reversal of this sign controlled by **FrontFace**). If the sign is positive, fragments generated by the primitive are front-facing; otherwise, they are back-facing. All other fragments are considered front-facing.

If a geometry shader is active, the built-in variable `gl_PrimitiveID` contains the ID value emitted by the geometry shader for the provoking vertex. If no geometry shader is active, `gl_PrimitiveID` contains 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.

Restarting a primitive using the primitive restart index (see section 2.8) has no effect on the primitive ID counter.

The value of `gl_PrimitiveID` is undefined for fragments generated by `POLYGON` primitives or from **DrawPixels** or **Bitmap** commands. Additionally, `gl_PrimitiveID` is only defined under the same conditions that `gl_VertexID` is defined, as described under "Shader Inputs" in section 2.14.4.

Similarly to the limit on geometry shader output components (see section 2.15.4), there is a limit on the number of components of built-in and user-defined input varying variables that can be read by the fragment shader, given by the value of the implementation-dependent constant `MAX_FRAGMENT_INPUT_COMPONENTS`.

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

Shader Outputs

The OpenGL Shading Language Specification describes the values that may be output by a fragment shader. These outputs are split into two categories, user-defined varying out variables and the built-in variables `gl_FragColor`, `gl_FragData[n]`, and `gl_FragDepth`. If fragment color clamping is enabled and the color buffer has an unsigned normalized fixed-point, signed normalized fixed-point, 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 $[0, 1]$. 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 2.16.1). For floating-point depth buffers, conversion is not performed but clamping is. Note that the depth range computation is not applied here, only the conversion to fixed-point.

Color values written by a fragment shader may be floating-point, signed integer, or unsigned integer. If the color buffer has an signed or unsigned normalized fixed-point format, color values are assumed to be floating-point and are converted to fixed-point as described in equations 2.6 or 2.4, respectively; otherwise no type conversion is applied. If the values written by the fragment shader do not match the format(s) of the corresponding color buffer(s), the result is undefined.

Writing to `gl_FragColor` specifies the fragment color (color number zero) that will be used by subsequent stages of the pipeline. Writing to `gl_FragData[n]` specifies the value of fragment color number n. Any colors, or color components, associated with a fragment that are not written by the fragment shader are undefined. A fragment shader may not statically assign values to more than one of `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 subsequent stages of the pipeline. Otherwise, the value assigned to `gl_FragDepth` is used, and is undefined for any fragments where statements assigning a value to `gl_FragDepth` are not executed. Thus, if a shader statically assigns a value to `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

`void` **BindFragDataLocation**(`uint` *program*, `uint` *colorNumber*, `const char` **name*);

specifies that the varying out variable *name* in *program* should be bound to fragment color *colorNumber* when the program is next linked. If *name* was bound previously, its assigned binding is replaced with *colorNumber*. *name* must be a null-terminated string. The error `INVALID_VALUE` is generated if *colorNumber* is equal or greater than `MAX_DRAW_BUFFERS`. **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 `INVALID_OPERATION` is generated if *name* starts with the reserved `gl_` prefix.

When a program is linked, any varying out variables without a binding specified through **BindFragDataLocation** will automatically be bound to fragment colors by the GL. Such bindings can be queried using the command **GetFragDataLocation**. **LinkProgram** will fail if the number of active outputs is greater than the value of `MAX_DRAW_BUFFERS`. **LinkProgram** will also fail if more than one varying out variable is bound to the same number. This type of aliasing is not allowed.

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 `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

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

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

3.13 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, and only applied if the corresponding color buffer in the framebuffer has a fixed- or floating-point format.

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, and only applied if the corresponding color buffer in the framebuffer has a fixed- or floating-point format.

Chapter 4

Per-Fragment Operations and the Framebuffer

The framebuffer, whether it is the default framebuffer or a framebuffer object (see section 2.1), consists of a set of pixels arranged as a two-dimensional array. For purposes of this discussion, each pixel in the framebuffer is simply a set of some number of bits. The number of bits per pixel may vary depending on the GL implementation, the type of framebuffer selected, and parameters specified when the framebuffer was created. Creation and management of the default framebuffer is outside the scope of this specification, while creation and management of framebuffer objects is described in detail in section 4.4.

Corresponding bits from each pixel in the framebuffer are grouped together into a *bitplane*; each bitplane contains a single bit from each pixel. These bitplanes are grouped into several *logical buffers*. These are the *color*, *accumulation*, *depth*, and *stencil* buffers. The color buffer actually consists of a number of buffers, and these color buffers serve related but slightly different purposes depending on whether the GL is bound to the default framebuffer or a framebuffer object.

For the default framebuffer, the color buffers are the *front left* buffer, the *front right* buffer, the *back left* buffer, 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 choose not to provide accumulation, depth or stencil buffers. If no default framebuffer is associated with the GL context, the framebuffer is incomplete except when a framebuffer object is bound (see sections 4.4.1 and 4.4.4).

Framebuffer objects are not visible, and do not have any of the color buffers present in the default framebuffer. Instead, the buffers of an framebuffer object are specified by attaching individual textures or renderbuffers (see section 4.4) to a set of attachment points. A framebuffer object has an array of color buffer attachment points, numbered zero through n, a depth buffer attachment point, and a stencil buffer attachment point. In order to be used for rendering, a framebuffer object must be *complete*, as described in section 4.4.4. Not all attachments of a framebuffer object need to be populated.

Each pixel in a color buffer consists of 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 signed or unsigned normalized fixed-point, floating-point, or signed or unsigned integer values; all components must have the same representation. Each pixel in a depth buffer consists of a single unsigned integer value in the format described in section 2.16.1 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. 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 accumulation, color, depth, and stencil buffers is dependent on the currently bound framebuffer. For the default framebuffer, the number of bitplanes is fixed. For framebuffer objects, the number of bitplanes in a given logical buffer may change if the image attached to the corresponding attachment point changes.

The GL has two active framebuffers; the *draw framebuffer* is the destination for rendering operations, and the *read framebuffer* is the source for readback operations. The same framebuffer may be used for both drawing and reading. Section 4.4.1 describes the mechanism for controlling framebuffer usage.

The default framebuffer is initially used as the draw and read framebuffer [1], and the initial state of all provided bitplanes is undefined. The format and encoding of buffers in the draw and read framebuffers can be queried as described in section 6.1.3.

---

[1] The window system binding API may allow associating a GL context with two separate "default framebuffers" provided by the window system as the draw and read framebuffers, but if so, both default framebuffers are referred to by the name zero at their respective binding points.

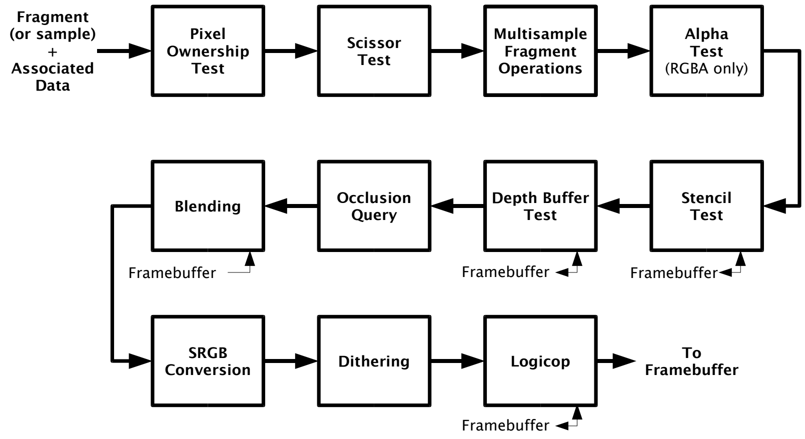


Figure 4.1. Per-fragment operations.

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 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

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

If $left \leq x_w < left + width$ and $bottom \leq y_w < bottom + height$, then the scissor test passes. Otherwise, the test fails and the fragment is discarded. The test is enabled or disabled using **Enable** or **Disable** using the constant `SCISSOR_TEST`. When disabled, it is as if the scissor test always passes. If either *width* or *height* is less than zero, then the error `INVALID_VALUE` is generated. The state required consists of four integer values and a bit indicating whether the test is enabled or disabled. In the initial state, $left = bottom = 0$. *width* and *height* are set to the width and height, respectively, of the window into which the GL is to do its rendering. If the default framebuffer is bound but no default framebuffer is associated with the GL context (see chapter 4), then *width* and *height* are initially set to zero. Initially, the scissor test is disabled.

4.1.3 Multisample Fragment Operations

This step modifies fragment alpha and coverage values based on the values of `SAMPLE_ALPHA_TO_COVERAGE`, `SAMPLE_ALPHA_TO_ONE`, `SAMPLE_COVERAGE`, `SAMPLE_COVERAGE_VALUE`, `SAMPLE_COVERAGE_INVERT`, `SAMPLE_MASK`, and `SAMPLE_MASK_VALUE`. No changes to the fragment alpha or coverage values are made at this step if `MULTISAMPLE` is disabled, or if the value of `SAMPLE_BUFFERS` is not one.

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

No specific algorithm is required for converting the sample alpha values to a temporary coverage value. It is intended that the number of 1's in the temporary coverage be proportional to the set of alpha values for the fragment, with all 1's corresponding to the maximum of all alpha values, and all 0's corresponding to all alpha values being 0. The alpha values used to generate a coverage value are clamped to the range $[0, 1]$. It is also intended that the algorithm be pseudo-random in nature, to avoid image artifacts due to regular coverage sample locations. The algorithm can and probably should be different at different pixel locations. If it does differ, it should be defined relative to window, not screen, coordinates, so that rendering results are invariant with respect to window position.

Next, if `SAMPLE_ALPHA_TO_ONE` is enabled, each alpha value is replaced by the maximum representable alpha value. Otherwise, the alpha values are not changed.

Next, if `SAMPLE_COVERAGE` is enabled, the fragment coverage is ANDed with another temporary coverage. This temporary coverage is generated in the same manner as the one described above, but as a function of the value of `SAMPLE_COVERAGE_VALUE`. The function need not be identical, but it must have the same properties of proportionality and invariance. If `SAMPLE_COVERAGE_INVERT` is `TRUE`, the temporary coverage is inverted (all bit values are inverted) before it is ANDed with the fragment coverage.

The values of `SAMPLE_COVERAGE_VALUE` and `SAMPLE_COVERAGE_INVERT` are specified by calling

`void` **SampleCoverage**(`clampf` *value*, `boolean` *invert*);

with *value* set to the desired coverage value, and *invert* set to `TRUE` or `FALSE`. *value* is clamped to [0,1] before being stored as `SAMPLE_COVERAGE_VALUE`. `SAMPLE_COVERAGE_VALUE` is queried by calling **GetFloatv** with *pname* set to `SAMPLE_COVERAGE_VALUE`. `SAMPLE_COVERAGE_INVERT` is queried by calling **GetBooleanv** with *pname* set to `SAMPLE_COVERAGE_INVERT`.

Finally, if `SAMPLE_MASK` is enabled, the fragment coverage is ANDed with the coverage value `SAMPLE_MASK_VALUE`. The value of `SAMPLE_MASK_VALUE` is specified using

`void` **SampleMaski**(`uint` *maskNumber*, `bitfield` *mask*);

with *mask* set to the desired mask for mask word *maskNumber*. `SAMPLE_MASK_VALUE` is queried by calling **GetIntegeri_v** with *pname* set to `SAMPLE_MASK_VALUE` and the index set to *maskNumber*. Bit B of mask word M corresponds to sample $32 \times M + B$ as described in section 3.3.1. The error `INVALID_VALUE` is generated if the mask word indexed is greater than or equal to the value of `MAX_SAMPLE_MASK_WORDS`.

4.1.4 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 **Enable** and **Disable** commands using the symbolic constant `ALPHA_TEST`. When disabled, it is as if the comparison always passes. The test is controlled with

`void` **AlphaFunc**(`enum` *func*, `clampf` *ref*);

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 equation 2.4 using the bit-width rule for an A component described in section 2.1.5, 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

`void` **StencilFunc**(`enum` *func*, `int` *ref*, `uint` *mask*);

`void` **StencilFuncSeparate**(`enum` *face*, `enum` *func*, `int` *ref*, `uint` *mask*);
`void` **StencilOp**(`enum` *sfail*, `enum` *dpfail*, `enum` *dppass*);
`void` **StencilOpSeparate**(`enum` *face*, `enum` *sfail*, `enum` *dpfail*, `enum` *dppass*);

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, and 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 2.13.1 and 3.6.1).

StencilFuncSeparate and **StencilOpSeparate** take a *face* argument which can be `FRONT`, `BACK`, or `FRONT_AND_BACK` and indicates which set of state is affected. **StencilFunc** and **StencilOp** set front and back stencil state to identical values.

StencilFunc and **StencilFuncSeparate** take three arguments that control whether the stencil test passes or fails. *ref* is an integer reference value that is used in the unsigned stencil comparison. Stencil comparison operations and queries of *ref* clamp its value to the range $[0, 2^s - 1]$, where s is the number of bits in the stencil buffer attached to the draw framebuffer. The s least significant bits of *mask* are bitwise ANDed with both the reference and the stored stencil value, and the resulting masked values are those that participate in the comparison controlled by *func*. *func* is a symbolic constant that determines the stencil comparison function; the eight symbolic constants are `NEVER`, `ALWAYS`, `LESS`, `LEQUAL`, `EQUAL`, `GEQUAL`, `GREATER`, or `NOTEQUAL`. Accordingly, the stencil test passes never, always, and if the masked reference value is less than, less than or equal to, equal to, greater than or equal to, greater than, or not equal to the masked stored value in the stencil buffer.

StencilOp and **StencilOpSeparate** take three arguments that indicate what happens to the stored stencil value if this or certain subsequent tests fail or pass. *sfail* indicates what action is taken if the stencil test fails. The symbolic constants are `KEEP`, `ZERO`, `REPLACE`, `INCR`, `DECR`, `INVERT`, `INCR_WRAP`, and `DECR_WRAP`. These correspond to keeping the current value, setting to zero, replacing with the reference value, incrementing with saturation, decrementing with saturation, bitwise inverting it, incrementing without saturation, and decrementing without saturation.

For purposes of increment and decrement, the stencil bits are considered as an unsigned integer. Incrementing or decrementing with saturation clamps the stencil value at 0 and the maximum representable value. Incrementing or decrementing without saturation will wrap such that incrementing the maximum representable value results in 0, and decrementing 0 results in the maximum representable value.

The same symbolic values are given to indicate the stencil action if the depth buffer test (see section 4.1.6) fails (*dpfail*), or if it passes (*dppass*).

If the stencil test fails, the incoming fragment is discarded. The state required consists of the most recent values passed to **StencilFunc** or **StencilFuncSeparate** and to **StencilOp** or **StencilOpSeparate**, and a bit indicating whether stencil testing is enabled or disabled. In the initial state, stenciling is disabled, the front and back stencil reference value are both zero, the front and back stencil comparison functions are both `ALWAYS`, and the front and back stencil mask are both set to the value $2^s - 1$, where s is greater than or equal to the number of bits in the deepest stencil buffer supported by the GL implementation. Initially, all three front and back stencil operations are `KEEP`.

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

`void` **DepthFunc**(`enum` *func*);

This command takes a single symbolic constant: one of `NEVER`, `ALWAYS`, `LESS`, `LEQUAL`, `EQUAL`, `GREATER`, `GEQUAL`, `NOTEQUAL`. Accordingly, the depth buffer test passes never, always, if the incoming fragment's z_w value is less than, less than or equal to, equal to, greater than, greater than or equal to, or not equal to the depth value stored at the location given by the incoming fragment's (x_w, y_w) coordinates.

If depth clamping (see section 2.22) is enabled, before the incoming fragment's z_w is compared z_w is clamped to the range $[min(n, f), max(n, f)]$, where n and f are the current near and far depth range values (see section 2.16.1)

If the depth buffer test fails, the incoming fragment is discarded. The stencil value at the fragment's (x_w, y_w) coordinates is updated according to the function currently in effect for depth buffer test failure. Otherwise, the fragment continues to the next operation and the value of the depth buffer at the fragment's (x_w, y_w) location is set to the fragment's z_w value. In this case the stencil value is updated according to the function currently in effect for depth buffer test success.

The necessary state is an eight-valued integer and a single bit indicating whether depth buffering is enabled or disabled. In the initial state the function is `LESS` and the test is disabled.

If there is no depth buffer, it is as if the depth buffer test always passes.

4.1.7 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 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

```
void Enablei( enum target, uint index );
void Disablei( enum target, uint index );
```

target is the symbolic constant `BLEND` and *index* is an integer i specifying the draw buffer associated with the symbolic constant `DRAW_BUFFER`i. If the color buffer associated with `DRAW_BUFFER`i is one of `FRONT`, `BACK`, `LEFT`, `RIGHT`, or `FRONT_AND_BACK` (specifying multiple color buffers), then the state enabled or disabled is applicable for all of the buffers. Blending can be enabled or disabled for all draw buffers using **Enable** or **Disable** with the symbolic constant `BLEND`. If blending is disabled for a particular draw buffer, or if logical operation on color values is enabled (section 4.1.11), 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

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

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

Signed or unsigned normalized fixed-point destination (framebuffer) components are represented as described in section 2.1.5. Constant color components, floating-point destination components, and source (fragment) components are taken to be floating point values. If source components are represented internally by the GL as fixed-point values, they are also interpreted according to section 2.1.5.

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

If `FRAMEBUFFER_SRGB` is enabled and the value of `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` for the framebuffer attachment corresponding to the destination buffer is `SRGB` (see section 6.1.3), the R, G, and B destination color values (after conversion from fixed-point to floating-point) are considered to be encoded for the sRGB color space and hence must be linearized prior to their use in blending. Each R, G, and B component is converted in the same fashion described for sRGB texture components in section 3.9.17.

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.

Mode	RGB Components	Alpha Component
`FUNC_ADD`	$R = R_s * S_r + R_d * D_r$ $G = G_s * S_g + G_d * D_g$ $B = B_s * S_b + B_d * D_b$	$A = A_s * S_a + A_d * D_a$
`FUNC_SUBTRACT`	$R = R_s * S_r - R_d * D_r$ $G = G_s * S_g - G_d * D_g$ $B = B_s * S_b - B_d * D_b$	$A = A_s * S_a - A_d * D_a$
`FUNC_REVERSE_SUBTRACT`	$R = R_d * D_r - R_s * S_r$ $G = G_d * D_g - G_s * S_g$ $B = B_d * D_b - B_s * S_b$	$A = A_d * D_a - A_s * S_a$
`MIN`	$R = \min(R_s, R_d)$ $G = \min(G_s, G_d)$ $B = \min(B_s, B_d)$	$A = \min(A_s, A_d)$
`MAX`	$R = \max(R_s, R_d)$ $G = \max(G_s, G_d)$ $B = \max(B_s, B_d)$	$A = \max(A_s, A_d)$

Table 4.1: RGB and alpha blend equations.

Blend Functions

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

`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

Function	RGB Blend Factors (S_r, S_g, S_b) or (D_r, D_g, D_b)	Alpha Blend Factor S_a or D_a
`ZERO`	$(0, 0, 0)$	0
`ONE`	$(1, 1, 1)$	1
`SRC_COLOR`	(R_s, G_s, B_s)	A_s
`ONE_MINUS_SRC_COLOR`	$(1, 1, 1) - (R_s, G_s, B_s)$	$1 - A_s$
`DST_COLOR`	(R_d, G_d, B_d)	A_d
`ONE_MINUS_DST_COLOR`	$(1, 1, 1) - (R_d, G_d, B_d)$	$1 - A_d$
`SRC_ALPHA`	(A_s, A_s, A_s)	A_s
`ONE_MINUS_SRC_ALPHA`	$(1, 1, 1) - (A_s, A_s, A_s)$	$1 - A_s$
`DST_ALPHA`	(A_d, A_d, A_d)	A_d
`ONE_MINUS_DST_ALPHA`	$(1, 1, 1) - (A_d, A_d, A_d)$	$1 - A_d$
`CONSTANT_COLOR`	(R_c, G_c, B_c)	A_c
`ONE_MINUS_CONSTANT_COLOR`	$(1, 1, 1) - (R_c, G_c, B_c)$	$1 - A_c$
`CONSTANT_ALPHA`	(A_c, A_c, A_c)	A_c
`ONE_MINUS_CONSTANT_ALPHA`	$(1, 1, 1) - (A_c, A_c, A_c)$	$1 - A_c$
`SRC_ALPHA_SATURATE`[1]	(f, f, f)[2]	1

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] $f = \min(A_s, 1 - A_d)$.

```
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 $(\mathrm{R}, \mathrm{G}, \mathrm{B}, \mathrm{A}) = (0, 0, 0, 0)$. Initially, blending is disabled for all draw buffers.

The value of the blend enable for draw buffer i can be queried by calling **IsEnabledi** 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.

4.1.9 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

$$c_s = \begin{cases} 0.0, & c_l \le 0 \\ 12.92 c_l, & 0 < c_l < 0.0031308 \\ 1.055 c_l^{0.41666} - 0.055, & 0.0031308 \le c_l < 1 \\ 1.0, & c_l \ge 1 \end{cases} \tag{4.1}$$

where c_l is the R, G, or B element and c_s is the result (effectively converted into an sRGB color space).

`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 using equation 2.4. 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 representable color value (for that particular color component) that is less than or equal to the incoming color component value, c, or the smallest representable color value that is greater than or equal to c. The selection may depend on the x_w and y_w coordinates of the pixel, as well as on the exact value of c. If one of the two values does not exist, then the selection defaults to the other value.

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

Argument value	Operation
`CLEAR`	0
`AND`	$s \wedge d$
`AND_REVERSE`	$s \wedge \neg d$
`COPY`	s
`AND_INVERTED`	$\neg s \wedge d$
`NOOP`	d
`XOR`	s xor d
`OR`	$s \vee d$
`NOR`	$\neg(s \vee d)$
`EQUIV`	$\neg(s \text{ xor } d)$
`INVERT`	$\neg d$
`OR_REVERSE`	$s \vee \neg d$
`COPY_INVERTED`	$\neg s$
`OR_INVERTED`	$\neg s \vee d$
`NAND`	$\neg(s \wedge d)$
`SET`	all 1's

Table 4.3: Arguments to **LogicOp** and their corresponding operations.

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

`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 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, dithering, and logical operations are performed for a pixel sample only if that sample's fragment coverage bit is a value of 1. If the corresponding coverage bit is 0, no operations are performed for that sample.

If `MULTISAMPLE` is disabled, and the value of `SAMPLE_BUFFERS` is one, the fragment may be treated exactly as described above, with optimization possible because the fragment coverage must be set to full coverage. Further optimization is allowed, however. An implementation may choose to identify a centermost sample, and to perform 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. If the framebuffer contains sRGB values, then it is recommended that the an average of sample values is computed in a linearized space, as for blending (see section 4.1.8). Otherwise, a simple average computed independently for each color component is recommended.

4.2 Whole Framebuffer Operations

The preceding sections described the operations that occur as individual fragments are sent to the framebuffer. This section describes operations that control or affect the whole framebuffer.

4.2.1 Selecting 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

`void` **DrawBuffer**(`enum` *buf*);

defines the set of color buffers to which fragment color zero is written. *buf* must be one of the values from tables 4.4 or 4.5. In addition, acceptable values for *buf* depend on whether the GL is using the default framebuffer (i.e., `DRAW_FRAMEBUFFER_BINDING` is zero), or a framebuffer object (i.e., `DRAW_FRAMEBUFFER_BINDING` is non-zero). In the initial state, the GL is bound to the default framebuffer. For more information about framebuffer objects, see section 4.4.

If the GL is bound to the default framebuffer, then *buf* must be one of the values listed in table 4.4, which summarizes the constants and the buffers they indicate. In this case, *buf* is a symbolic constant specifying zero, one, two, or four buffers for writing. These constants refer to the four potentially visible buffers (front left, front right, back left, and back right), and to the auxiliary buffers. Arguments other than `AUX`i that omit reference to `LEFT` or `RIGHT` refer to both left and right buffers. Arguments other than `AUX`i that omit reference to `FRONT` or `BACK` refer to both front and back buffers. `AUX`i enables drawing only to auxiliary buffer i. Each `AUX`i adheres to `AUX`i = `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_ATTACHMENT`i enables drawing only to the image attached to the framebuffer at `COLOR_ATTACHMENT`i. Each `COLOR_ATTACHMENT`i adheres to `COLOR_ATTACHMENT`i = `COLOR_ATTACHMENT0` + i. The intial value of `DRAW_BUFFER` for framebuffer objects is `COLOR_ATTACHMENT0`.

Symbolic Constant	Front Left	Front Right	Back Left	Back Right	Aux i
`NONE`					
`FRONT_LEFT`	●				
`FRONT_RIGHT`		●			
`BACK_LEFT`			●		
`BACK_RIGHT`				●	
`FRONT`	●	●			
`BACK`			●	●	
`LEFT`	●		●		
`RIGHT`		●		●	
`FRONT_AND_BACK`	●	●	●	●	
`AUX`i					●

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

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_ATTACHMENT`m 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

Symbolic Constant	Meaning
`NONE`	No buffer
`COLOR_ATTACHMENT`i (see caption)	Output fragment color to image attached at color attachment point i

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.

Symbolic Constant	Front Left	Front Right	Back Left	Back Right	Aux i
`NONE`					
`FRONT_LEFT`	●				
`FRONT_RIGHT`		●			
`BACK_LEFT`			●		
`BACK_RIGHT`				●	
`AUX`i					●

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

`NONE`.

The command

`void` **DrawBuffers**(`sizei` *n*, `const enum` **bufs*);

defines the draw buffers to which all fragment colors are written. *n* specifies the number of buffers in *bufs*. *bufs* is a pointer to an array of symbolic constants specifying the buffer to which each fragment color is written.

Each buffer listed in *bufs* must be one of the values from tables 4.5 or 4.6. Otherwise, an `INVALID_ENUM` error is generated. Further, acceptable values for the constants in *bufs* depend on whether the GL is using the default framebuffer (i.e., `DRAW_FRAMEBUFFER_BINDING` is zero), or a framebuffer object (i.e., `DRAW_FRAMEBUFFER_BINDING` is non-zero). For more information about framebuffer objects, see section 4.4.

If the GL is bound to the default framebuffer, then each of the constants must be one of the values listed in table 4.6.

If the GL is bound to an framebuffer object, then each of the constants must be one of the values listed in table 4.5.

In both cases, the draw buffers being defined correspond in order to the respective fragment colors. The draw buffer for fragment colors beyond *n* is set to `NONE`.

The maximum number of draw buffers is implementation-dependent. The number of draw buffers supported can be queried by calling **GetIntegerv** with the symbolic constant `MAX_DRAW_BUFFERS`. An `INVALID_VALUE` error is generated if *n* is greater than `MAX_DRAW_BUFFERS`.

Except for `NONE`, a buffer may not appear more then once in the array pointed 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-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_ENUM`. This restriction is because these constants may themselves refer to multiple buffers, as shown in table 4.4.

If the GL is bound to the default framebuffer and **DrawBuffers** is supplied with a constant (other than `NONE`) that does not indicate any of the color buffers allocated to the GL context by the window system, the error `INVALID_OPERATION` will be generated.

If the GL is bound to a framebuffer object and **DrawBuffers** is supplied with a constant from table 4.6, or `COLOR_ATTACHMENT`m where m is greater than or equal to the value of `MAX_COLOR_ATTACHMENTS`, then the error `INVALID_OPERATION` results.

Indicating a buffer or buffers using **DrawBuffer** or **DrawBuffers** causes subsequent pixel color value writes to affect the indicated buffers.

Specifying `NONE` as the draw buffer for a fragment color will inhibit that fragment color from being written to any buffer.

Monoscopic contexts include only left buffers, while stereoscopic contexts include both left and right buffers. Likewise, single buffered contexts include only front buffers, while double buffered contexts include both front and back buffers. The type of context is selected at GL initialization.

The state required to handle color buffer selection for each framebuffer is an integer for each supported fragment color. For the default framebuffer, in the initial state the draw buffer for fragment color zero is `BACK` if there is a back buffer; `FRONT` if there is no back buffer; and `NONE` if no default framebuffer is associated with the context. For framebuffer objects, in the initial state the draw buffer for fragment color zero is `COLOR_ATTACHMENT0`. For both the default framebuffer and framebuffer objects, the initial state of draw buffers for fragment colors other then zero is `NONE`.

The value of the draw buffer selected for fragment color i can be queried by calling **GetIntegerv** with the symbolic constant `DRAW_BUFFER`i. `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*);
> `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, **ColorMask** and **ColorMaski** are used to mask the writing of R, G, B and A values to the draw buffer or buffers. **ColorMaski** sets the mask for a particular draw buffer. The mask for `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 `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 `DRAW_BUFFER`i. If `DRAW_BUFFER`i is one of `FRONT`, `BACK`, `LEFT`, `RIGHT`, or `FRONT_AND_BACK` (specifying multiple color buffers) then the mask is applied to all of the buffers.

ColorMask sets the mask for all draw buffers to the same values as specified by *r*, *g*, *b*, and *a*.

An `INVALID_VALUE` error is generated if *index* is greater than the value of `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 **GetBooleani_v** with *target* `COLOR_WRITEMASK` and *index* i. The value of the color writemask for draw buffer zero may also be queried by calling **GetBooleanv** with *value* `COLOR_WRITEMASK`.

The depth buffer can be enabled or disabled for writing z_w values using void **DepthMask**(boolean *mask*);

If *mask* is non-zero, the depth buffer is enabled for writing; otherwise, it is disabled. In the initial state, the depth buffer is enabled for writing.

The commands

void **StencilMask**(uint *mask*);
void **StencilMaskSeparate**(enum *face*, uint *mask*);

control the writing of particular bits into the stencil planes.

The least significant s bits of *mask*, where s is the number of bits in the stencil buffer, specify an integer mask. Where a 1 appears in this mask, the corresponding bit in the stencil buffer is written; where a 0 appears, the bit is not written. The *face* parameter of **StencilMaskSeparate** can be `FRONT`, `BACK`, or `FRONT_AND_BACK` and indicates whether the front or back stencil mask state is affected. **StencilMask** sets both front and back stencil mask state to identical values.

Fragments generated by front-facing primitives use the front mask and fragments generated by back-facing primitives use the back mask (see section 4.1.5). The clear operation always uses the front stencil write mask when clearing the stencil buffer.

The state required for the various masking operations is an integer for color indices, two integers for the front and back stencil values, and a bit for depth values. A set of four bits is also required indicating which color components of an RGBA value should be written. In the initial state, the integer masks are all ones, as are the bits controlling depth value and RGBA component writing.

Fine Control of Multisample Buffer Updates

When the value of `SAMPLE_BUFFERS` is one, **ColorMask**, **DepthMask**, and **StencilMask** or **StencilMaskSeparate** control the modification of values in the multisample buffer. The color mask has no effect on modifications to the color buffers. If the color mask is entirely disabled, the color sample values must still be combined (as described above) and the result used to replace the color values of the buffers enabled by **DrawBuffer**.

4.2.3 Clearing the Buffers

The GL provides a means for setting portions of every pixel in a particular buffer to the same value. The argument to

void **Clear**(bitfield *buf*);

is zero or the bitwise OR of one or more values indicating which buffers are to be cleared. The values are `COLOR_BUFFER_BIT`, `ACCUM_BUFFER_BIT`, `DEPTH_BUFFER_BIT`, and `STENCIL_BUFFER_BIT`, indicating the buffers currently enabled for color writing, the accumulation buffer, the depth buffer, and the stencil buffer (see below), respectively. The value to which each buffer is cleared depends on the setting of the clear value for that buffer. If *buf* is zero, no buffers are cleared. If *buf* contains any bits other than `COLOR_BUFFER_BIT`, `ACCUM_BUFFER_BIT`, `DEPTH_BUFFER_BIT`, or `STENCIL_BUFFER_BIT`, then the error `INVALID_VALUE` is generated.

`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 2.16.1. 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. Unsigned normalized fixed-point and signed normalized fixed-point RGBA color buffers are cleared to color values derived by clamping each component of the clear color to the range $[0, 1]$ or $[-1, 1]$ respectively, then converting to fixed-point using equations 2.4 or 2.6, respectively. The result of clearing integer color buffers is undefined.

The state required for clearing is a clear value for each of the color buffer, the accumulation buffer, the depth buffer, and the stencil buffer. Initially, the RGBA color clear value is $(0, 0, 0, 0)$, the accumulation buffer clear value is $(0, 0, 0, 0)$, the clear color index is 0, the depth buffer clear value is 1.0, and the stencil buffer clear index is 0.

Individual buffers of the currently bound draw framebuffer may be cleared with the command

`void` **ClearBuffer{if ui}v**(`enum` *buffer*, `int` *drawbuffer*,
`const T` **value*);

where *buffer* and *drawbuffer* identify a buffer to clear, and *value* specifies the value or values to clear it to.

If *buffer* is `COLOR`, a particular draw buffer `DRAW_BUFFER`i is specified by passing *i* as the parameter *drawbuffer*, and *value* points to a four-element vector specifying the R, G, B, and A color to clear that draw buffer to. If the draw buffer is one of `FRONT`, `BACK`, `LEFT`, `RIGHT`, or `FRONT_AND_BACK`, identifying multiple buffers, each selected buffer is cleared to the same value. The **ClearBufferfv**, **ClearBufferiv**, and **ClearBufferuiv** commands should be used to clear fixed- and floating-point, signed integer, and unsigned integer color buffers respectively. Clamping and conversion for fixed-point color buffers are performed in the same fashion as **ClearColor**.

If *buffer* is `DEPTH`, *drawbuffer* must be zero, and *value* points to the single depth value to clear the depth buffer to. Clamping and type conversion for fixed-point depth buffers are performed in the same fashion as **ClearDepth**. Only **ClearBufferfv** should be used to clear depth buffers.

If *buffer* is `STENCIL`, *drawbuffer* must be zero, and *value* points to the single stencil value to clear the stencil buffer to. Masking 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*, `int` *drawbuffer*,
`float` *depth*, `int` *stencil*);

clears both depth and stencil buffers of the currently bound draw framebuffer. *buffer* must be `DEPTH_STENCIL` and *drawbuffer* must be zero. *depth* and *stencil* are the values to clear the depth and stencil buffers to, respectively. Clamping and type conversion of *depth* for fixed-point depth buffers is performed in the same fashion as **ClearDepth**. Masking of *stencil* for stencil buffers is performed in the same fashion as **ClearStencil**. **ClearBufferfi** is equivalent to clearing the depth and stencil buffers separately, but may be faster when a buffer of internal format `DEPTH_STENCIL` is being cleared.

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

When **ClearBuffer** is called, the same per-fragment and masking operations defined for **Clear** are applied.

Errors

ClearBuffer{**if ui**}**v** generates an `INVALID_ENUM` error if *buffer* is not `COLOR`, `DEPTH`, or `STENCIL`. **ClearBufferfi** generates an `INVALID_ENUM` error if *buffer* is not `DEPTH_STENCIL`.

ClearBuffer generates an `INVALID_VALUE` error if *buffer* is `COLOR` and *drawbuffer* is less than zero, or greater than the value of `MAX_DRAW_BUFFERS` minus one; or if *buffer* is `DEPTH`, `STENCIL`, or `DEPTH_STENCIL` and *drawbuffer* is not zero.

ClearBuffer generates an `INVALID_OPERATION` error if *buffer* is `COLOR` and the GL is in color index mode.

Clearing the Multisample Buffer

The color samples of the multisample buffer are cleared when one or more color buffers are cleared, as specified by the **Clear** mask bit `COLOR_BUFFER_BIT` and the **DrawBuffer** mode. If the **DrawBuffer** mode is `NONE`, the color samples of the multisample buffer cannot be cleared using **Clear**.

If the **Clear** mask bits `DEPTH_BUFFER_BIT` or `STENCIL_BUFFER_BIT` are set, then the corresponding depth or stencil samples, respectively, are cleared.

The **ClearBuffer** commands also clear color, depth, or stencil samples of multisample buffers corresponding to the specified buffer.

4.2.4 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 to every affected pixel, so we describe the effect of each operation on an individual pixel. Accumulation buffer values are taken to be signed values in the range $[-1, 1]$. Using `ACCUM` obtains R, G, B, and A components from the buffer currently selected for reading (section 4.3.2). If the color buffer is fixed-point, each component is considered as an unsigned normalized value in the range $[0, 1]$ and is converted to floating-point using equation 2.1. Each result is then multiplied by *value*. The results of this multiplication are then added to the corresponding color component currently in the accumulation buffer, and the resulting color value replaces the current accumulation buffer color value.

The `LOAD` operation has the same effect as `ACCUM`, but the computed values replace the corresponding accumulation buffer components rather than being added to them.

The `RETURN` operation takes each color value from the accumulation buffer, multiplies each of the R, G, B, and A components by *value*. If fragment color clamping is enabled, the results are then clamped to the range $[0, 1]$. The resulting color value is placed in the buffers currently enabled for color writing as if it were a fragment produced from rasterization, except that the only per-fragment operations that are applied (if enabled) are the pixel ownership test, the scissor test (section 4.1.2), and dithering (section 4.1.10). Color masking (section 4.2.2) is also applied.

The `MULT` operation multiplies each R, G, B, and A in the accumulation buffer by *value* and then returns the scaled color components to their corresponding accumulation buffer locations. `ADD` is the same as `MULT` except that *value* is added to each of the color components.

The color components operated on by **Accum** must be clamped only if the operation is `RETURN`. In this case, a value sent to the enabled color buffers is first clamped to $[0,1]$. Otherwise, results are undefined if the result of an operation on a color component is out of the range $[-1,1]$.

If there is no accumulation buffer; if the `DRAW_FRAMEBUFFER` and `READ_FRAMEBUFFER` bindings (see section 4.4.4) do not refer to the same object; or if the GL is in color index mode, **Accum** generates the error `INVALID_OPERATION`.

No state (beyond the accumulation buffer itself) is required for accumulation buffering.

4.3 Drawing, Reading, and Copying Pixels

Pixels may be written to the framebuffer using **DrawPixels**. Pixels may be read from the framebuffer using **ReadPixels**. **CopyPixels** and **BlitFramebuffer** can be used to copy a block of pixels from 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 3.7.5, 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 **DepthFunc** 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 pixel results into client memory pointer parameters. However, if a non-zero buffer object is bound as the current

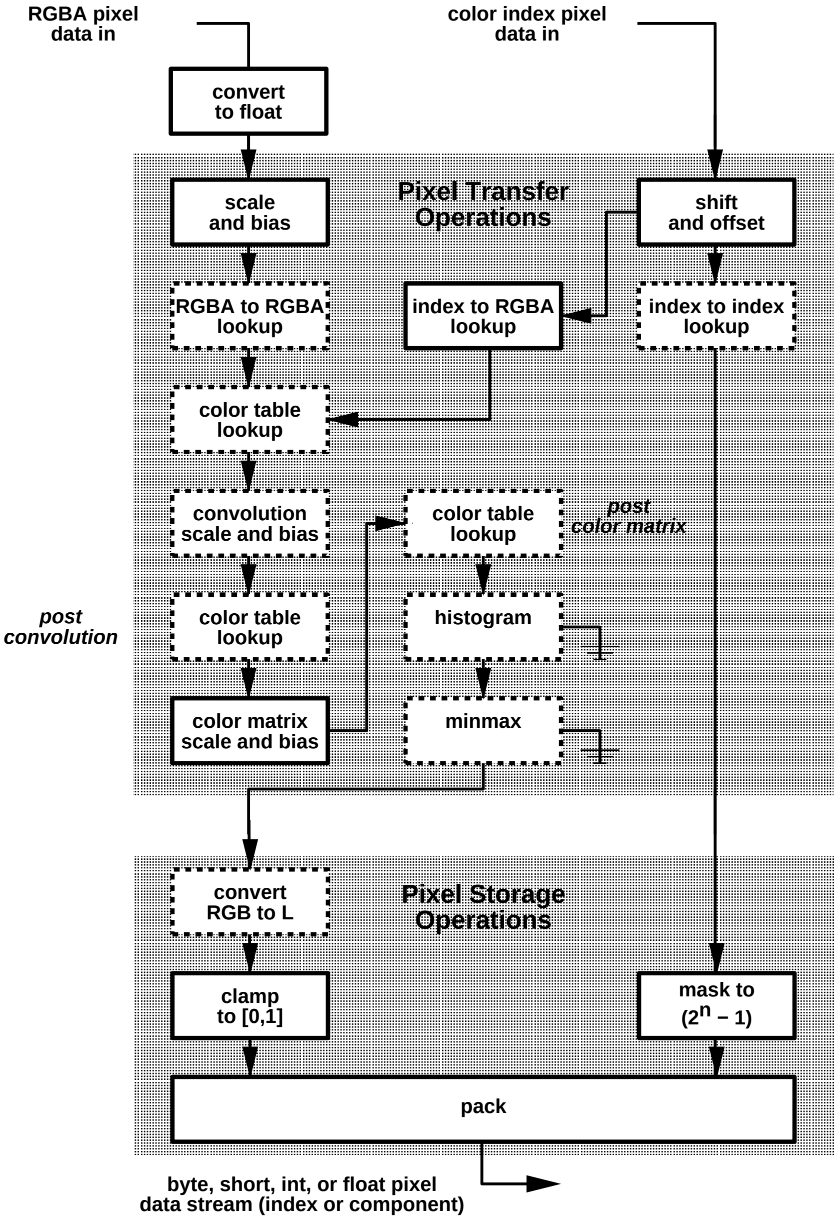


Figure 4.2. Operation of **ReadPixels**. Operations in dashed boxes may be enabled or disabled, except in the case of ”convert RGB to L”, which is only applied when reading color data in luminosity formats. RGBA and color index pixel paths are shown; depth and stencil pixel paths are not shown.

Parameter Name	Type	Initial Value	Valid Range
`PACK_SWAP_BYTES`	boolean	`FALSE`	`TRUE`/`FALSE`
`PACK_LSB_FIRST`	boolean	`FALSE`	`TRUE`/`FALSE`
`PACK_ROW_LENGTH`	integer	0	$[0, \infty)$
`PACK_SKIP_ROWS`	integer	0	$[0, \infty)$
`PACK_SKIP_PIXELS`	integer	0	$[0, \infty)$
`PACK_ALIGNMENT`	integer	4	1,2,4,8
`PACK_IMAGE_HEIGHT`	integer	0	$[0, \infty)$
`PACK_SKIP_IMAGES`	integer	0	$[0, \infty)$

Table 4.7: **PixelStore** parameters pertaining to **ReadPixels**, **GetColorTable**, **GetConvolutionFilter**, **GetSeparableFilter**, **GetHistogram**, **GetMinmax**, **GetPolygonStipple**, and **GetTexImage**.

pixel pack buffer, then the pointer parameter is treated as an offset into the designated buffer object.

Pixels are read using

`void` **ReadPixels**(`int` *x*, `int` *y*, `sizei` *width*, `sizei` *height*,
`enum` *format*, `enum` *type*, `void` **data*);

The arguments after *x* and *y* to **ReadPixels** are described in section 3.7.4. The pixel storage modes that apply to **ReadPixels** and other commands that query images (see section 6.1) are summarized in table 4.7.

ReadPixels generates an `INVALID_OPERATION` error if `READ_FRAMEBUFFER_BINDING` (see section 4.4) is non-zero, the read framebuffer is framebuffer complete, and the value of `SAMPLE_BUFFERS` for the read framebuffer is greater than zero.

Obtaining Pixels from the Framebuffer

If the *format* is `DEPTH_COMPONENT`, then values are obtained from the depth buffer. If there is no depth buffer, the error `INVALID_OPERATION` occurs.

If there is a multisample buffer (the value of `SAMPLE_BUFFERS` is one), then values are obtained from the depth samples in this buffer. It is recommended that the depth value of the centermost sample be used, though implementations may choose any function of the depth sample values at each pixel.

If the *format* is `DEPTH_STENCIL`, then values are taken from both the depth buffer and the stencil buffer. If there is no depth buffer or if there is no stencil buffer, then the error `INVALID_OPERATION` occurs. If the *type* parameter is not `UNSIGNED_INT_24_8` or `FLOAT_32_UNSIGNED_INT_24_8_REV`, then the error `INVALID_ENUM` occurs.

If there is a multisample buffer, then values are obtained from the depth and stencil samples in this buffer. It is recommended that the depth and stencil values of the centermost sample be used, though implementations may choose any function of the depth and stencil sample values at each pixel.

If the *format* is `STENCIL_INDEX`, then values are taken from the stencil buffer; again, if there is no stencil buffer, the error `INVALID_OPERATION` occurs.

If there is a multisample buffer, then values are obtained from the stencil samples in this buffer. It is recommended that the stencil value of the centermost sample be used, though implementations may choose any function of the stencil sample values at each pixel.

For all other formats, the *read buffer* from which values are obtained is one of the color buffers; the selection of color buffer is controlled with **ReadBuffer**.

The command

`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_ATTACHMENT`i enables reading from the image attached to the framebuffer at `COLOR_ATTACHMENT`i. 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 ith pixel in the jth row. If any of these pixels lies outside of the window allocated to the current GL context, or outside of the image attached to the currently bound framebuffer object, then the values obtained for those pixels are undefined. When `READ_FRAMEBUFFER_BINDING` is zero, values are also undefined for individual pixels that are not owned by the current context. Otherwise, **ReadPixels** obtains values from the selected buffer, regardless of how those values were placed there.

If the GL is in RGBA mode, and *format* is one of `LUMINANCE`, `LUMINANCE_ALPHA`, `RED`, `GREEN`, `BLUE`, `ALPHA`, `RG`, `RGB`, `RGBA`, `BGR`, or `BGRA`, then red, green, blue, and alpha values are obtained from the selected buffer at each pixel location. If the framebuffer does not support alpha values then the A that is obtained is 1.0. If *format* is `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 3.16). 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 signed or unsigned normalized fixed-point color buffer, each element is converted to floating-point using equations 2.3 or 2.1, respectively. For an integer or floating-point color buffer, the elements are unmodified.

Conversion of Depth values

This step applies only if *format* is `DEPTH_COMPONENT` or `DEPTH_STENCIL` and the depth buffer uses a fixed-point representation. An element is taken to be a fixed-point value in $[0, 1]$ with m bits, where m is the number of bits in the depth buffer (see section 2.16.1). No conversion is necessary if the depth buffer uses a floating-point representation.

Pixel Transfer Operations

This step is actually the sequence of steps that was described separately in section 3.7.6. 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.

Read color clamping is controlled by calling **ClampColor** (see section 3.7.5) with *target* set to `CLAMP_READ_COLOR`. If *clamp* is `TRUE`, read color clamping is enabled; if *clamp* is `FALSE`, read color clamping is disabled. If *clamp* is `FIXED_ONLY`, read color clamping is enabled if the selected read color buffer has fixed-point components.

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

type Parameter	Index Mask
`UNSIGNED_BYTE`	$2^8 - 1$
`BITMAP`	1
`BYTE`	$2^7 - 1$
`UNSIGNED_SHORT`	$2^{16} - 1$
`SHORT`	$2^{15} - 1$
`UNSIGNED_INT`	$2^{32} - 1$
`INT`	$2^{31} - 1$
`UNSIGNED_INT_24_8`	$2^8 - 1$
`FLOAT_32_UNSIGNED_INT_24_8_REV`	$2^8 - 1$

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

In the special case of calling **ReadPixels** with *type* of `UNSIGNED_INT_10F_11F_11F_REV` and *format* of `RGB`, conversion is performed as follows: the returned data are packed into a series of `uint` values. The red, green, and blue components are converted to unsigned 11-bit floating-point, unsigned 11-bit floating-point, and unsigned 10-bit floating point as described in sections 2.1.3 and 2.1.4. The resulting red 11 bits, green 11 bits, and blue 10 bits are then packed as the 1st, 2nd, and 3rd components of the `UNSIGNED_INT_10F_11F_11F_REV` format as shown in table 3.11.

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, $green_s$, $blue_s$, 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

type Parameter	GL Data Type	Component Conversion Formula
`UNSIGNED_BYTE`	`ubyte`	$c = (2^8 - 1)f$
`BYTE`	`byte`	$c = \frac{(2^8-1)f-1}{2}$
`UNSIGNED_SHORT`	`ushort`	$c = (2^{16} - 1)f$
`SHORT`	`short`	$c = \frac{(2^{16}-1)f-1}{2}$
`UNSIGNED_INT`	`uint`	$c = (2^{32} - 1)f$
`INT`	`int`	$c = \frac{(2^{32}-1)f-1}{2}$
`HALF_FLOAT`	`half`	$c = f$
`FLOAT`	`float`	$c = f$
`UNSIGNED_BYTE_3_3_2`	`ubyte`	$c = (2^N - 1)f$
`UNSIGNED_BYTE_2_3_3_REV`	`ubyte`	$c = (2^N - 1)f$
`UNSIGNED_SHORT_5_6_5`	`ushort`	$c = (2^N - 1)f$
`UNSIGNED_SHORT_5_6_5_REV`	`ushort`	$c = (2^N - 1)f$
`UNSIGNED_SHORT_4_4_4_4`	`ushort`	$c = (2^N - 1)f$
`UNSIGNED_SHORT_4_4_4_4_REV`	`ushort`	$c = (2^N - 1)f$
`UNSIGNED_SHORT_5_5_5_1`	`ushort`	$c = (2^N - 1)f$
`UNSIGNED_SHORT_1_5_5_5_REV`	`ushort`	$c = (2^N - 1)f$
`UNSIGNED_INT_8_8_8_8`	`uint`	$c = (2^N - 1)f$
`UNSIGNED_INT_8_8_8_8_REV`	`uint`	$c = (2^N - 1)f$
`UNSIGNED_INT_10_10_10_2`	`uint`	$c = (2^N - 1)f$
`UNSIGNED_INT_2_10_10_10_REV`	`uint`	$c = (2^N - 1)f$
`UNSIGNED_INT_24_8`	`uint`	$c = (2^N - 1)f$
`UNSIGNED_INT_10F_11F_11F_REV`	`uint`	Special
`UNSIGNED_INT_5_9_9_9_REV`	`uint`	Special
`FLOAT_32_UNSIGNED_INT_24_8_REV`	`float`	$c = f$ (depth only)

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.

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 when transferring pixel rectangles to the GL. That is, the ith group of the jth row (corresponding to the ith pixel in the jth row) is placed in memory just where the ith group of the jth row would be taken from when transferring pixels. 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 `LUMINANCE`, `RED`, `GREEN`, `BLUE`, or `ALPHA`, only the corresponding single element is written. Likewise if the *format* is `LUMINANCE_ALPHA`, `RG`, `RGB`, or `BGR`, only the corresponding two or three elements are written. Otherwise all the elements of each group are written.

4.3.3 Copying Pixels

The command

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 3.7.6, 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.

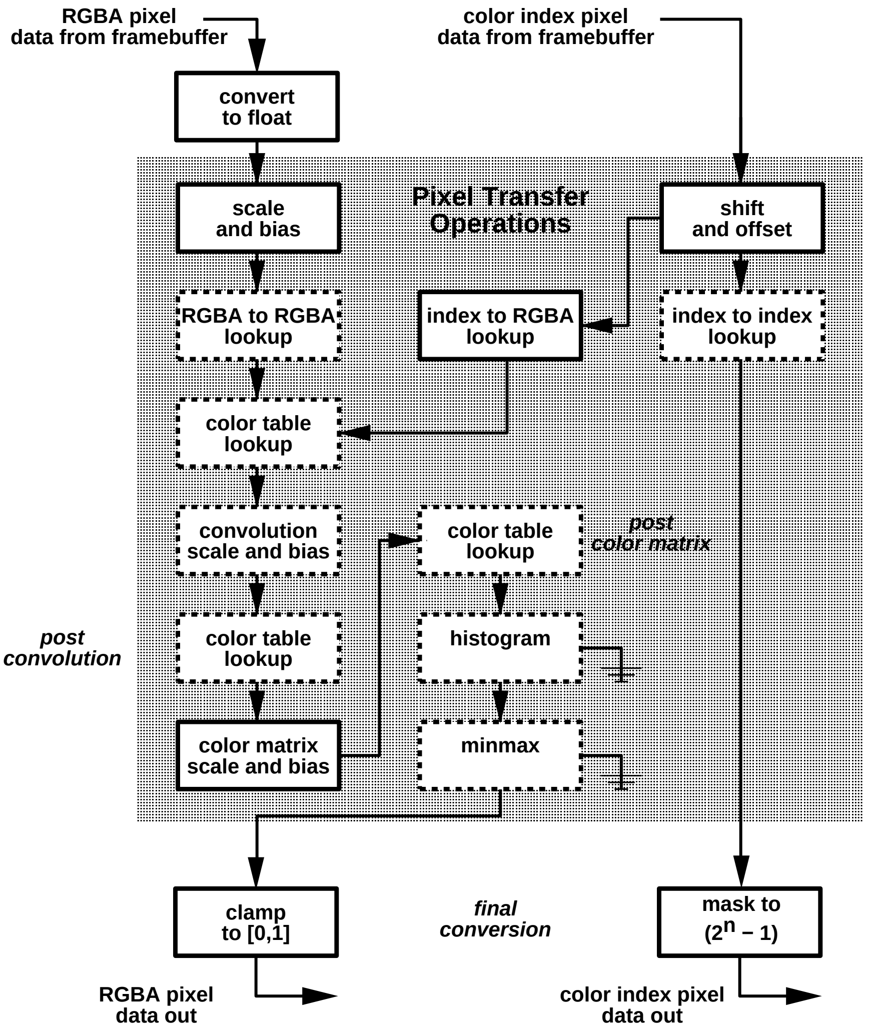


Figure 4.3. Operation of **CopyPixels**. Operations in dashed boxes may be enabled or disabled. Index-to-RGBA lookup is currently never performed. RGBA and color index pixel paths are shown; depth and stencil pixel paths are not shown.

`DEPTH_BITS`	`STENCIL_BITS`	*format*
zero	zero	`DEPTH_STENCIL`
zero	non-zero	`DEPTH_COMPONENT`
non-zero	zero	`STENCIL_INDEX`
non-zero	non-zero	`DEPTH_STENCIL`

Table 4.10: Effective **ReadPixels** format for `DEPTH_STENCIL` **CopyPixels** operation.

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.

Blitting Pixel Rectangles

The command

`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.

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 ($X1 < X0$ or $Y1 < Y0$), the image is reversed in the corresponding direction. If both the source and destination rectangles specify a negative width or height for the same direction, no reversal is performed. If a linear filter is selected and the rules of `LINEAR` sampling would require sampling outside the bounds of a source buffer, it is as though `CLAMP_TO_EDGE` texture sampling were being performed. If a linear filter is selected and sampling would be required outside the bounds of the specified source region, but within the bounds of a source buffer, the implementation may choose to clamp while sampling or not.

If the source and destination buffers are identical, and the source and destination rectangles overlap, the result of the blit operation is undefined.

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

If the read framebuffer is layered (see section 4.4.7), pixel values are read from layer zero. If the draw framebuffer is layered, pixel values are written to layer zero. If both read and draw framebuffers are layered, the blit operation is still performed only on layer zero.

If a buffer is specified in *mask* and does not exist in both the read and draw framebuffers, the corresponding bit is silently ignored.

If the color formats of the read and draw buffers do not match, and *mask* includes `COLOR_BUFFER_BIT`, pixel groups are converted to match the destination format as in **CopyPixels**. However, no pixel transfer operations are applied, and colors are clamped only if all draw color buffers have fixed-point components, as if `CLAMP_FRAGMENT_COLOR` were set to `FIXED_ONLY`. Format conversion is not supported for all data types, and an `INVALID_OPERATION` error is generated under any of the following conditions:

- The read buffer contains fixed-point or floating-point values and any draw buffer contains neither fixed-point nor floating-point values.
- The read buffer contains unsigned integer values and any draw buffer does not contain unsigned integer values.
- The read buffer contains signed integer values and any draw buffer does not contain signed integer values.

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

Calling **BlitFramebuffer** will result in an `INVALID_OPERATION` error if *mask* includes `DEPTH_BUFFER_BIT` or `STENCIL_BUFFER_BIT`, and the source and destination depth and stencil buffer formats do not match.

Calling **BlitFramebuffer** will result in an `INVALID_OPERATION` error if *filter* is `LINEAR` and read buffer contains integer data.

If `SAMPLE_BUFFERS` for the read framebuffer is greater than zero and `SAMPLE_BUFFERS` for the draw framebuffer is zero, the samples corresponding to each pixel location in the source are converted to a single sample before being written to the destination.

If `SAMPLE_BUFFERS` for the read framebuffer is zero and `SAMPLE_BUFFERS` for the draw framebuffer is greater than zero, the value of the source sample is replicated in each of the destination samples.

If `SAMPLE_BUFFERS` for either the read framebuffer or draw framebuffer is greater than zero, no copy is performed and an `INVALID_OPERATION` error is generated if the dimensions of the source and destination rectangles provided to **BlitFramebuffer** are not identical, if the formats of the read and draw framebuffers are not identical, or if the values of `SAMPLES` for the read and draw buffers are not identical.

If `SAMPLE_BUFFERS` for both the read and draw framebuffers are greater than zero, and the values of `SAMPLES` for the read and draw framebuffers are identical, the samples are copied without modification from the read framebuffer to the draw framebuffer. Otherwise, no copy is performed and an `INVALID_OPERATION` error is generated. Note that the samples in the draw buffer are not guaranteed to be at the same sample location as the read buffer, so rendering using this newly created buffer can potentially have geometry cracks or incorrect antialiasing. This may occur if the sizes of the framebuffers do not match, if the formats differ, or if the source and destination rectangles are not defined with the same $(X0, Y0)$ 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 (section 2.24), an integer indicating the current setting of **ReadBuffer**, and a three-valued integer controlling clamping during final conversion. For the default framebuffer, in the initial state the read buffer is `BACK` if there is a back buffer; `FRONT` if there is no back buffer; and `NONE` if no default framebuffer is associated with the context. The initial value of read color clamping is `FIXED_ONLY`. State set with **PixelStore** is GL client state.

4.4 Framebuffer Objects

As described in chapter 1 and section 2.1, the GL renders into (and reads values from) a framebuffer. GL defines two classes of framebuffers: window system-provided and application-created.

Initially, the GL uses the default framebuffer. The storage, dimensions, allocation, and format of the images attached to this framebuffer are managed entirely by the window system. Consequently, the state of the default framebuffer, including its images, can not be changed by the GL, nor can the default framebuffer be deleted by the GL.

The routines described in the following sections, however, can be used to create, destroy, and modify the state and attachments of framebuffer objects.

Framebuffer objects encapsulate the state of a framebuffer in a similar manner to the way texture objects encapsulate the state of a texture. In particular, a framebuffer object encapsulates state necessary to describe a collection of color, depth, and stencil logical buffers (other types of buffers are not allowed). For each logical buffer, a framebuffer-attachable image can be attached to the framebuffer to store the rendered output for that logical buffer. Examples of framebuffer-attachable images include texture images and renderbuffer images. Renderbuffers are described further in section 4.4.2

By allowing the images of a renderbuffer to be attached to a framebuffer, the GL provides a mechanism to support *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

`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 6.30, as well as one set of the state values listed in table 6.31 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 6.60 and 6.33 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 objectss 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_ATTACHMENT`n.
- 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 number of layers of each attachment are not all identical, rendering will be limited to the smallest number of layers of any attachment.
- If the attachment sizes are not all identical, the values of pixels outside the common intersection area after rendering are undefined.

Framebuffer objects are deleted by calling

`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

`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.

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`.

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 6.31

There are several types of framebuffer-attachable images:

- The image of a renderbuffer object, which is always two-dimensional.
- A single level of a one-dimensional texture, which is treated as a two-dimensional image with a height of one.
- A single level of a two-dimensional or rectangle texture.
- A single face of a cube map texture level, which is treated as a two-dimensional image.
- A single layer of a one-or two-dimensional array texture or three-dimensional texture, which is treated as a two-dimensional image.

Additionally, an entire level of a three-dimensional texture, cube map texture, or one-or two-dimensional array texture can be attached to an attachment point. Such attachments are treated as an array of two-dimensional images, arranged in layers, and the corresponding attachment point is considered to be *layered* (also see section 4.4.7).

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

`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 6.33. Any previous binding to *target* is broken.

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

While a renderbuffer object is bound, GL operations on the target to which it is bound affect the bound renderbuffer object, and queries of the target to which a renderbuffer object is bound return state from the bound object.

The name zero is reserved. A renderbuffer object cannot be created with the name zero. If *renderbuffer* is zero, then any previous binding to *target* is broken and the *target* binding is restored to the initial state.

In the initial state, the reserved name zero is bound to `RENDERBUFFER`. There is no renderbuffer object corresponding to the name zero, so client attempts to modify or query renderbuffer state for the target `RENDERBUFFER` while zero is bound will generate GL errors, as described in section 6.1.3.

The current `RENDERBUFFER` binding can be determined by calling **GetIntegerv** with the symbolic constant `RENDERBUFFER_BINDING`.

BindRenderbuffer fails and an `INVALID_OPERATION` error is generated if *renderbuffer* is not zero or a name returned from a previous call to **GenRenderbuffers**, or if such a name has since been deleted with **DeleteRenderbuffers**.

Renderbuffer objects are deleted by calling

`void` **DeleteRenderbuffers**(`sizei` *n*, `const`
`uint` **renderbuffers*);

where *renderbuffers* contains *n* names of renderbuffer objects to be deleted. After a renderbuffer object is deleted, it has no contents, and its name is again unused. If a renderbuffer that is currently bound to `RENDERBUFFER` is deleted, it is as though **BindRenderbuffer** had been executed with the *target* `RENDERBUFFER` and *name* of zero. Additionally, special care must be taken when deleting a renderbuffer if the image of the renderbuffer is attached to a framebuffer object (see section 4.4.2). Unused names in *renderbuffers* are silently ignored, as is the value zero.

The command

`void` **GenRenderbuffers**(`sizei` *n*, `uint` **renderbuffers*);

returns *n* previously unused renderbuffer object names in *renderbuffers*. These names are marked as used, for the purposes of **GenRenderbuffers** only, but they acquire renderbuffer state only when they are first bound.

The command

`void` **RenderbufferStorageMultisample**(`enum` *target*,
`sizei` *samples*, `enum` *internalformat*, `sizei` *width*,
`sizei` *height*);

Sized Internal Format	Base Internal Format	S bits
`STENCIL_INDEX1`	`STENCIL_INDEX`	1
`STENCIL_INDEX4`	`STENCIL_INDEX`	4
`STENCIL_INDEX8`	`STENCIL_INDEX`	8
`STENCIL_INDEX16`	`STENCIL_INDEX`	16

Table 4.11: Correspondence of sized internal formats to base internal formats for formats that can be used only with renderbuffers.

establishes the data storage, format, dimensions, and number of samples of a renderbuffer object's image. *target* must be `RENDERBUFFER`. *internalformat* must be color-renderable, depth-renderable, or stencil-renderable (as defined in section 4.4.4). *width* and *height* are the dimensions in pixels of the renderbuffer. If either *width* or *height* is greater than the value of `MAX_RENDERBUFFER_SIZE`, or if *samples* is greater than the value of `MAX_SAMPLES`, then the error `INVALID_VALUE` is generated. If *internalformat* is a signed or unsigned integer format and *samples* is greater than the value of `MAX_INTEGER_SAMPLES`, then the error `INVALID_OPERATION` is generated (see "Required Renderbuffer Formats" below). If the GL is unable to create a data store of the requested size, the error `OUT_OF_MEMORY` is generated.

Upon success, **RenderbufferStorageMultisample** deletes any existing data store for the renderbuffer image and the contents of the data store after calling **RenderbufferStorageMultisample** are undefined. `RENDERBUFFER_WIDTH` is set to *width*, `RENDERBUFFER_HEIGHT` is set to *height*, and `RENDERBUFFER_INTERNAL_FORMAT` is set to *internalformat*.

If *samples* is zero, then `RENDERBUFFER_SAMPLES` is set to zero. Otherwise *samples* represents a request for a desired minimum number of samples. Since different implementations may support different sample counts for multisampled rendering, the actual number of samples allocated for the renderbuffer image is 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

`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 the color formats labelled "texture-only". Requesting one of these internal formats for a renderbuffer will allocate at least the internal component sizes and exactly the component types shown for that format in tables 3.17- 3.19.

Implementations must support creation of renderbuffers in these required formats with up to the value of `MAX_SAMPLES` multisamples, with the exception that the signed and unsigned integer formats are required only to support creation of renderbuffers with up to the value of `MAX_INTEGER_SAMPLES` multisamples, which must be at least one.

Attaching Renderbuffer Images to a Framebuffer

A renderbuffer can be attached as one of the logical buffers of the currently bound framebuffer object by calling

`void` **FramebufferRenderbuffer**(`enum` *target*,
`enum` *attachment*, `enum` *renderbuffertarget*,
`uint` *renderbuffer*);

target must be `DRAW_FRAMEBUFFER`, `READ_FRAMEBUFFER`, or `FRAMEBUFFER`. `FRAMEBUFFER` is equivalent to `DRAW_FRAMEBUFFER`. An `INVALID_OPERATION` error is generated if the value of the corresponding binding is zero. *attachment* should be set to one of the attachment points of the framebuffer listed in table 4.12.

renderbuffertarget must be `RENDERBUFFER` and *renderbuffer* should be set to the name of the renderbuffer object to be attached to the framebuffer. *renderbuffer* must be either zero or the name of an existing renderbuffer object of type *renderbuffertarget*, otherwise an `INVALID_OPERATION` error is generated. If *renderbuffer* is zero, then the value of *renderbuffertarget* is ignored.

If *renderbuffer* is not zero and if **FramebufferRenderbuffer** is successful, then the renderbuffer named *renderbuffer* will be used as the logical buffer identified by *attachment* of the framebuffer currently bound to *target*. The value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` for the specified attachment point is set to `RENDERBUFFER` and the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` is set to *renderbuffer*. All other state values of the attachment point specified by *attachment* are set to their default values listed in table 6.31. No change is made to the state of the renderbuffer object and any previous attachment to the *attachment* logical buffer of the framebuffer object bound to framebuffer *target* is broken. If the attachment is not successful, then no change is made to the state of either the renderbuffer object or the framebuffer object.

Calling **FramebufferRenderbuffer** with the *renderbuffer* name zero will detach the image, if any, identified by *attachment*, in the framebuffer currently bound to *target*. All state values of the attachment point specified by *attachment* in the object bound to *target* are set to their default values listed in table 6.31.

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.

Name of attachment
`COLOR_ATTACHMENT`i (see caption)
`DEPTH_ATTACHMENT`
`STENCIL_ATTACHMENT`
`DEPTH_STENCIL_ATTACHMENT`

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 level of a texture object can be attached as one of the logical buffers of the currently bound framebuffer object by calling:

`void` **FramebufferTexture**(`enum` *target*, `enum` *attachment*,
`uint` *texture*, `int` *level*);

target must be `DRAW_FRAMEBUFFER`, `READ_FRAMEBUFFER`, or `FRAMEBUFFER`. `FRAMEBUFFER` is equivalent to `DRAW_FRAMEBUFFER`. An `INVALID_OPERATION` error is generated if the value of the corresponding binding is zero. *attachment* must be one of the attachment points of the framebuffer listed in table 4.12.

If *texture* is non-zero, the specified mipmap *level* of the texture object named *texture* is attached to the framebuffer attachment point named by *attachment*. An `INVALID_VALUE` error is generated if *texture* is not the name of a texture object, or if *level* is not a supported texture level number for textures of the type corresponding to *target*. An `INVALID_OPERATION` error is generated if *texture* is the name of a buffer texture.

If *texture* is the name of a three-dimensional texture, cube map texture, one-or two-dimensional array texture, or two-dimensional multisample array texture, the texture level attached to the framebuffer attachment point is an array of images, and the framebuffer attachment is considered layered.

Additionally, a specified image from a texture object can be attached as one of the logical buffers of the currently bound framebuffer object by calling one of the following routines, depending on the type of the texture:

`void` **FramebufferTexture1D**(`enum` *target*, `enum` *attachment*,
`enum` *textarget*, `uint` *texture*, `int` *level*);
`void` **FramebufferTexture2D**(`enum` *target*, `enum` *attachment*,
`enum` *textarget*, `uint` *texture*, `int` *level*);
`void` **FramebufferTexture3D**(`enum` *target*, `enum` *attachment*,
`enum` *textarget*, `uint` *texture*, `int` *level*, `int` *layer*);

In all three routines, *target* must be `DRAW_FRAMEBUFFER`, `READ_FRAMEBUFFER`, or `FRAMEBUFFER`. `FRAMEBUFFER` is equivalent to `DRAW_FRAMEBUFFER`. An `INVALID_OPERATION` error is generated if the value of the corresponding binding is zero. *attachment* must be one of the attachment points of the framebuffer listed in table 4.12.

If *texture* is not zero, then *texture* must either name an existing texture object with an target of *textarget*, or *texture* must name an existing cube map texture and *textarget* must be one of `TEXTURE_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, or `TEXTURE_CUBE_MAP_NEGATIVE_Z`. Otherwise, an `INVALID_OPERATION` error is generated.

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

If *textarget* is `TEXTURE_RECTANGLE`, `TEXTURE_2D_MULTISAMPLE`, or `TEXTURE_2D_MULTISAMPLE_ARRAY`, then *level* must be zero. If *textarget* is `TEXTURE_3D`, then *level* must be greater than or equal to 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 texture. 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_RECTANGLE`, `TEXTURE_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, `TEXTURE_CUBE_MAP_NEGATIVE_Z`, or `TEXTURE_2D_MULTISAMPLE`.

For **FramebufferTexture3D**, if *texture* is not zero, then *textarget* must be `TEXTURE_3D`.

The command

`void` **FramebufferTextureLayer**(`enum` *target*, `enum` *attachment*, `uint` *texture*, `int` *level*, `int` *layer*);

operates identically to **FramebufferTexture3D**, except that it attaches a single layer of a three-dimensional, one-or two-dimensional array texture, or two-dimensional multisample array texture. *layer* is an integer indicating the layer number, and is treated identically to the *layer* parameter in **FramebufferTexture3D**.

The error `INVALID_VALUE` is generated if *texture* is non-zero and *layer* is negative. The error `INVALID_OPERATION` is generated if *texture* is non-zero and is not the name of a three dimensional, two-dimensional multisample array, or one-or two-dimensional array texture. Unlike **FramebufferTexture3D**, no *textarget* parameter is accepted.

If *texture* is non-zero and the command does not result in an error, the framebuffer attachment state corresponding to *attachment* is updated as in the other **FramebufferTexture** commands, except that the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER` is set to *layer*.

Effects of Attaching a Texture Image

The remaining comments in this section apply to all forms of **FramebufferTexture***.

If *texture* is zero, any image or array of images attached to the attachment point named by *attachment* is detached. Any additional parameters (*level*, *textarget*, and/or *layer*) are ignored when *texture* is zero. All state values of the attachment point specified by *attachment* are set to their default values listed in table 6.31.

If *texture* is not zero, and if **FramebufferTexture*** is successful, then the specified texture image will be used as the logical buffer identified by *attachment* of the framebuffer currently bound to *target*. State values of the specified attachment point are set as follows:

- The value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is set to `TEXTURE`.
- The value of `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` is set to *texture*.
- The value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL` is set to *level*.
- If **FramebufferTexture2D** is called and *texture* is a cube map texture, then the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE` is set to *textarget*; otherwise it is set to `TEXTURE_CUBE_MAP_POSITIVE_X`.
- If **FramebufferTextureLayer** or **FramebufferTexture3D** is called, then the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER` is set to *layer*; otherwise it is set to zero.
- If **FramebufferTexture** is called and *texture* is the name of a three-dimensional, cube map, two-dimensional multisample array, or one-or two-dimensional array texture, the value of `FRAMEBUFFER_ATTACHMENT_LAYERED` is set to `TRUE`; otherwise it is set to `FALSE`.

All other state values of the attachment point specified by *attachment* are set to their default values listed in table 6.31. No change is made to the state of the texture object, and any previous attachment to the *attachment* logical buffer of the framebuffer object bound to framebuffer *target* is broken. If the attachment is not successful, then no change is made to the state of either the texture object or the framebuffer object.

Setting *attachment* to the value `DEPTH_STENCIL_ATTACHMENT` is a special case causing both the depth and stencil attachments of the framebuffer object to be set to *texture*. *texture* must have base internal format `DEPTH_STENCIL`, or the depth and stencil framebuffer attachments will be incomplete (see section 4.4.4).

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

4.4.3 Feedback Loops Between Textures and the Framebuffer

A *feedback loop* may exist when a texture object is used as both the source and destination of a GL operation. When a feedback loop exists, undefined behavior results. This section describes *rendering feedback loops* (see section 3.9.9) and *texture copying feedback loops* (see section 3.9.2) in more detail.

Rendering Feedback Loops

The mechanisms for attaching textures to a framebuffer object do not prevent a one-or two-dimensional texture level, a face of a cube map texture level, or a layer of a two-dimensional array or three-dimensional texture from being attached to the draw framebuffer while the same texture is bound to a texture unit. While this conditions holds, texturing operations accessing that image will produce undefined results, as described at the end of section 3.9.9. Conditions resulting in such undefined behavior are defined in more detail below. Such undefined texturing operations are likely to leave the final results of 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 rendering feedback loop between the writing of pixels by GL rendering operations and the simultaneous reading of those same pixels when used as texels in the currently bound texture. In this scenario, the framebuffer will be considered framebuffer complete (see section 4.4.4), but the values of fragments rendered while in this state will be undefined. The values of texture samples may be undefined as well, as described under "Rendering Feedback Loops" in section 3.9.9

Specifically, the values of rendered fragments are undefined if all of the following conditions are true:

- an image from texture object *T* is attached to the currently bound draw framebuffer at attachment point *A*
- the texture object *T* is currently bound to a texture unit *U*, and
- the current 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 `TEXTURE_BASE_LEVEL` for the texture object *T*
- the value of `TEXTURE_MIN_FILTER` for texture object *T* is one of `NEAREST_MIPMAP_NEAREST`, `NEAREST_MIPMAP_LINEAR`, `LINEAR_MIPMAP_NEAREST`, or `LINEAR_MIPMAP_LINEAR`, and the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL` for attachment point *A* is within the the range specified by the current values of `TEXTURE_BASE_LEVEL` to q, inclusive, for the texture object *T*. (q is defined in the **Mipmapping** discussion of section 3.9.9).

For the purpose of this discussion, it is *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 3.9.19
- The active fragment or vertex shader contains any instructions that might sample from the texture object *T* bound to *U*, even if those instructions might only be executed conditionally.

Note that if `TEXTURE_BASE_LEVEL` and `TEXTURE_MAX_LEVEL` exclude any levels containing image(s) attached to the currently bound framebuffer, then the above conditions will not be met (i.e., the above rule will not cause the values of rendered fragments to be undefined.)

Texture Copying Feedback Loops

Similarly to rendering feedback loops, it is possible for a texture image to be attached to the read framebuffer while the same texture image is the destination of a **CopyTexImage*** operation, as described under "Texture Copying Feedback Loops" in section 3.9.2. While this condition holds, a texture copying feedback loop between the writing of texels by the copying operation and the reading of those same texels when used as pixels in the read framebuffer may exist. In this scenario, the values of texels written by the copying operation will be undefined (in the same fashion that overlapping copies via **BlitFramebuffer** are undefined).

Specifically, the values of copied texels are undefined if all of the following conditions are true:

- an image from texture object T is attached to the currently bound read framebuffer at attachment point A
- the selected read buffer is attachment point A
- T is bound to the texture target of a **CopyTexImage*** operation
- the *level* argument of the copying operation selects the same image that is attached to A

4.4.4 Framebuffer Completeness

A framebuffer must be *framebuffer complete* to effectively be used as the draw or read framebuffer of the GL.

The default framebuffer is always complete if it exists; however, if no default framebuffer exists (no window system-provided drawable is associated with the GL context), it is deemed to be incomplete.

A framebuffer object is said to be framebuffer complete if all of its attached images, and all framebuffer parameters required to utilize the framebuffer for rendering and reading, are consistently defined and meet the requirements defined below. The rules of framebuffer completeness are dependent on the properties of the attached images, and on certain implementation-dependent restrictions.

The internal formats of the attached images can affect the completeness of the framebuffer, so it is useful to first define the relationship between the internal format of an image and the attachment points to which it can be attached.

- The following base internal formats from table 3.16 are *color-renderable*: `ALPHA`, `RED`, `RG`, `RGB`, and `RGBA`. The sized internal formats from table 3.17 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 3.19 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 3.19 whose base internal format is `DEPTH_STENCIL`. No other formats are stencil-renderable.

Framebuffer Attachment Completeness

If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` for the framebuffer attachment point *attachment* is not `NONE`, then it is said that a framebuffer-attachable image, named *image*, is attached to the framebuffer at the attachment point. *image* is identified by the state in *attachment* as described in section 4.4.2.

The framebuffer attachment point *attachment* is said to be *framebuffer attachment complete* if the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` for *attachment* is `NONE` (i.e., no image is attached), or if all of the following conditions are true:

- *image* is a component of an existing object with the name specified by the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME`, and of the type specified by the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE`.
- The width and height of *image* are non-zero.
- If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is `TEXTURE` and the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` names a three-dimensional texture, then the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER` must be smaller than the depth of the texture.

- If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is `TEXTURE` and the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` names a one-or two-dimensional array texture, then the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER` must be smaller than the number of layers in the texture.
- If *attachment* is `COLOR_ATTACHMENT`i, 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_BUFFER`i.

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

{ `FRAMEBUFFER_INCOMPLETE_READ_BUFFER` }

- The combination of internal formats of the attached images does not violate an implementation-dependent set of restrictions.

  { `FRAMEBUFFER_UNSUPPORTED` }

- The value of `RENDERBUFFER_SAMPLES` is the same for all attached renderbuffers; the value of `TEXTURE_SAMPLES` is the same for all attached textures; and, if the attached images are a mix of renderbuffers and textures, the value of `RENDERBUFFER_SAMPLES` matches the value of `TEXTURE_SAMPLES`.

  { `FRAMEBUFFER_INCOMPLETE_MULTISAMPLE` }

- The value of `TEXTURE_FIXED_SAMPLE_LOCATIONS` is the same for all attached textures; and, if the attached images are a mix of renderbuffers and textures, the value of `TEXTURE_FIXED_SAMPLE_LOCATIONS` must be `TRUE` for all attached textures.

  { `FRAMEBUFFER_INCOMPLETE_MULTISAMPLE` }

- If any framebuffer attachment is layered, all populated attachments must be layered. Additionally, all populated color attachments must be from textures of the same target.

  { `FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS` }

The token in brackets after each clause of the framebuffer completeness rules specifies the return value of **CheckFramebufferStatus** (see below) that is generated when that clause is violated. If more than one clause is violated, it is implementation-dependent which value will be returned by **CheckFramebufferStatus**.

Performing any of the following actions may change whether the framebuffer is considered complete or incomplete:

- Binding to a different framebuffer with **BindFramebuffer**.
- Attaching an image to the framebuffer with **FramebufferTexture*** or **FramebufferRenderbuffer**.

- Detaching an image from the framebuffer with **FramebufferTexture*** or **FramebufferRenderbuffer**.
- Changing the internal format of a texture image that is attached to the framebuffer by calling **CopyTexImage*** or **CompressedTexImage***.
- Changing the internal format of a renderbuffer that is attached to the framebuffer by calling **RenderbufferStorage**.
- Deleting, with **DeleteTextures** or **DeleteRenderbuffers**, an object containing an image that is attached to a framebuffer object that is bound to the framebuffer.
- Changing the read buffer or one of the draw buffers.
- Associating a different window system-provided drawable, or no drawable, with the default framebuffer using a window system binding API such as those described in section 1.7.2.

Although the GL defines a wide variety of internal formats for framebuffer-attachable images, such as texture images and renderbuffer images, some implementations may not support rendering to particular combinations of internal formats. If the combination of formats of the images attached to a framebuffer object are not supported by the implementation, then the framebuffer is not complete under the clause labeled `FRAMEBUFFER_UNSUPPORTED`.

Implementations are required to support certain combinations of framebuffer internal formats as described under "Required Framebuffer Formats" in section 4.4.4.

Because of the *implementation-dependent* clause of the framebuffer completeness test in particular, and because framebuffer completeness can change when the set of attached images is modified, it is strongly advised, though not required, that an application check to see if the framebuffer is complete prior to rendering. The status of the framebuffer object currently bound to *target* can be queried by calling

`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 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 complete will generate an `INVALID_FRAMEBUFFER_OPERATION` error. This means that rendering commands such as **Begin**, **RasterPos**, any command that performs 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 framebuffer complete. This error is generated regardless of whether fragments are actually read from or written to the framebuffer. For example, it will be generated when a rendering command is called and the framebuffer is incomplete even if `RASTERIZER_DISCARD` is enabled.

4.4.5 Effects of Framebuffer State on Framebuffer Dependent Values

The values of the state variables listed in table 6.60 may change when a change is made to `DRAW_FRAMEBUFFER_BINDING`, to the state of the currently bound framebuffer 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 6.60 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 variables listed in table 6.60 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 6.60 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 attachments 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 attached to that point. If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` at a particular attachment point is `RENDERBUFFER`, the sizes may be determined by calling **GetRenderbufferParameteriv** as described in section 6.1.3. If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` at a particular attachment point is `TEXTURE`, the sizes may be determined by calling **GetTexParameter**, as described in section 6.1.3.

4.4.6 Mapping between Pixel and Element in Attached Image

When `DRAW_FRAMEBUFFER_BINDING` is non-zero, an operation that writes to the framebuffer modifies the image attached to the selected logical buffer, and an 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:

$$i = (x_w - b)$$

$$j = (y_w - b)$$

$$k = (layer - b)$$

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

(x_w, y_w) corresponds to a border texel if x_w, y_w, or $layer$ is less than the border width, or if x_w, y_w, or $layer$ is greater than or equal to the border width plus the width, height, or depth, respectively, of the texture image.

Conversion to Framebuffer-Attachable Image Components

When an enabled color value is written to the framebuffer while the draw framebuffer binding is non-zero, for each draw buffer the R, G, B, and A values are converted to internal components as described in table 3.16, according to the table row corresponding to the internal format of the framebuffer-attachable image attached to the selected logical buffer, and the resulting internal components are written to the image attached to logical buffer. The masking operations described in section 4.2.2 are also effective.

Conversion to RGBA Values

When a color value is read or is used as the source of a logical operation or blending while the read framebuffer binding is non-zero, the components of the framebuffer-attachable image that is attached to the logical buffer selected by `READ_BUFFER` are first converted to R, G, B, and A values according to table 3.25 and the internal format of the attached image.

4.4.7 Layered Framebuffers

A framebuffer is considered to be layered if it is complete and all of its populated attachments are layered. When rendering to a layered framebuffer, each fragment generated by the GL is assigned a layer number. The layer number for a fragment is zero if

- the fragment is generated by **DrawPixels**, **CopyPixels**, or **Bitmap**,
- geometry shaders are disabled, or
- the current geometry shader does not statically assign a value to the built-in output variable `gl_Layer`.

Layer Number	Cube Map Face
0	`TEXTURE_CUBE_MAP_POSITIVE_X`
1	`TEXTURE_CUBE_MAP_NEGATIVE_X`
2	`TEXTURE_CUBE_MAP_POSITIVE_Y`
3	`TEXTURE_CUBE_MAP_NEGATIVE_Y`
4	`TEXTURE_CUBE_MAP_POSITIVE_Z`
5	`TEXTURE_CUBE_MAP_NEGATIVE_Z`

Table 4.13: Layer numbers for cube map texture faces. The layers are numbered in the same sequence as the cube map face token values.

Otherwise, the layer for each point, line, or triangle emitted by the geometry shader is taken from the `gl_Layer` output of one of the vertices of the primitive. The vertex used is implementation-dependent. To get defined results, all vertices of each primitive emitted should set the same value for `gl_Layer`. Since the `EndPrimitive` built-in function starts a new output primitive, defined results can be achieved if `EndPrimitive` is called between two vertices emitted with different layer numbers. A layer number written by a geometry shader has no effect if the framebuffer is not layered.

When fragments are written to a layered framebuffer, the fragment's layer number selects an image from the array of images at each attachment point to use for the stencil test (see section 4.1.5), depth buffer test (see section 4.1.6), and for blending and color buffer writes (see section 4.1.8). If the fragment's layer number is negative, or greater than the minimum number of layers of any attachment, the effects of the fragment on the framebuffer contents are undefined.

When the **Clear** or **ClearBuffer*** commands are used to clear a layered framebuffer attachment, all layers of the attachment are cleared.

When commands such as **ReadPixels** or **CopyPixels** read from a layered framebuffer, the image at layer zero of the selected attachment is always used to obtain pixel values.

When cube map texture levels are attached to a layered framebuffer, there are six layers, numbered zero through five. Each layer number corresponds to a cube map face, as shown in table 4.13.

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 R^k-valued polynomial $\mathbf{p}(u)$ defined by

$$\mathbf{p}(u) = \sum_{i=0}^{n} B_i^n(u)\mathbf{R}_i \tag{5.1}$$

with $\mathbf{R}_i \in R^k$ and

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i},$$

the ith Bernstein polynomial of degree n (recall that $0^0 \equiv 1$ and $\binom{n}{0} \equiv 1$). Each $\mathbf{R}_i$ is a *control point*. The relevant command is

void **Map1{fd}**(enum *target*, T u_1, T u_2, int *stride*,
  int *order*, T *points*);

target	k	Values
`MAP1_VERTEX_3`	3	x, y, z vertex coordinates
`MAP1_VERTEX_4`	4	x, y, z, w vertex coordinates
`MAP1_INDEX`	1	color index
`MAP1_COLOR_4`	4	R, G, B, A
`MAP1_NORMAL`	3	x, y, z normal coordinates
`MAP1_TEXTURE_COORD_1`	1	s texture coordinate
`MAP1_TEXTURE_COORD_2`	2	s, t texture coordinates
`MAP1_TEXTURE_COORD_3`	3	s, t, r texture coordinates
`MAP1_TEXTURE_COORD_4`	4	s, t, r, q texture coordinates

Table 5.1: Values specified by the *target* to **Map1**. Values are given in the order in which they are taken.

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 preimage of the map. When a value u' is presented for evaluation, the formula used is

$$\mathbf{p}'(u') = \mathbf{p}(\frac{u' - u_1}{u_2 - u_1}).$$

The error `INVALID_VALUE` results if $u_1 = u_2$.

Map2 is analogous to **Map1**, except that it describes bivariate polynomials of the form

$$\mathbf{p}(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u) B_j^m(v) \mathbf{R}_{ij}.$$

The form of the **Map2** command is

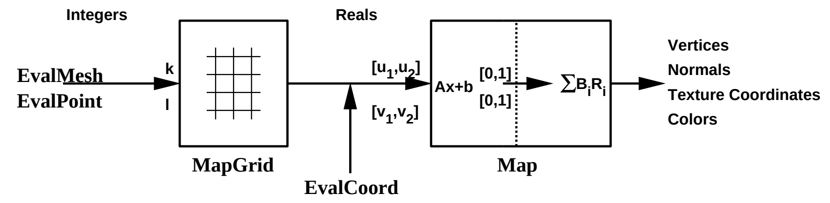


Figure 5.1. Map Evaluation.

void **Map2{fd}**(enum *target*, T u_1, T u_2, int *ustride*,
int *uorder*, T v_1, T v_2, int *vstride*, int *vorder*, T *points*);

target is a range type selected from the same group as is used for **Map1**, except 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 error INVALID_VALUE is generated if either *uorder* or *vorder* is less than one or greater than MAX_EVAL_ORDER). The values comprising $\mathbf{R}_{ij}$ are located

$$ustride \times i + vstride \times j$$

values (either single- or double-precision floating-point, as appropriate) past the first value pointed to by *points*. u_1, u_2, v_1, and v_2 define the pre-image rectangle of the map; a domain point (u', v') is evaluated as

$$\mathbf{p}'(u', v') = \mathbf{p}(\frac{u' - u_1}{u_2 - u_1}, \frac{v' - v_1}{v_2 - v_1}).$$

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 u_1 is equal to $u2$, or if v_1 is equal to v_2. If the value of ACTIVE_TEXTURE is not TEXTURE0, calling **Map{12}** 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

void **EvalCoord{12}{fd}**(T *arg*);
void **EvalCoord{12}{fd}v**(T *arg*);

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 `MAP2_VERTEX_3`, let $\mathbf{q} = \mathbf{p}$. For `MAP2_VERTEX_4`, let $\mathbf{q} = (x/w, y/w, z/w)$, where $(x, y, z, w) = \mathbf{p}$. Then let

$$\mathbf{m} = \frac{\partial \mathbf{q}}{\partial u} \times \frac{\partial \mathbf{q}}{\partial v}.$$

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.

This is done using

`void` **MapGrid1{fd}**`(` `int` n`,` `T` u'_1`,` `T` u'_2 `);`

for a one-dimensional map or

`void` **MapGrid2{fd}**`(` `int` n_u`,` `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 $n \leq 0$. 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 $n_u \leq 0$ or $n_v \leq 0$, 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

`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 $\Delta u' = (u'_2 - u'_1)/n$:

```
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 $i * \Delta u' + u'_1$ is precisely u'_1 or u'_2, respectively.

The corresponding commands for two-dimensional maps are

`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 $\Delta v' = (v'_2 - v'_1)/m$:

```
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 *mode* is LINE, then a call to **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 *mode* is POINT, then a call to **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:

void **EvalPoint1**(int *p*);

Calling it is equivalent to the command

EvalCoord1($p * \Delta u' + u'_1$);

with $\Delta u'$ and u'_1 defined as above.

void **EvalPoint2**(int *p*, int *q*);

is equivalent to the command

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.

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

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.

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

`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

`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 2.22) 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

`void` **FeedbackBuffer**(`sizei` *n*, `enum` *type*, `float` **buffer*);

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 2.22.1. 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

Type	coordinates	color	texture	total values
`2D`	x, y	–	–	2
`3D`	x, y, z	–	–	3
`3D_COLOR`	x, y, z	k	–	$3+k$
`3D_COLOR_TEXTURE`	x, y, z	k	4	$7+k$
`4D_COLOR_TEXTURE`	x, y, z, w	k	4	$8+k$

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.

which the overflow flag is reset. While in feedback mode, values are not guaranteed 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 *type* and the number of values returned for each vertex.

The command

`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

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 f
  f

tex:
  f 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.

given normally. The only exception pertains to commands that rely upon client 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 command. 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 accumulated into a display list.)

A display list is begun by calling

`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

`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

`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:

```
$offset \leftarrow 0$
for $i = 1$ to $b$
    $offset \leftarrow offset$ shifted left 8 bits
    $offset \leftarrow offset + byte$
    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 `COMPILE_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). **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. **GenLists** returns 0 if there is no group of s contiguous previously unused display list indices, or if $s = 0$.

`boolean` **IsList**(`uint` *list*);

returns `TRUE` if *list* is the index of some display list.

A contiguous group of display lists may be deleted by calling

`void` **DeleteLists**(`uint` *list*, `sizei` *range*);

where *list* is the index of the first display list to be deleted and *range* is the number of display lists to be deleted. All information about the display lists is lost, and the indices become unused. Indices to which no display list corresponds are ignored. If $range = 0$, nothing happens.

5.4.1 Commands Not Usable In Display Lists

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

Display lists: **GenLists** and **DeleteLists**.

Render modes: **FeedbackBuffer**, **SelectBuffer**, and **RenderMode**.

Vertex arrays: **ClientActiveTexture**, **ColorPointer**, **EdgeFlagPointer**, **FogCoordPointer**, **IndexPointer**, **InterleavedArrays**, **NormalPointer**, **SecondaryColorPointer**, **TexCoordPointer**, **VertexAttribPointer**, **VertexAttribIPointer**, **VertexPointer**, **PrimitiveRestartIndex**, **GenVertexArrays**, **DeleteVertexArrays**, and **BindVertexArray**.

Client state: **EnableClientState**, **DisableClientState**, **EnableVertexAttribArray**, **DisableVertexAttribArray**, **PushClientAttrib**, and **PopClientAttrib**.

Pixels and textures: **PixelStore**, **ReadPixels**, **GenTextures**, **DeleteTextures**, **AreTexturesResident**, **TexBuffer**, and **GenerateMipmap**.

Occlusion queries: **GenQueries** and **DeleteQueries**.

Buffer objects: **GenBuffers**, **DeleteBuffers**, **BindBuffer**, **BindBufferRange**, **BindBufferBase**, **TransformFeedbackVaryings**, **BufferData**, **BufferSubData**, **MapBuffer**, **MapBufferRange**, **FlushMappedBufferRange**, and **UnmapBuffer**.

Framebuffer and renderbuffer objects: **GenFramebuffers**, **BindFramebuffer**, **DeleteFramebuffers**, **CheckFramebufferStatus**, **GenRenderbuffers**,

BindRenderbuffer, **DeleteRenderbuffers**, **RenderbufferStorage**, **RenderbufferStorageMultisample**, **FramebufferTexture**, **FramebufferTexture1D**, **FramebufferTexture2D**, **FramebufferTexture3D**, **FramebufferTextureLayer**, **FramebufferRenderbuffer**, and **BlitFramebuffer**.

Program and shader objects: **CreateProgram**, **CreateShader**, **DeleteProgram**, **DeleteShader**, **AttachShader**, **DetachShader**, **BindAttribLocation**, **BindFragDataLocation**, **CompileShader**, **ShaderSource**, **LinkProgram**, and **ValidateProgram**.

GL command stream management: **ClientWaitSync**, **FenceSync**, **Finish**, and **Flush**.

Other queries: All query commands whose names begin with **Get** and **Is** (see chapter 6).

GL commands that source data from buffer objects dereference the buffer object data in question at display list compile time, rather than encoding the buffer ID and buffer offset into the display list. Only GL commands that are executed immediately, rather than being compiled into a display list, are permitted to use a buffer object as a data sink.

TexImage3D, **TexImage2D**, **TexImage1D**, **Histogram**, and **ColorTable** are executed immediately when called with the corresponding proxy arguments `PROXY_TEXTURE_3D` or `PROXY_TEXTURE_2D_ARRAY`; `PROXY_TEXTURE_2D` `PROXY_TEXTURE_1D_ARRAY`, or `PROXY_TEXTURE_CUBE_MAP`; `PROXY_TEXTURE_1D`; `PROXY_HISTOGRAM`; and `PROXY_COLOR_TABLE`, `PROXY_POST_CONVOLUTION_COLOR_TABLE`, or `PROXY_POST_COLOR_MATRIX_COLOR_TABLE`.

When a program object is in use, a display list may be executed whose vertex attribute calls do not match up exactly with what is expected by the vertex shader contained in that program object. Handling of this mismatch is described in section 2.14.3.

Display lists require one bit of state to indicate whether a GL command should be executed immediately or placed in a display list. In the initial state, commands are executed immediately. If the bit indicates display list creation, an index is required to indicate the current display list being defined. Another bit indicates, during display list creation, whether or not commands should be executed as they are compiled into the display list. One integer is required for the current **ListBase** setting; its initial value is zero. Finally, state must be maintained to indicate which integers are currently in use as display list indices. In the initial state, no indices are in use.

5.5 Flush and Finish

The command

`void` **Flush**(`void`);

indicates that all commands that have previously been sent to the GL must complete in finite time.

The command

`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 Sync Objects and Fences

Sync objects act as a *synchronization primitive* - a representation of events whose completion status can be tested or waited upon. Sync objects may be used for synchronization with operations occuring in the GL state machine or in the graphics pipeline, and for synchronizing between multiple graphics contexts, among other purposes.

Sync objects have a status value with two possible states: *signaled* and *unsignaled*. Events are associated with a sync object. When a sync object is created, its status is set to unsignaled. When the associated event occurs, the sync object is signaled (its status is set to signaled). The GL may be asked to wait for a sync object to become signaled.

Initially, only one specific type of sync object is defined: the fence sync object, whose associated event is triggered by a fence command placed in the GL command stream. Fence sync objects are used to wait for partial completion of the GL command stream, as a more flexible form of **Finish**.

The command

`sync` **FenceSync**(`enum` *condition*, `bitfield` *flags*);

creates a new fence sync object, inserts a fence command in the GL command stream and associates it with that sync object, and returns a non-zero name corresponding to the sync object.

Property Name	Property Value
`OBJECT_TYPE`	`SYNC_FENCE`
`SYNC_CONDITION`	*condition*
`SYNC_STATUS`	`UNSIGNALED`
`SYNC_FLAGS`	*flags*

Table 5.3: Initial properties of a sync object created with **FenceSync**.

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

condition must be `SYNC_GPU_COMMANDS_COMPLETE`. This condition is satisfied by completion of the fence command corresponding to the sync object and all preceding commands in the same command stream. The sync object will not be signaled until all effects from these commands on GL client and server state and the framebuffer are fully realized. Note that completion of the fence command occurs once the state of the corresponding sync object has been changed, but commands waiting on that sync object may not be unblocked until after the fence command completes.

flags must be 0[1].

Each sync object contains a number of *properties* which determine the state of the object and the behavior of any commands associated with it. Each property has a *property name* and *property value*. The initial property values for a sync object created by **FenceSync** are shown in table 5.3.

Properties of a sync object may be queried with **GetSynciv** (see section 6.1.13). The `SYNC_STATUS` property will be changed to `SIGNALED` when *condition* is satisfied.

If **FenceSync** fails to create a sync object, zero will be returned and a GL error will be generated as described. An `INVALID_ENUM` error is generated if *condition* is not `SYNC_GPU_COMMANDS_COMPLETE`. If *flags* is not zero, an `INVALID_VALUE` error is generated

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

`void` **DeleteSync**(`sync` *sync*);

If the fence command corresponding to the specified sync object has completed, or if no **ClientWaitSync** or **WaitSync** commands are blocking on *sync*, the

---

[1] *flags* is a placeholder for anticipated future extensions of fence sync object capabilities.

object is deleted immediately. Otherwise, *sync* is flagged for deletion and will be deleted when it is no longer associated with any fence command and is no longer blocking any **ClientWaitSync** or **WaitSync** command. In either case, after returning from **DeleteSync** the *sync* name is invalid and can no longer be used to refer to the sync object.

DeleteSync will silently ignore a *sync* value of zero. An `INVALID_VALUE` error is generated if *sync* is neither zero nor the name of a sync object.

5.6.1 Waiting for Sync Objects

The command

`enum` **ClientWaitSync**(`sync` *sync*, `bitfield` *flags*,
`uint64` *timeout*);

causes the GL to block, and will not return until the sync object *sync* is signaled, or until the specified *timeout* period expires. *timeout* is in units of nanoseconds. *timeout* is adjusted to the closest value allowed by the implementation-dependent timeout accuracy, which may be substantially longer than one nanosecond, and may be longer than the requested period.

If *sync* is signaled at the time **ClientWaitSync** is called, then **ClientWaitSync** returns immediately. If *sync* is unsignaled at the time **ClientWaitSync** is called, then **ClientWaitSync** will block and will wait up to *timeout* nanoseconds for *sync* to become signaled. *flags* controls command flushing behavior, and may be `SYNC_FLUSH_COMMANDS_BIT`, as discussed in section 5.6.2.

ClientWaitSync returns one of four status values. A return value of `ALREADY_SIGNALED` indicates that *sync* was signaled at the time **ClientWaitSync** was called. `ALREADY_SIGNALED` will always be returned if *sync* was signaled, even if the value of *timeout* is zero. A return value of `TIMEOUT_EXPIRED` indicates that the specified timeout period expired before *sync* was signaled. A return value of `CONDITION_SATISFIED` indicates that *sync* was signaled before the timeout expired. Finally, if an error occurs, in addition to generating a GL error as specified below, **ClientWaitSync** immediately returns `WAIT_FAILED` without blocking.

If the value of *timeout* is zero, then **ClientWaitSync** does not block, but simply tests the current state of *sync*. `TIMEOUT_EXPIRED` will be returned in this case if *sync* is not signaled, even though no actual wait was performed.

If *sync* is not the name of a sync object, an `INVALID_VALUE` error is generated. If *flags* contains any bits other than `SYNC_FLUSH_COMMANDS_BIT`, an `INVALID_VALUE` error is generated.

The command

```
void WaitSync( sync sync, bitfield flags,
   uint64 timeout );
```

is similar to **ClientWaitSync**, but instead of blocking and not returning to the application until *sync* is signaled, **WaitSync** returns immediately, instead causing the GL server to block [2] until *sync* is signaled [3].

sync has the same meaning as for **ClientWaitSync**.

timeout must currently be the special value `TIMEOUT_IGNORED`, and is not used. Instead, **WaitSync** will always wait no longer than an implementation-dependent timeout. The duration of this timeout in nanoseconds may be queried by calling **GetInteger64v** with *value* `MAX_SERVER_WAIT_TIMEOUT`. There is currently no way to determine whether **WaitSync** unblocked because the timeout expired or because the sync object being waited on was signaled.

flags must be 0.

If an error occurs, **WaitSync** generates a GL error as specified below, and does not cause the GL server to block.

If *sync* is not the name of a sync object, an `INVALID_VALUE` error is generated. If *timeout* is not `TIMEOUT_IGNORED` or *flags* is not zero, an `INVALID_VALUE` error is generated[4].

Multiple Waiters

It is possible for both the GL client to be blocked on a sync object in a **ClientWaitSync** command, the GL server to be blocked as the result of a previous **WaitSync** command, and for additional **WaitSync** commands to be queued in the GL server, all for a single sync object. When such a sync object is signaled in this situation, the client will be unblocked, the server will be unblocked, and all such queued **WaitSync** commands will continue immediately when they are reached.

See appendix D.2 for more information about blocking on a sync object in multiple GL contexts.

---

[2] The GL server may choose to wait either in the CPU executing server-side code, or in the GPU hardware if it supports this operation.

[3] **WaitSync** allows applications to continue to queue commands from the client in anticipation of the sync being signalled, increasing client-server parallelism.

[4] *flags* and *timeout* are placeholders for anticipated future extensions of sync object capabilities. They must have these reserved values in order that existing code calling **WaitSync** operate properly in the presence of such extensions.

5.6.2 Signalling

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

If the sync object being blocked upon will not be signaled in finite time (for example, by an associated fence command issued previously, but not yet flushed to the graphics pipeline), then **ClientWaitSync** may hang forever. To help prevent this behavior [5], if the `SYNC_FLUSH_COMMANDS_BIT` bit is set in *flags*, and *sync* is unsignaled when **ClientWaitSync** is called, then the equivalent of **Flush** will be performed before blocking on *sync*.

If a sync object is marked for deletion while a client is blocking on that object in a **ClientWaitSync** command, or a GL server is blocking on that object as a result of a prior **WaitSync** command, deletion is deferred until the sync object is signaled and all blocked GL clients and servers are unblocked.

Additional constraints on the use of sync objects are discussed in appendix D.

State must be maintained to indicate which sync object names are currently in use. The state require for each sync object in use is an integer for the specific type, an integer for the condition, and a bit indicating whether the object is signaled or unsignaled. The initial values of sync object state are defined as specified by **FenceSync**.

5.7 Hints

Certain aspects of GL behavior, when there is room for variation, may be controlled with hints. A hint is specified using

`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 5.4; 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

---

[5] The simple flushing behavior defined by `SYNC_FLUSH_COMMANDS_BIT` will not help when waiting for a fence command issued in another context's command stream to complete. Applications which block on a fence sync object must take additional steps to assure that the context from which the corresponding fence command was issued has flushed that command to the graphics pipeline.

Target	Hint description
`PERSPECTIVE_CORRECTION_HINT`	Quality of parameter interpolation
`POINT_SMOOTH_HINT`	Point sampling quality
`LINE_SMOOTH_HINT`	Line sampling quality
`POLYGON_SMOOTH_HINT`	Polygon sampling quality
`FOG_HINT`	Fog quality (calculated per-pixel or per-vertex)
`GENERATE_MIPMAP_HINT`	Quality and performance of automatic mipmap level generation
`TEXTURE_COMPRESSION_HINT`	Quality and performance of texture image compression
`FRAGMENT_SHADER_DERIVATIVE_HINT`	Derivative accuracy for fragment processing built-in functions `dFdx`, `dFdy` and `fwidth`

Table 5.4: Hint targets and descriptions.

the texture images be compressed with as little image degradation as possible. `FASTEST` should be used for one-time texture compression, and `NICEST` should be used if the compression results are to be retrieved by **GetCompressedTexImage** (section 6.1.4) for reuse.

The interpretation of hints is implementation-dependent. An implementation may ignore them entirely.

The initial value of all hints is `DONT_CARE`.

Chapter 6

State and State Requests

The state required to describe the GL machine is enumerated in section 6.2. Most state is set through the calls described in previous chapters, and can be queried using the calls described in section 6.1.

6.1 Querying GL State

6.1.1 Simple Queries

Much of the GL state is completely identified by symbolic constants. The values of these state variables can be obtained using a set of **Get** commands. There are four commands for obtaining simple state variables:

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

The commands obtain boolean, integer, 64-bit integer, floating-point, or double-precision state variables. *value* is a symbolic constant indicating the state variable to return. *data* is a pointer to a scalar or array of the indicated type in which to place the returned data.

Indexed simple state variables are queried with the commands

`void` **GetBooleani_v**(`enum` *target*, `uint` *index*,
  `boolean` **data*);
`void` **GetIntegeri_v**(`enum` *target*, `uint` *index*, `int` **data*);

`void` **GetInteger64i_v**(`enum` *target*, `uint` *index*,
`int64` **data*);

target is the name of the indexed state and *index* is the index of the particular element being queried. *data* is a pointer to a scalar or array of the indicated type in which to place the returned data. An `INVALID_VALUE` error is generated if *index* is outside the valid range for the indexed state *target*.

Finally,

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

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

`boolean` **IsEnabledi**(`enum` *target*, `uint` *index*);

can be used to determine if the indexed state corresponding to *target* and *index* is enabled or disabled. An `INVALID_VALUE` error is generated if *index* is outside the valid range for the indexed state *target*.

6.1.2 Data Conversions

If a **Get** command is issued that returns value types different from the type of the value being obtained, a type conversion is performed. If **GetBooleanv** is called, a floating-point or integer value converts to `FALSE` if and only if it is zero (otherwise it converts to `TRUE`). If any of the other simple queries are called, a boolean value of `TRUE` or `FALSE` is interpreted as 1 or 0, respectively. If **GetIntegerv** or **GetInteger64v** are called, a floating-point value is rounded to the nearest integer, unless the value is an RGBA color component, a normal coordinate, a **DepthRange** value, or a depth buffer clear value. In these cases, the **Get** command converts the floating-point value to an integer according to the `INT` entry of table 4.9; a value not in $[-1, 1]$ converts to an undefined value. If **GetFloatv** is called, a boolean value of `TRUE` or `FALSE` is interpreted as 1.0 or 0.0, respectively, an integer is coerced to floating-point, and a double-precision floating-point value is converted to single-precision. Analogous conversions are carried out in the case of **GetDoublev**. If a value is so large in magnitude that it cannot be represented with the requested type, then the nearest value representable using the requested type is returned.

Unless otherwise indicated, multi-valued state variables return their multiple values in the same order as they are given as arguments to the commands that set them. For instance, the two **DepthRange** parameters are returned in the order *n* followed by *f*. Similarly, points for evaluator maps are returned in the order that they appeared when passed to **Map1**. **Map2** returns $\mathbf{R}_{ij}$ in the $[(uorder)i + j]$th block of values (see page 345 for i, j, *uorder*, and $\mathbf{R}_{ij}$).

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 6.6, 6.7, 6.13, 6.20, 6.24, and 6.52 indicate Table 6.20 indicates 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 6.7- 6.10 define the set of state stored in a vertex array object.

6.1.3 Enumerated Queries

Other commands exist to obtain state variables that are identified by a category as well as a symbolic constant.

`void` **GetClipPlane**(`enum` *plane*`,` `double` *eqn[4]* `);`

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).

`void` **GetLight{if}v**(`enum` *light*`,` `enum` *value*`,` `T` *data* `);`

places information about light parameter *value* for *light* in *data*. `POSITION` and `SPOT_DIRECTION` return values in eye coordinates. Again, these are the coordinates that were computed when the position or direction was specified.

`void` **GetMaterial{if}v**(`enum` *face*`,` `enum` *value*`,` `T` *data* `);`

places information about material property *value* for *face* in *data*. *face* must be either `FRONT` or `BACK`, indicating the front or back material, respectively.

`void` **GetTexEnv{if}v**(`enum` *env*`,` `enum` *value*`,` `T` *data* `);`

places information about *value* for *env* in *data*. *env* must be either `POINT_SPRITE`, `TEXTURE_ENV`, or `TEXTURE_FILTER_CONTROL`.

`void` **GetTexGen{ifd}v**(`enum` *coord*`,` `enum` *value*`,` `T` *data* `);`

places information about *value* for *coord* in *data*. *coord* must be one of `S`, `T`, `R`, or `Q`. `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.

`void` **GetPixelMap{ui us f}v**(`enum` *map*`,` `T` *data* `);`

returns all values in the pixel map *map* in *data*. *map* must be a map name from table 3.3. **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.

`void` **GetMap{ifd}v**(`enum` *map*, `enum` *value*, `T` *data*);

places information about *value* for *map* in *data*. *map* must be one of the map types described in section 5.1, and *value* must be one of `ORDER`, `COEFF`, or `DOMAIN`.

The commands

`void` **GetTexParameter{if}v**(`enum` *target*, `enum` *value*, `T` *data*);
`void` **GetTexParameterI{i ui}v**(`enum` *target*, `enum` *value*, `T` *data*);

place information about texture parameter *value* for the specified *target* into *data*. *value* must be `TEXTURE_RESIDENT` or one of the symbolic values in table 3.22.

target may be one of `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`, `TEXTURE_RECTANGLE`, or `TEXTURE_CUBE_MAP`, indicating the currently bound one-, two-, three-dimensional, one- or two-dimensional array, rectangular, or cube map texture object.

Querying *value* `TEXTURE_BORDER_COLOR` with **GetTexParameterIiv** or **GetTexParameterIuiv** returns the border color values as signed integers or unsigned integers, respectively; otherwise the values are returned as described in section 6.1.2. If the border color is queried with a type that does not match the original type with which it was specified, the result is undefined.

`void` **GetTexLevelParameter{if}v**(`enum` *target*, `int` *lod*, `enum` *value*, `T` *data*);

places information about texture image parameter *value* for level-of-detail *lod* of the specified *target* into *data*. *value* must be one of the symbolic values in table 6.23.

target may be one of `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`, `TEXTURE_RECTANGLE`, `TEXTURE_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, `TEXTURE_CUBE_MAP_NEGATIVE_Z`, `TEXTURE_2D_MULTISAMPLE`, `TEXTURE_2D_MULTISAMPLE_ARRAY`, `PROXY_TEXTURE_1D`, `PROXY_TEXTURE_2D`, `PROXY_TEXTURE_3D`, `PROXY_TEXTURE_1D_ARRAY`, `PROXY_TEXTURE_2D_ARRAY`, `PROXY_TEXTURE_RECTANGLE`, `PROXY_TEXTURE_CUBE_MAP`, `PROXY_TEXTURE_2D_MULTISAMPLE`, or `PROXY_TEXTURE_2D_MULTISAMPLE_ARRAY`, indicating the one-, two-, or three-dimensional texture, one-or two-dimensional array texture, rectangular texture, one of the six distinct 2D images making up the cube map texture object, two-dimensional multisample texture, two-dimensional multisample array texture; or the one-, two-, three-dimensional, one-or two-dimensional array, rectangular, cube map, two-dimensional multisample, or two-dimensional multisample array proxy state vector.

target may also be `TEXTURE_BUFFER`, indicating the texture buffer. In the case *lod* must be zero or an `INVALID_VALUE` error is generated.

Note that `TEXTURE_CUBE_MAP` is not a valid *target* parameter for **GetTexLevelParameter**, because it does not specify a particular cube map face.

lod determines which level-of-detail's state is returned. If *lod* is less than zero or larger than the maximum allowable level-of-detail, then an `INVALID_VALUE` error is generated.

For texture images with uncompressed internal formats, queries of *value* `TEXTURE_RED_TYPE`, `TEXTURE_GREEN_TYPE`, `TEXTURE_BLUE_TYPE`, `TEXTURE_ALPHA_TYPE`, `TEXTURE_LUMINANCE_TYPE`, `TEXTURE_INTENSITY_TYPE`, and `TEXTURE_DEPTH_TYPE` return the data type used to store the component. Types `NONE`, `SIGNED_NORMALIZED`, `UNSIGNED_NORMALIZED`, `FLOAT`, `INT`, and `UNSIGNED_INT` respectively indicate missing, signed normalized fixed-point, unsigned normalized fixed-point, floating-point, signed unnormalized integer, and unsigned unnormalized integer components. Queries of *value* `TEXTURE_RED_SIZE`, `TEXTURE_GREEN_SIZE`, `TEXTURE_BLUE_SIZE`, `TEXTURE_ALPHA_SIZE`, `TEXTURE_LUMINANCE_SIZE`, `TEXTURE_INTENSITY_SIZE`, `TEXTURE_DEPTH_SIZE`, `TEXTURE_STENCIL_SIZE`, and `TEXTURE_SHARED_SIZE` return the actual resolutions of the stored image array components, not the resolutions specified when the image array was defined.

For texture images with compressed internal formats, the types returned specify how components are interpreted after decompression, while the resolutions returned specify the component resolution of an uncompressed internal format that produces an image of roughly the same quality as the compressed image in question. Since the quality of the implementation's compression algorithm is likely data-dependent, the returned component sizes should be treated only as rough approximations.

Querying *value* `TEXTURE_COMPRESSED_IMAGE_SIZE` returns the size (in `ubytes`) of the compressed texture image that would be returned by **GetCompressedTexImage** (section 6.1.4). Querying `TEXTURE_COMPRESSED_IMAGE_SIZE` is not allowed on texture images with an uncompressed internal format or on proxy targets and will result in an `INVALID_OPERATION` error if attempted.

Queries of *value* `TEXTURE_BORDER`, `TEXTURE_WIDTH`, `TEXTURE_HEIGHT`, and `TEXTURE_DEPTH` return the border width, width, height, and depth 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.

6.1.4 Texture Queries

The command

void **GetTexImage**(enum *tex*, int *lod*, enum *format*,
  enum *type*, void **img*);

is used to obtain texture images. It is somewhat different from the other **Get*** commands; *tex* is a symbolic value indicating which texture (or texture face in the case of a cube map texture target name) is to be obtained. `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`, and `TEXTURE_RECTANGLE` indicate a one-, two-, or three-dimensional, one- or two-dimensional array, or rectangular texture respectively. `TEXTURE_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, and `TEXTURE_CUBE_MAP_NEGATIVE_Z` indicate the respective face of a cube map texture. *lod* is a level-of-detail number, *format* is a pixel format from table 3.6, *type* is a pixel type from table 3.5.

Any of the following mismatches between *format* and the internal format of the texture image will generate an `INVALID_OPERATION` error:

- *format* is a color format (one of the formats in table 3.6 whose target is the color buffer) and the base internal format of the texture image is not a color format.
- *format* is `DEPTH_COMPONENT` and the base internal format is not `DEPTH_COMPONENT` or `DEPTH_STENCIL`.
- *format* is `DEPTH_STENCIL` and the base internal format is not `DEPTH_STENCIL`.
- *format* is one of the integer formats in table 3.6 and the internal format of the texture image is not integer, or *format* is not one of the integer formats in table 3.6 and the internal format is integer.

GetTexImage obtains component groups from a texture image with the indicated level-of-detail. If *format* is a color format then the components are assigned among R, G, B, and A according to table 6.1, starting with the first group in the first row, and continuing by obtaining groups in order from each row and proceeding from the first row to the last, and from the first image to the last for three-dimensional textures. One- and two-dimensional array textures are treated as two- and three-dimensional images, respectively, where the layers are treated as rows or images. If *format* is `DEPTH_COMPONENT`, then each depth component is assigned with the same ordering of rows and images. If *format* is `DEPTH_STENCIL`, then each depth component and each stencil index is assigned with the same ordering of rows and images.

These groups are then packed and placed in client or pixel buffer object memory. If a pixel pack buffer is bound (as indicated by a non-zero value of `PIXEL_PACK_BUFFER_BINDING`), *img* is an offset into the pixel pack buffer; otherwise, *img* is a pointer to client memory. 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 operations are applied as if the image were two-dimensional, except that the additional pixel storage state values `PACK_IMAGE_HEIGHT` and `PACK_SKIP_IMAGES` are applied. The correspondence of texels to memory locations is as defined for **TexImage3D** in section 3.9.1.

The row length, number of rows, image depth, and number of images are determined by the size of the texture image (including any borders). Calling **GetTexImage** with *lod* less than zero or larger than the maximum allowable causes the error `INVALID_VALUE`. Calling **GetTexImage** with a *format* of `COLOR_INDEX` or `STENCIL_INDEX` causes the error `INVALID_ENUM`. Calling **GetTexImage** with a non-zero *lod* when *tex* is `TEXTURE_RECTANGLE` causes the error `INVALID_VALUE`. If a pixel pack buffer object is bound and packing the texture image into the buffer's memory would exceed the size of the buffer, an `INVALID_OPERATION` error results. If a pixel pack buffer object is bound and *img* is not evenly divisible by the number of basic machine units needed to store in memory the GL data type corresponding to *type* (see table 3.5), an `INVALID_OPERATION` error results.

The command

```
void GetCompressedTexImage( enum target, int lod,
    void *img );
```

is used to obtain texture images stored in compressed form. The parameters *target*, *lod*, and *img* are interpreted in the same manner as in **GetTexImage**. When called, **GetCompressedTexImage** writes n `ubyte`s of compressed image data to the pixel pack buffer or client memory pointed to by *img*, where n is the value of `TEXTURE_COMPRESSED_IMAGE_SIZE` for the texture. The compressed image data is formatted according to the definition of the texture's internal format. All

Base Internal Format	R	G	B	A
`ALPHA`	0	0	0	A_i
`LUMINANCE`	L_i	0	0	1
`LUMINANCE_ALPHA`	L_i	0	0	A_i
`INTENSITY`	I_i	0	0	1
`RED`	R_i	0	0	1
`RG`	R_i	G_i	0	1
`RGB` (or 3)	R_i	G_i	B_i	1
`RGBA` (or 4)	R_i	G_i	B_i	A_i

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.

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

`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

`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.

6.1.7 Color Table Query

The current contents of a color table are queried using

`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

`void` **GetColorTableParameter**{**if**}**v**(`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

> `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*. Pixel processing and component mapping are identical to those of **GetTexImage**.

The current contents of a separable filter image are queried using

> `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

> `void` **GetConvolutionParameter**{**if**}**v**(`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

> `void` **GetHistogram**(`enum` *target*, `boolean` *reset*,
> `enum` *format*, `enum` *type*, `void`$*$ *values*);

target must be `HISTOGRAM`. *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 histogram table image is returned to pixel pack buffer or client memory starting at *values*. 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

`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}v**(`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`. *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`. 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

`void` **GetMinmaxParameter**{**if**}**v**(`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

`void` **GetPointerv**(`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 [1]. The command

---

[1] Applications making copies of these static strings should never use a fixed-length buffer, because the strings may grow unpredictably between releases, resulting in buffer overflow when copying. This is particularly true of the `EXTENSIONS` string, which has become extremely long in some GL implementations.

Value	OpenGL Profile
`CONTEXT_CORE_PROFILE_BIT`	Core
`CONTEXT_COMPATIBILITY_PROFILE_BIT`	Compatibility

Table 6.2: Context profile bits returned by the `CONTEXT_PROFILE_MASK` query.

`ubyte` ***GetString**(`enum` *name*);

accepts *name* values of `RENDERER`, `VENDOR`, `EXTENSIONS`, `VERSION`, and `SHADING_LANGUAGE_VERSION`. 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>

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 version, profile, and additional properties of the context may also be queried by calling **GetIntegerv** with *values* `MAJOR_VERSION` and `MINOR_VERSION`, which respectively return the same values as *major_number* and *minor_number* in the `VERSION` string; with *value* `CONTEXT_PROFILE_MASK`, which returns a mask containing one of the bits in table 6.2, corresponding to the API profile implemented by the context (see appendix E.1); or with *value* `CONTEXT_FLAGS`, which returns a set of flags defining additional properties of a context. If `CONTEXT_FLAG_FORWARD_COMPATIBLE_BIT` is set in `CONTEXT_FLAGS`, then the context is a forward-compatible context as defined in appendix E, and the deprecated features described in that appendix are not supported; otherwise the context is a full context, and all features described in the specification are supported.

Indexed strings are queried with the command

`ubyte` ***GetStringi**(`enum` *name*, `uint` *index*);

name is the name of the indexed state and *index* is the index of the particular element being queried. *name* may only be `EXTENSIONS`, indicating that the extension name corresponding to the $index$th supported extension should be returned. $index$ 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 no defined relationship between any particular extension name and the $index$ values; an extension name may correspond to a different $index$ in different GL contexts and/or implementations.

An `INVALID_VALUE` error is generated if *index* is outside the valid range for the indexed state *name*.

6.1.12 Asynchronous Queries

The command

`boolean` **IsQuery**(`uint` *id*);

returns `TRUE` if *id* is the name of a query object. If *id* is zero, or if *id* is a non-zero value that is not the name of a query object, **IsQuery** returns `FALSE`.

Information about a query target can be queried with the command

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

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

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

If *pname* is `QUERY_COUNTER_BITS`, the implementation-dependent number of bits used to hold the query result for *target* will be placed in *params*. The number of query counter bits may be zero, in which case the counter contains no useful information.

For primitive queries (`PRIMITIVES_GENERATED` and `TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN`) if the number of bits is non-zero, the minimum number of bits allowed is 32.

For occlusion queries (`SAMPLES_PASSED`), if the number of bits is non-zero, the minimum number of bits allowed is a function of the implementation's maximum viewport dimensions (`MAX_VIEWPORT_DIMS`). The counter must be able to represent at least two overdraws for every pixel in the viewport. The formula to compute the allowable minimum value (where n is the minimum number of bits) is

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

The state of a query object can be queried with the commands

void **GetQueryObjectiv**(uint *id*, enum *pname*,
  int **params*);
void **GetQueryObjectuiv**(uint *id*, enum *pname*,
  uint **params*);

If *id* is not the name of a query object, or if the query object named by *id* is currently active, then an `INVALID_OPERATION` error is generated.

If *pname* is `QUERY_RESULT`, then the query object's result value is returned as a single integer in *params*. If the value is so large in magnitude that it cannot be represented with the requested type, then the nearest value representable using the requested type is returned. If the number of query counter bits for *target* is zero, then the result is returned as a single integer with the value zero.

There may be an indeterminate delay before the above query returns. If *pname* is `QUERY_RESULT_AVAILABLE`, `FALSE` is returned if such a delay would be required; otherwise `TRUE` is returned. It must always be true that if any query object returns a result available of `TRUE`, all queries of the same type issued prior to that query must also return `TRUE`.

Querying the state for any given query object forces that occlusion query to complete within a finite amount of time.

If multiple queries are issued using the same object name prior to calling **GetQueryObject[u]iv**, the result and availability information returned will always be from the last query issued. The results from any queries before the last one will be lost if they are not retrieved before starting a new query on the same *target* and *id*.

6.1.13 Sync Object Queries

Properties of sync objects may be queried using the command

void **GetSynciv**(sync *sync*, enum *pname*, sizei *bufSize*,
  sizei **length*, int **values*);

The value or values being queried are returned in the parameters *length* and *values*.

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

If *pname* is `OBJECT_TYPE`, a single value representing the specific type of the sync object is placed in *values*. The only type supported is `SYNC_FENCE`.

If *pname* is `SYNC_STATUS`, a single value representing the status of the sync object (`SIGNALED` or `UNSIGNALED`) is placed in *values*.

If *pname* is `SYNC_CONDITION`, a single value representing the condition of the sync object is placed in *values*. The only condition supported is `SYNC_GPU_COMMANDS_COMPLETE`.

If *pname* is `SYNC_FLAGS`, a single value representing the flags with which the sync object was created is placed in *values*. No flags are currently supported.

If *sync* is not the name of a sync object, an `INVALID_VALUE` error is generated. If *pname* is not one of the values described above, an `INVALID_ENUM` error is generated. If an error occurs, nothing will be written to *values* or *length*.

The command

`boolean` **IsSync**(`sync` *sync*);

returns `TRUE` if *sync* is the name of a sync object. If *sync* is not the name of a sync object, or if an error condition occurs, **IsSync** returns `FALSE` (note that zero is not the name of a sync object).

Sync object names immediately become invalid after calling **DeleteSync**, as discussed in sections 5.6 and D.2, but the underlying sync object will not be deleted until it is no longer associated with any fence command and no longer blocking any ***WaitSync** command.

6.1.14 Buffer Object Queries

The command

`boolean` **IsBuffer**(`uint` *buffer*);

returns `TRUE` if *buffer* is the name of an buffer object. If *buffer* is zero, or if *buffer* is a non-zero value that is not the name of an buffer object, **IsBuffer** returns `FALSE`.

The commands

`void` **GetBufferParameteriv**(`enum` *target*, `enum` *pname*,
`int` **data*);
`void` **GetBufferParameteri64v**(`enum` *target*, `enum` *pname*,
`int64` **data*);

return information about a bound buffer object. *target* must be one of the targets listed in table 2.7, and *pname* must be one of the buffer object parameters in table 2.8, other than `BUFFER_MAP_POINTER`. The value of the specified parameter of the buffer object bound to *target* is returned in *data*.

The command

`void` **GetBufferSubData**(`enum` *target*, `intptr` *offset*, `sizeiptr` *size*, `void` **data*);

queries the data contents of a buffer object. *target* must be one of the targets listed in table 2.7. *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

`void` **GetBufferPointerv**(`enum` *target*, `enum` *pname*, `void` ***params*);

with *target* set to one of the targets listed in table 2.7 and *pname* set to `BUFFER_MAP_POINTER`. The single buffer map pointer is returned in *params*. **GetBufferPointerv** returns the `NULL` pointer value if the buffer's data store is not currently mapped, or if the requesting client did not map the buffer object's data store, and the implementation is unable to support mappings on multiple clients.

To query which buffer objects are bound to the array of uniform buffer binding points and will be used as the storage for active uniform blocks, call **GetIntegeri_v** with *param* set to `UNIFORM_BUFFER_BINDING`. *index* must be in the range zero to the value of `MAX_UNIFORM_BUFFER_BINDINGS` - 1. The name of the buffer object bound to *index* is returned in *values*. If no buffer object is bound for *index*, zero is returned in *values*.

To query the starting offset or size of the range of each buffer object binding used for uniform buffers, call **GetInteger64i_v** with *param* set to `UNIFORM_BUFFER_START` or `UNIFORM_BUFFER_SIZE` respectively. *index* must be in the range zero to the value of `MAX_UNIFORM_BUFFER_BINDINGS` - 1. If the parameter (starting offset or size) was not specified when the buffer object was bound, zero is returned. If no buffer object is bound to *index*, -1 is returned.

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 **GetIntegeri_v** 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 **GetInteger64i_v** 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.

6.1.15 Vertex Array Object Queries

The command

`boolean` **IsVertexArray**(`uint` *array*);

returns `TRUE` if *array* is the name of a vertex array object. If *array* is zero, or a non-zero value that is not the name of a vertex array object, **IsVertexArray** returns `FALSE`. No error is generated if *array* is not a valid vertex array object name.

6.1.16 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

`boolean` **IsShader**(`uint` *shader*);

returns `TRUE` if *shader* is the name of a shader object. If *shader* is zero, or a non-zero value that is not the name of a shader object, **IsShader** returns `FALSE`. No error is generated if *shader* is not a valid shader object name.

The command

`void` **GetShaderiv**(`uint` *shader*, `enum` *pname*, `int` **params*);

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

If *pname* is `SHADER_TYPE`, `VERTEX_SHADER`, `GEOMETRY_SHADER`, or `FRAGMENT_SHADER` is returned if *shader* is a vertex, geometry, or fragment shader object respectively. If *pname* is `DELETE_STATUS`, `TRUE` is returned if the shader has been flagged for deletion and `FALSE` is returned otherwise. If *pname* is `COMPILE_STATUS`, `TRUE` is returned if the shader was last compiled successfully, and `FALSE` is returned otherwise. If *pname* is `INFO_LOG_LENGTH`, the length of the info log, including a null terminator, is returned. If there is no info log, zero is returned. If *pname* is `SHADER_SOURCE_LENGTH`, the length of the concatenation of the source strings making up the shader source, including a null terminator, is returned. If no source has been defined, zero is returned.

The command

`boolean` **IsProgram**(`uint` *program*);

returns `TRUE` if *program* is the name of a program object. If *program* is zero, or a non-zero value that is not the name of a program object, **IsProgram** returns `FALSE`. No error is generated if *program* is not a valid program object name.

The command

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

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

If *pname* is `DELETE_STATUS`, `TRUE` is returned if the program has been flagged for deletion, and `FALSE` is returned otherwise. If *pname* is `LINK_STATUS`, `TRUE` is returned if the program was last compiled successfully, and `FALSE` is returned otherwise. If *pname* is `VALIDATE_STATUS`, `TRUE` is returned if the last call to **ValidateProgram** with *program* was successful, and `FALSE` is returned otherwise. If *pname* is `INFO_LOG_LENGTH`, the length of the info log, including a null terminator, is returned. If there is no info log, zero is returned. If *pname* is `ATTACHED_SHADERS`, the number of objects attached is returned. If *pname* is `ACTIVE_ATTRIBUTES`, the number of active attributes in *program* is returned. If no active attributes exist, zero is returned. If *pname* is `ACTIVE_ATTRIBUTE_MAX_LENGTH`, the length of the longest active attribute name, including a null terminator, is returned. If no active attributes exist, zero is returned. If *pname* is `ACTIVE_UNIFORMS`, the number of active uniforms is returned. If no active uniforms exist, zero is returned. If *pname* is `ACTIVE_UNIFORM_MAX_LENGTH`, the length of the longest active uniform name, including a null terminator, is returned. If no active uniforms exist, zero is returned. If *pname* is `TRANSFORM_FEEDBACK_BUFFER_MODE`, the buffer mode used when transform feedback is active is returned. It can be one of `SEPARATE_ATTRIBS` or `INTERLEAVED_ATTRIBS`. If *pname* is `TRANSFORM_FEEDBACK_VARYINGS`, the number of varying variables to capture in transform feedback mode for the program is returned. If *pname* is `TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH`, the length of the longest varying name specified to be used for transform feedback, including a null terminator, is returned. If no varyings are used for transform feedback, zero is returned. If *pname* is `ACTIVE_UNIFORM_BLOCKS`, the number of uniform blocks for *program* containing active uniforms is returned. If *pname* is `ACTIVE_UNIFORM_BLOCK_MAX_NAME_LENGTH`, the length of the longest active uniform block name, including the null terminator, is returned. If *pname* is `GEOMETRY_VERTICES_OUT`, the maximum number of vertices the geometry shader will output is returned. If *pname* is `GEOMETRY_INPUT_TYPE`, the geometry shader input type, which must be one of `POINTS`, `LINES`, `LINES_ADJACENCY`, `TRIANGLES` or `TRIANGLES_ADJACENCY`, is returned. If *pname* is `GEOMETRY_OUTPUT_TYPE`, the geometry shader output type, which must be one of `POINTS`, `LINE_STRIP` or `TRIANGLE_STRIP`, is returned. If `GEOMETRY_VERTICES_OUT`, `GEOMETRY_INPUT_TYPE`, or `GEOMETRY_OUTPUT_TYPE` are queried for a program which has not been linked successfully, or which does not contain objects to form a geometry shader, then an `INVALID_OPERATION` error is generated.

The command

`void` **GetAttachedShaders**(`uint` *program*, `sizei` *maxCount*,
`sizei` **count*, `uint` **shaders*);

returns the names of shader objects attached to *program* in *shaders*. The actual number of shader names written into *shaders* is returned in *count*. If no shaders are attached, *count* is set to zero. If *count* is `NULL` then it is ignored. The maximum number of shader names that may be written into *shaders* is specified by *maxCount*. The number of objects attached to *program* is given by can be queried by calling **GetProgramiv** with `ATTACHED_SHADERS`.

A string that contains information about the last compilation attempt on a shader object or last link or validation attempt on a program object, called the *info log*, can be obtained with the commands

`void` **GetShaderInfoLog**(`uint` *shader*, `sizei` *bufSize*,
`sizei` **length*, `char` **infoLog*);
`void` **GetProgramInfoLog**(`uint` *program*, `sizei` *bufSize*,
`sizei` **length*, `char` **infoLog*);

These commands return the info log string in *infoLog*. This string will be null-terminated. The actual number of characters written into *infoLog*, excluding the null terminator, is returned in *length*. If *length* is `NULL`, then no length is returned. The maximum number of characters that may be written into *infoLog*, including the null terminator, is specified by *bufSize*. The number of characters in the info log can be queried with **GetShaderiv** or **GetProgramiv** with `INFO_LOG_LENGTH`. If *shader* is a shader object, the returned info log will either be an empty string or it will contain information about the last compilation attempt for that object. If *program* is a program object, the returned info log will either be an empty string or it will contain information about the last link attempt or last validation attempt for that object.

The info log is typically only useful during application development and an application should not expect different GL implementations to produce identical info logs.

The command

`void` **GetShaderSource**(`uint` *shader*, `sizei` *bufSize*,
`sizei` **length*, `char` **source*);

returns in *source* the string making up the source code for the shader object *shader*. The string *source* will be null-terminated. The actual number of characters written into *source*, excluding the null terminator, is returned in *length*. If *length* is `NULL`, no length is returned. The maximum number of characters that may be written into *source*, including the null terminator, is specified by *bufSize*. The string *source* is a concatenation of the strings passed to the GL using **ShaderSource**. The length of this concatenation is given by `SHADER_SOURCE_LENGTH`, which can be queried with **GetShaderiv**.

The commands

`void` **GetVertexAttribdv**(`uint` *index*, `enum` *pname*,
`double` **params*);
`void` **GetVertexAttribfv**(`uint` *index*, `enum` *pname*,
`float` **params*);
`void` **GetVertexAttribiv**(`uint` *index*, `enum` *pname*,
`int` **params*);
`void` **GetVertexAttribIiv**(`uint` *index*, `enum` *pname*,
`int` **params*);
`void` **GetVertexAttribIuiv**(`uint` *index*, `enum` *pname*,
`uint` **params*);

obtain the vertex attribute state named by *pname* for the generic vertex attribute numbered *index* and places the information in the array *params*. *pname* must be one of `VERTEX_ATTRIB_ARRAY_BUFFER_BINDING`, `VERTEX_ATTRIB_ARRAY_ENABLED`, `VERTEX_ATTRIB_ARRAY_SIZE`, `VERTEX_ATTRIB_ARRAY_STRIDE`, `VERTEX_ATTRIB_ARRAY_TYPE`, `VERTEX_ATTRIB_ARRAY_NORMALIZED`, `VERTEX_ATTRIB_ARRAY_INTEGER`, or `CURRENT_VERTEX_ATTRIB`. Note that all the queries except `CURRENT_VERTEX_ATTRIB` return values stored in the currently bound vertex array object (the value of `VERTEX_ARRAY_BINDING`). 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 **VertexAttribPointer** and `TRUE` by **VertexAttribIPointer**.

The query `CURRENT_VERTEX_ATTRIB` returns the current value for the generic attribute *index*. **GetVertexAttribdv** and **GetVertexAttribfv** read and return the current attribute values as 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 command

```
void GetVertexAttribPointerv( uint index, enum pname,
   void **pointer );
```

obtains the pointer named *pname* for the vertex attribute numbered *index* and places the information in the array *pointer*. *pname* must be `VERTEX_ATTRIB_ARRAY_POINTER`. The value returned is queried from the currently bound vertex array object. 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

```
void GetUniformfv( uint program, int location,
   float *params );
```

`void` **GetUniformiv**(`uint` *program*, `int` *location*,
`int` **params*);
`void` **GetUniformuiv**(`uint` *program*, `int` *location*,
`uint` **params*);

return the value or values of the uniform at location *location* of the default uniform block for program object *program* in the array *params*. The type of the uniform at *location* determines the number of values returned. The error `INVALID_OPERATION` is generated if *program* has not been linked successfully, or if *location* is not a valid location for *program*. In order to query the values of an array of uniforms, a **GetUniform*** command needs to be issued for each array element. If the uniform queried is a matrix, the values of the matrix are returned in column major order. If an error occurred, *params* will not be modified.

6.1.17 Framebuffer Object Queries

The command

`boolean` **IsFramebuffer**(`uint` *framebuffer*);

returns `TRUE` if *framebuffer* is the name of an framebuffer object. If *framebuffer* is zero, or if *framebuffer* is a non-zero value that is not the name of an framebuffer object, **IsFramebuffer** return `FALSE`.

The command

`void` **GetFramebufferAttachmentParameteriv**(`enum` *target*,
`enum` *attachment*, `enum` *pname*, `int` **params*);

returns information about attachments of a bound framebuffer object. *target* must be `DRAW_FRAMEBUFFER`, `READ_FRAMEBUFFER`, or `FRAMEBUFFER`. `FRAMEBUFFER` is equivalent to `DRAW_FRAMEBUFFER`.

If the default framebuffer is bound to *target*, then *attachment* must be one of `FRONT_LEFT`, `FRONT_RIGHT`, `BACK_LEFT`, `BACK_RIGHT`, or `AUX`i, identifying a color buffer; `DEPTH`, identifying the depth buffer; or `STENCIL`, identifying the stencil buffer.

If a framebuffer object is bound to *target*, then *attachment* must be one of the attachment points of the framebuffer listed in table 4.12.

If *attachment* is `DEPTH_STENCIL_ATTACHMENT`, and different objects are bound to the depth and stencil attachment points of *target*, the query will fail and generate an `INVALID_OPERATION` error. If the same object is bound to both attachment points, information about that object will be returned.

Upon successful return from **GetFramebufferAttachmentParameteriv**, if *pname* is `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE`, then *param* will contain one of `NONE`, `FRAMEBUFFER_DEFAULT`, `TEXTURE`, or `RENDERBUFFER`, identifying the type of object which contains the attached image. Other values accepted for *pname* depend on the type of object, as described below.

If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is `NONE`, no framebuffer is bound to *target*. In this case querying *pname* `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` will return zero, and all other queries will generate an `INVALID_OPERATION` error.

If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is not `NONE`, these queries apply to all other framebuffer types:

- If *pname* is `FRAMEBUFFER_ATTACHMENT_RED_SIZE`, `FRAMEBUFFER_ATTACHMENT_GREEN_SIZE`, `FRAMEBUFFER_ATTACHMENT_BLUE_SIZE`, `FRAMEBUFFER_ATTACHMENT_ALPHA_SIZE`, `FRAMEBUFFER_ATTACHMENT_DEPTH_SIZE`, or `FRAMEBUFFER_ATTACHMENT_STENCIL_SIZE`, then *param* will contain the number of bits in the corresponding red, green, blue, alpha, depth, or stencil component of the specified *attachment*. Zero is returned if the requested component is not present in *attachment*.

- If *pname* is `FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE`, *param* will contain the format of components of the specified attachment, one of `FLOAT`, `INDEX`, `INT`, `UNSIGNED_INT`, `SIGNED_NORMALIZED`, or `UNSIGNED_NORMALIZED` for floating-point, index, signed integer, unsigned integer, signed normalized fixed-point, or unsigned normalized fixed-point components respectively. Only color buffers may have index or integer components.

- If *pname* is `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING`, *param* will contain the encoding of components of the specified attachment, one of `LINEAR` or `SRGB` for linear or sRGB-encoded components, respectively. Only color buffer components may be sRGB-encoded; such components are treated as described in sections 4.1.8 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 3.9.17.

If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is `RENDERBUFFER`, then

- If *pname* is `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME`, *params* will contain the name of the renderbuffer object which contains the attached image.

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 cubemap 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 layer of 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.
- If *pname* is `FRAMEBUFFER_ATTACHMENT_LAYERED`, then *params* will contain `TRUE` if an entire level of a three-dimesional texture, cube map texture, or one-or two-dimensional array texture is attached. Otherwise, *params* will contain `FALSE`.

Any combinations of framebuffer type and *pname* not described above will generate an `INVALID_ENUM` error.

6.1.18 Renderbuffer Object Queries

The command

`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`.

The command

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

returns information about a bound renderbuffer object. *target* must be `RENDERBUFFER` and *pname* must be one of the symbolic values in table 6.33. 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.

6.1.19 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

`void` **PushAttrib**(`bitfield` *mask*);
`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_STACK_DEPTH` or `MAX_CLIENT_ATTRIB_STACK_DEPTH` 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

```
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 6.3 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 6.22), 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 `TEXTURE`k 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 `TEXTURE`k 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 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 6.4 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,

Stack	Attribute	Constant
server	accum-buffer	`ACCUM_BUFFER_BIT`
server	color-buffer	`COLOR_BUFFER_BIT`
server	current	`CURRENT_BIT`
server	depth-buffer	`DEPTH_BUFFER_BIT`
server	enable	`ENABLE_BIT`
server	eval	`EVAL_BIT`
server	fog	`FOG_BIT`
server	hint	`HINT_BIT`
server	lighting	`LIGHTING_BIT`
server	line	`LINE_BIT`
server	list	`LIST_BIT`
server	multisample	`MULTISAMPLE_BIT`
server	pixel	`PIXEL_MODE_BIT`
server	point	`POINT_BIT`
server	polygon	`POLYGON_BIT`
server	polygon-stipple	`POLYGON_STIPPLE_BIT`
server	scissor	`SCISSOR_BIT`
server	stencil-buffer	`STENCIL_BUFFER_BIT`
server	texture	`TEXTURE_BIT`
server	transform	`TRANSFORM_BIT`
server	viewport	`VIEWPORT_BIT`
server		`ALL_ATTRIB_BITS`
client	vertex-array	`CLIENT_VERTEX_ARRAY_BIT`
client	pixel-store	`CLIENT_PIXEL_STORE_BIT`
client	select	can’t be pushed or popped
client	feedback	can’t be pushed or popped
client		`CLIENT_ALL_ATTRIB_BITS`

Table 6.3: Attribute groups

where only the value pertaining to the selected light is returned; with evaluator maps, where only the selected map is returned; and with textures, where only the selected texture or texture parameter 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 by using that command or any of its typed variants, although information may be lost when not using the listed command. Unless otherwise specified, when floating-point state is returned as integer values or integer state is returned as floating-point values it is converted in the fashion described in section 6.1.2.

State table entries which are required only by the imaging subset (see section 3.7.2) are typeset against a gray background .

Type code	Explanation
B	Boolean
BMU	Basic machine units
C	Color (floating-point R, G, B, and A values)
CI	Color index (floating-point index value)
T	Texture coordinates (floating-point (s, t, r, q) values)
N	Normal coordinates (floating-point (x, y, z) values)
V	Vertex, including associated data
Z	Integer
Z^+	Non-negative integer or enumerated token value
Z_k, Z_{k*}	k-valued integer ($k*$ indicates k is minimum)
R	Floating-point number
R^+	Non-negative floating-point number
$R^{[a,b]}$	Floating-point number in the range $[a, b]$
R^k	k-tuple of floating-point numbers
P	Position ((x, y, z, w) floating-point coordinates)
D	Direction ((x, y, z) floating-point coordinates)
M^4	4×4 floating-point matrix
S	`NULL`-terminated string
I	Image
A	Attribute stack entry, including mask
Y	Pointer (data type unspecified)
$n \times type$	n copies of type $type$ ($n*$ indicates n is minimum)

Table 6.4: State Variable Types

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
–	Z_{15}	–	0	When $\neq 0$, indicates **begin/end** object	2.6.1	–
–	V	–	–	Previous vertex in **Begin/End line**	2.6.1	–
–	B	–	–	Indicates if *line-vertex* is the first	2.6.1	–
–	V	–	–	First vertex of a **Begin/End line loop**	2.6.1	–
–	Z^+	–	–	Line stipple counter	3.5	–
–	$n \times V$	–	–	Vertices inside of **Begin/End polygon**	2.6.1	–
–	Z^+	–	–	Number of *polygon-vertices*	2.6.1	–
–	$2 \times V$	–	–	Previous two vertices in a **Begin/End triangle strip**	2.6.1	–
–	Z_3	–	–	Number of vertices so far in triangle strip: 0, 1, or more	2.6.1	–
–	Z_2	–	–	Triangle strip A/B vertex pointer	2.6.1	–
–	$3 \times V$	–	–	Vertices of the quad under construction	2.6.1	–
–	Z_4	–	–	Number of vertices so far in quad strip: 0, 1, 2, or more	2.6.1	–

Table 6.5. GL Internal begin-end state variables (inaccessible)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CURRENT_COLOR	C	**GetFloatv**	1,1,1,1	Current color	2.7	current
CURRENT_SECONDARY_COLOR	C	**GetFloatv**	0,0,0,1	Current secondary color	2.7	current
CURRENT_INDEX	CI	**GetIntegerv**	1	Current color index	2.7	current
CURRENT_TEXTURE_COORDS	$8 * \times T$	**GetFloatv**	0,0,0,1	Current texture coordinates	2.7	current
CURRENT_NORMAL	N	**GetFloatv**	0,0,1	Current normal	2.7	current
CURRENT_FOG_COORD	R	**GetFloatv**	0	Current fog coordinate	2.7	current
–	C	**–**	-	Color associated with last vertex	2.5	–
–	CI	**–**	-	Color index associated with last vertex	2.5	–
–	T	**–**	-	Texture coordinates associated with last vertex	2.5	–
CURRENT_RASTER_POSITION	R^4	**GetFloatv**	0,0,0,1	Current raster position	2.24	current
CURRENT_RASTER_DISTANCE	R^+	**GetFloatv**	0	Current raster distance	2.24	current
CURRENT_RASTER_COLOR	C	**GetFloatv**	1,1,1,1	Color associated with raster position	2.24	current
CURRENT_RASTER_SECONDARY_COLOR	C	**GetFloatv**	0,0,0,1	Secondary color associated with raster position	2.24	current
CURRENT_RASTER_INDEX	CI	**GetIntegerv**	1	Color index associated with raster position	2.24	current
CURRENT_RASTER_TEXTURE_COORDS	$8 * \times T$	**GetFloatv**	0,0,0,1	Texture coordinates associated with raster position	2.24	current
CURRENT_RASTER_POSITION_VALID	B	**GetBooleanv**	TRUE	Raster position valid bit	2.24	current
EDGE_FLAG	B	**GetBooleanv**	TRUE	Edge flag	2.6.2	current

Table 6.6. Current Values and Associated Data

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_ARRAY	B	**IsEnabled**	FALSE	Vertex array enable	2.8	vertex-array
VERTEX_ARRAY_SIZE	Z^+	**GetIntegerv**	4	Coordinates per vertex	2.8	vertex-array
VERTEX_ARRAY_TYPE	Z_4	**GetIntegerv**	FLOAT	Type of vertex coordinates	2.8	vertex-array
VERTEX_ARRAY_STRIDE	Z^+	**GetIntegerv**	0	Stride between vertices	2.8	vertex-array
VERTEX_ARRAY_POINTER	Y	**GetPointerv**	0	Pointer to the vertex array	2.8	vertex-array
NORMAL_ARRAY	B	**IsEnabled**	FALSE	Normal array enable	2.8	vertex-array
NORMAL_ARRAY_TYPE	Z_5	**GetIntegerv**	FLOAT	Type of normal coordinates	2.8	vertex-array
NORMAL_ARRAY_STRIDE	Z^+	**GetIntegerv**	0	Stride between normals	2.8	vertex-array
NORMAL_ARRAY_POINTER	Y	**GetPointerv**	0	Pointer to the normal array	2.8	vertex-array
FOG_COORD_ARRAY	B	**IsEnabled**	FALSE	Fog coord array enable	2.8	vertex-array
FOG_COORD_ARRAY_TYPE	Z_2	**GetIntegerv**	FLOAT	Type of fog coord components	2.8	vertex-array
FOG_COORD_ARRAY_STRIDE	Z^+	**GetIntegerv**	0	Stride between fog coords	2.8	vertex-array
FOG_COORD_ARRAY_POINTER	Y	**GetPointerv**	0	Pointer to the fog coord array	2.8	vertex-array
COLOR_ARRAY	B	**IsEnabled**	FALSE	Color array enable	2.8	vertex-array
COLOR_ARRAY_SIZE	Z_3	**GetIntegerv**	4	Color components per vertex	2.8	vertex-array
COLOR_ARRAY_TYPE	Z_8	**GetIntegerv**	FLOAT	Type of color components	2.8	vertex-array
COLOR_ARRAY_STRIDE	Z^+	**GetIntegerv**	0	Stride between colors	2.8	vertex-array
COLOR_ARRAY_POINTER	Y	**GetPointerv**	0	Pointer to the color array	2.8	vertex-array

Table 6.7. Vertex Array Object State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SECONDARY_COLOR_ARRAY	B	**IsEnabled**	FALSE	Secondary color array enable	2.8	vertex-array
SECONDARY_COLOR_ARRAY_SIZE	Z_2	**GetIntegerv**	3	Secondary color components per vertex	2.8	vertex-array
SECONDARY_COLOR_ARRAY_TYPE	Z_8	**GetIntegerv**	FLOAT	Type of secondary color components	2.8	vertex-array
SECONDARY_COLOR_ARRAY_STRIDE	Z^+	**GetIntegerv**	0	Stride between secondary colors	2.8	vertex-array
SECONDARY_COLOR_ARRAY_POINTER	Y	**GetPointerv**	0	Pointer to the secondary color array	2.8	vertex-array
INDEX_ARRAY	B	**IsEnabled**	FALSE	Index array enable	2.8	vertex-array
INDEX_ARRAY_TYPE	Z_4	**GetIntegerv**	FLOAT	Type of indices	2.8	vertex-array
INDEX_ARRAY_STRIDE	Z^+	**GetIntegerv**	0	Stride between indices	2.8	vertex-array
INDEX_ARRAY_POINTER	Y	**GetPointerv**	0	Pointer to the index array	2.8	vertex-array
TEXTURE_COORD_ARRAY	$8 * \times B$	**IsEnabled**	FALSE	Texture coordinate array enable	2.8	vertex-array
TEXTURE_COORD_ARRAY_SIZE	$8 * \times Z^+$	**GetIntegerv**	4	Coordinates per element	2.8	vertex-array
TEXTURE_COORD_ARRAY_TYPE	$8 * \times Z_4$	**GetIntegerv**	FLOAT	Type of texture coordinates	2.8	vertex-array
TEXTURE_COORD_ARRAY_STRIDE	$8 * \times Z^+$	**GetIntegerv**	0	Stride between texture coordinates	2.8	vertex-array
TEXTURE_COORD_ARRAY_POINTER	$8 * \times Y$	**GetPointerv**	0	Pointer to the texture coordinate array	2.8	vertex-array

Table 6.8. Vertex Array Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_ATTRIB_ARRAY_ENABLED	$16 * \times B$	**GetVertexAttribiv**	`FALSE`	Vertex attrib array enable	2.8	vertex-array
VERTEX_ATTRIB_ARRAY_SIZE	$16 * \times Z_5$	**GetVertexAttribiv**	4	Vertex attrib array size	2.8	vertex-array
VERTEX_ATTRIB_ARRAY_STRIDE	$16 * \times Z^+$	**GetVertexAttribiv**	0	Vertex attrib array stride	2.8	vertex-array
VERTEX_ATTRIB_ARRAY_TYPE	$16 * \times Z_9$	**GetVertexAttribiv**	`FLOAT`	Vertex attrib array type	2.8	vertex-array
VERTEX_ATTRIB_ARRAY_NORMALIZED	$16 * \times B$	**GetVertexAttribiv**	`FALSE`	Vertex attrib array normalized	2.8	vertex-array
VERTEX_ATTRIB_ARRAY_INTEGER	$16 * \times B$	**GetVertexAttribiv**	`FALSE`	Vertex attrib array has unconverted integers	2.8	vertex-array
VERTEX_ATTRIB_ARRAY_POINTER	$16 * \times Y$	**GetVertexAttribPointerv**	`NULL`	Vertex attrib array pointer	2.8	vertex-array
EDGE_FLAG_ARRAY	B	**IsEnabled**	`FALSE`	Edge flag array enable	2.8	vertex-array
EDGE_FLAG_ARRAY_STRIDE	Z^+	**GetIntegerv**	0	Stride between edge flags	2.8	vertex-array
EDGE_FLAG_ARRAY_POINTER	Y	**GetPointerv**	0	Pointer to the edge flag array	2.8	vertex-array

Table 6.9. Vertex Array Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Vertex array buffer binding	2.9	vertex-array
NORMAL_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Normal array buffer binding	2.9	vertex-array
COLOR_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Color array buffer binding	2.9	vertex-array
INDEX_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Index array buffer binding	2.9	vertex-array
TEXTURE_COORD_ARRAY_BUFFER_BINDING	$8 * \times Z^+$	**GetIntegerv**	0	Texcoord array buffer binding	2.9	vertex-array
EDGE_FLAG_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Edge flag array buffer binding	2.9	vertex-array
SECONDARY_COLOR_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Secondary color array buffer binding	2.9	vertex-array
FOG_COORD_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Fog coordinate array buffer binding	2.9	vertex-array
ELEMENT_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Element array buffer binding	2.9.7	vertex-array
VERTEX_ATTRIB_ARRAY_BUFFER_BINDING	$16 * \times Z^+$	**GetVertexAttribiv**	0	Attribute array buffer binding	2.9	vertex-array

Table 6.10. Vertex Array Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CLIENT_ACTIVE_TEXTURE	Z_{8*}	**GetIntegerv**	TEXTURE0	Client active texture unit selector	2.7	vertex-array
ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Current buffer binding	2.9	vertex-array
VERTEX_ARRAY_BINDING	Z^+	**GetIntegerv**	0	Current vertex array object binding	2.10	vertex-array
PRIMITIVE_RESTART	B	**IsEnabled**	FALSE	Primitive restart enable	2.8	vertex-array
PRIMITIVE_RESTART_INDEX	Z^+	**GetIntegerv**	0	Primitive restart index	2.8	vertex-array

Table 6.11. Vertex Array Data (not in Vertex Array objects)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
–	$n \times BMU$	**GetBufferSubData**	-	Buffer data	2.9	-
BUFFER_SIZE	$n \times Z^+$	**GetBufferParameteri64v**	0	Buffer data size	2.9	-
BUFFER_USAGE	$n \times Z_9$	**GetBufferParameteriv**	`STATIC_DRAW`	Buffer usage pattern	2.9	-
BUFFER_ACCESS	$n \times Z_3$	**GetBufferParameteriv**	`READ_WRITE`	Buffer access flag	2.9	-
BUFFER_ACCESS_FLAGS	$n \times Z^+$	**GetBufferParameteriv**	0	Extended buffer access flag	2.9	-
BUFFER_MAPPED	$n \times B$	**GetBufferParameteriv**	`FALSE`	Buffer map flag	2.9	-
BUFFER_MAP_POINTER	$n \times Y$	**GetBufferPointerv**	`NULL`	Mapped buffer pointer	2.9	-
BUFFER_MAP_OFFSET	$n \times Z^+$	**GetBufferParameteri64v**	0	Start of mapped buffer range	2.9	-
BUFFER_MAP_LENGTH	$n \times Z^+$	**GetBufferParameteri64v**	0	Size of mapped buffer range	2.9	-

Table 6.12. Buffer Object State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
COLOR_MATRIX (TRANSPOSE_COLOR_MATRIX)	$2 * \times M^4$	**GetFloatv**	Identity	Color matrix stack	3.7.3	–
MODELVIEW_MATRIX (TRANSPOSE_MODELVIEW_MATRIX)	$32 * \times M^4$	**GetFloatv**	Identity	Model-view matrix stack	2.12.1	–
PROJECTION_MATRIX (TRANSPOSE_PROJECTION_MATRIX)	$2 * \times M^4$	**GetFloatv**	Identity	Projection matrix stack	2.12.1	–
TEXTURE_MATRIX (TRANSPOSE_TEXTURE_MATRIX)	$8 * \times 2 * \times M^4$	**GetFloatv**	Identity	Texture matrix stack	2.12.1	–
VIEWPORT	$4 \times Z$	**GetIntegerv**	see 2.16.1	Viewport origin & extent	2.16.1	viewport
DEPTH_RANGE	$2 \times R^+$	**GetFloatv**	0,1	Depth range near & far	2.16.1	viewport
COLOR_MATRIX_STACK_DEPTH	Z^+	**GetIntegerv**	1	Color matrix stack pointer	3.7.3	–
MODELVIEW_STACK_DEPTH	Z^+	**GetIntegerv**	1	Model-view matrix stack pointer	2.12.1	–
PROJECTION_STACK_DEPTH	Z^+	**GetIntegerv**	1	Projection matrix stack pointer	2.12.1	–
TEXTURE_STACK_DEPTH	$8 * \times Z^+$	**GetIntegerv**	1	Texture matrix stack pointer	2.12.1	–
MATRIX_MODE	Z_4	**GetIntegerv**	`MODELVIEW`	Current matrix mode	2.12.1	transform
NORMALIZE	B	**IsEnabled**	`FALSE`	Current normal normalization on/off	2.12.2	transform/enable
RESCALE_NORMAL	B	**IsEnabled**	`FALSE`	Current normal rescaling on/off	2.12.2	transform/enable
CLIP_PLANEi	$6 * \times R^4$	**GetClipPlane**	0,0,0,0	User clipping plane coefficients	2.22	transform
CLIP_DISTANCEi	$6 * \times B$	**IsEnabled**	`FALSE`	ith user clipping plane enabled	2.22	transform/enable
DEPTH_CLAMP	B	**IsEnabled**	`FALSE`	Depth clamping enabled	2.22	transform/enable

Table 6.13. Transformation state

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
FOG_COLOR	C	**GetFloatv**	0,0,0,0	Fog color	3.11	fog
FOG_INDEX	CI	**GetFloatv**	0	Fog index	3.11	fog
FOG_DENSITY	R	**GetFloatv**	1.0	Exponential fog density	3.11	fog
FOG_START	R	**GetFloatv**	0.0	Linear fog start	3.11	fog
FOG_END	R	**GetFloatv**	1.0	Linear fog end	3.11	fog
FOG_MODE	Z_3	**GetIntegerv**	`EXP`	Fog mode	3.11	fog
FOG	B	**IsEnabled**	`FALSE`	True if fog enabled	3.11	fog/enable
FOG_COORD_SRC	Z_2	**GetIntegerv**	`FRAGMENT_DEPTH`	Source of coordinate for fog calculation	3.11	fog
COLOR_SUM	B	**IsEnabled**	`FALSE`	True if color sum enabled	3.10	fog/enable
SHADE_MODEL	Z^+	**GetIntegerv**	`SMOOTH`	**ShadeModel** setting	2.21	lighting
CLAMP_VERTEX_COLOR	Z_3	**GetIntegerv**	`TRUE`	Vertex color clamping	2.13.6	lighting/enable
CLAMP_FRAGMENT_COLOR	Z_3	**GetIntegerv**	`FIXED_ONLY`	Fragment color clamping	3.7.5	color-buffer/enable
CLAMP_READ_COLOR	Z_3	**GetIntegerv**	`FIXED_ONLY`	Read color clamping	4.3.2	color-buffer/enable
PROVOKING_VERTEX	Z_2	**GetIntegerv**	`LAST_VERTEX_CONVENTION`	Provoking vertex convention	2.21	lighting

Table 6.14. Coloring

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
LIGHTING	B	**IsEnabled**	`FALSE`	True if lighting is enabled	2.13.1	lighting/enable
COLOR_MATERIAL	B	**IsEnabled**	`FALSE`	True if color tracking is enabled	2.13.3	lighting/enable
COLOR_MATERIAL_PARAMETER	Z_5	**GetIntegerv**	`AMBIENT_AND_DIFFUSE`	Material properties tracking current color	2.13.3	lighting
COLOR_MATERIAL_FACE	Z_3	**GetIntegerv**	`FRONT_AND_BACK`	Face(s) affected by color tracking	2.13.3	lighting
AMBIENT	$2 \times C$	**GetMaterialfv**	(0.2,0.2,0.2,1.0)	Ambient material color	2.13.1	lighting
DIFFUSE	$2 \times C$	**GetMaterialfv**	(0.8,0.8,0.8,1.0)	Diffuse material color	2.13.1	lighting
SPECULAR	$2 \times C$	**GetMaterialfv**	(0.0,0.0,0.0,1.0)	Specular material color	2.13.1	lighting
EMISSION	$2 \times C$	**GetMaterialfv**	(0.0,0.0,0.0,1.0)	Emissive mat. color	2.13.1	lighting
SHININESS	$2 \times R$	**GetMaterialfv**	0.0	Specular exponent of material	2.13.1	lighting
LIGHT_MODEL_AMBIENT	C	**GetFloatv**	(0.2,0.2,0.2,1.0)	Ambient scene color	2.13.1	lighting
LIGHT_MODEL_LOCAL_VIEWER	B	**GetBooleanv**	`FALSE`	Viewer is local	2.13.1	lighting
LIGHT_MODEL_TWO_SIDE	B	**GetBooleanv**	`FALSE`	Use two-sided lighting	2.13.1	lighting
LIGHT_MODEL_COLOR_CONTROL	Z_2	**GetIntegerv**	`SINGLE_COLOR`	Color control	2.13.1	lighting

Table 6.15. Lighting (see also table 2.11 for defaults)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
AMBIENT	$8 * \times C$	**GetLightfv**	(0.0,0.0,0.0,1.0)	Ambient intensity of light i	2.13.1	lighting
DIFFUSE	$8 * \times C$	**GetLightfv**	see table 2.11	Diffuse intensity of light i	2.13.1	lighting
SPECULAR	$8 * \times C$	**GetLightfv**	see table 2.11	Specular intensity of light i	2.13.1	lighting
POSITION	$8 * \times P$	**GetLightfv**	(0.0,0.0,1.0,0.0)	Position of light i	2.13.1	lighting
CONSTANT_ATTENUATION	$8 * \times R^+$	**GetLightfv**	1.0	Constant atten. factor	2.13.1	lighting
LINEAR_ATTENUATION	$8 * \times R^+$	**GetLightfv**	0.0	Linear atten. factor	2.13.1	lighting
QUADRATIC_ATTENUATION	$8 * \times R^+$	**GetLightfv**	0.0	Quadratic atten. factor	2.13.1	lighting
SPOT_DIRECTION	$8 * \times D$	**GetLightfv**	(0.0,0.0,-1.0)	Spotlight direction of light i	2.13.1	lighting
SPOT_EXPONENT	$8 * \times R^+$	**GetLightfv**	0.0	Spotlight exponent of light i	2.13.1	lighting
SPOT_CUTOFF	$8 * \times R^+$	**GetLightfv**	180.0	Spot. angle of light i	2.13.1	lighting
LIGHTi	$8 * \times B$	**IsEnabled**	`FALSE`	True if light i enabled	2.13.1	lighting/enable
COLOR_INDEXES	$2 \times 3 \times R$	**GetMaterialfv**	0,1,1	a_m, d_m, and s_m for color index lighting	2.13.1	lighting

Table 6.16. Lighting (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
POINT_SIZE	R^+	**GetFloatv**	1.0	Point size	3.4	point
POINT_SMOOTH	B	**IsEnabled**	`FALSE`	Point antialiasing on	3.4	point/enable
POINT_SPRITE	B	**IsEnabled**	`FALSE`	Point sprite enable	3.4	point/enable
POINT_SIZE_MIN	R^+	**GetFloatv**	0.0	Attenuated minimum point size	3.4	point
POINT_SIZE_MAX	R^+	**GetFloatv**	[1]	Attenuated maximum point size. [1] Max. of the impl. dependent max. aliased and smooth point sizes.	3.4	point
POINT_FADE_THRESHOLD_SIZE	R^+	**GetFloatv**	1.0	Threshold for alpha attenuation	3.4	point
POINT_DISTANCE_ATTENUATION	$3 \times R^+$	**GetFloatv**	1,0,0	Attenuation coefficients	3.4	point
POINT_SPRITE_COORD_ORIGIN	Z_2	**GetIntegerv**	`UPPER_LEFT`	Origin orientation for point sprites	3.4	point
LINE_WIDTH	R^+	**GetFloatv**	1.0	Line width	3.5	line
LINE_SMOOTH	B	**IsEnabled**	`FALSE`	Line antialiasing on	3.5	line/enable
LINE_STIPPLE_PATTERN	Z^+	**GetIntegerv**	1's	Line stipple	3.5.2	line
LINE_STIPPLE_REPEAT	Z^+	**GetIntegerv**	1	Line stipple repeat	3.5.2	line
LINE_STIPPLE	B	**IsEnabled**	`FALSE`	Line stipple enable	3.5.2	line/enable

Table 6.17. Rasterization

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CULL_FACE	B	**IsEnabled**	`FALSE`	Polygon culling enabled	3.6.1	polygon/enable
CULL_FACE_MODE	Z_3	**GetIntegerv**	`BACK`	Cull front-/back-facing polygons	3.6.1	polygon
FRONT_FACE	Z_2	**GetIntegerv**	`CCW`	Polygon frontface CW/CCW indicator	3.6.1	polygon
POLYGON_SMOOTH	B	**IsEnabled**	`FALSE`	Polygon antialiasing on	3.6	polygon/enable
POLYGON_MODE	$2 \times Z_3$	**GetIntegerv**	`FILL`	Polygon rasterization mode (front & back)	3.6.4	polygon
POLYGON_OFFSET_FACTOR	R	**GetFloatv**	0	Polygon offset factor	3.6.5	polygon
POLYGON_OFFSET_UNITS	R	**GetFloatv**	0	Polygon offset units	3.6.5	polygon
POLYGON_OFFSET_POINT	B	**IsEnabled**	`FALSE`	Polygon offset enable for `POINT` mode rasterization	3.6.5	polygon/enable
POLYGON_OFFSET_LINE	B	**IsEnabled**	`FALSE`	Polygon offset enable for `LINE` mode rasterization	3.6.5	polygon/enable
POLYGON_OFFSET_FILL	B	**IsEnabled**	`FALSE`	Polygon offset enable for `FILL` mode rasterization	3.6.5	polygon/enable
–	I	**GetPolygonStipple**	1's	Polygon stipple	3.6	polygon-stipple
POLYGON_STIPPLE	B	**IsEnabled**	`FALSE`	Polygon stipple enable	3.6.2	polygon/enable

Table 6.18. Rasterization (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
MULTISAMPLE	B	**IsEnabled**	`TRUE`	Multisample rasterization	3.3.1	multisample/enable
SAMPLE_ALPHA_TO_COVERAGE	B	**IsEnabled**	`FALSE`	Modify coverage from alpha	4.1.3	multisample/enable
SAMPLE_ALPHA_TO_ONE	B	**IsEnabled**	`FALSE`	Set alpha to maximum	4.1.3	multisample/enable
SAMPLE_COVERAGE	B	**IsEnabled**	`FALSE`	Mask to modify coverage	4.1.3	multisample/enable
SAMPLE_COVERAGE_VALUE	R^+	**GetFloatv**	1	Coverage mask value	4.1.3	multisample
SAMPLE_COVERAGE_INVERT	B	**GetBooleanv**	`FALSE`	Invert coverage mask value	4.1.3	multisample
SAMPLE_MASK	B	**IsEnabled**	`FALSE`	Sample mask enable	4.1.3	–
SAMPLE_MASK_VALUE	$n \times Z^+$	**GetIntegeri_v**	$n\times$ 1's	Sample mask words	4.1.3	–

Table 6.19. Multisampling

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_xD	$8 * \times 3 \times B$	**IsEnabled**	`FALSE`	True if xD texturing is enabled; x is 1, 2, or 3	3.9.19	texture/enable
TEXTURE_CUBE_MAP	$8 * \times B$	**IsEnabled**	`FALSE`	True if cube map texturing is enabled	3.9.15	texture/enable
TEXTURE_BINDING_xD	$48 * \times 3 \times Z^+$	**GetIntegerv**	0	Texture object bound to `TEXTURE_`x`D`	3.9.14	texture
TEXTURE_BINDING_1D_ARRAY	$48 * \times Z^+$	**GetIntegerv**	0	Texture object bound to `TEXTURE_1D_ARRAY`	3.9.14	texture
TEXTURE_BINDING_2D_ARRAY	$48 * \times Z^+$	**GetIntegerv**	0	Texture object bound to `TEXTURE_2D_ARRAY`	3.9.14	texture

Table 6.20. Textures (state per texture unit and binding point)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_BINDING_RECTANGLE	$48 * \times Z^+$	**GetIntegerv**	0	Texture object bound to `TEXTURE_RECTANGLE`	3.9.14	texture
TEXTURE_BINDING_BUFFER	$48 * \times Z^+$	**GetIntegerv**	0	Texture object bound to `TEXTURE_BUFFER`	3.9.14	texture
TEXTURE_BINDING_CUBE_MAP	$48 * \times Z^+$	**GetIntegerv**	0	Texture object bound to `TEXTURE_CUBE_MAP`	3.9.14	texture
TEXTURE_BINDING_2D_MULTISAMPLE	$48 * \times Z^+$	**GetIntegerv**	0	Texture object bound to `TEXTURE_2D_MULTISAMPLE`	3.9.13	–
TEXTURE_BINDING_2D_MULTISAMPLE_ARRAY	$48 * \times Z^+$	**GetIntegerv**	0	Texture object bound to `TEXTURE_2D_MULTISAMPLE_ARRAY`	3.9.13	–
TEXTURE_xD	$n \times I$	**GetTexImage**	see 3.9	xD texture image at l.o.d. i	3.9	–
TEXTURE_1D_ARRAY	$n \times I$	**GetTexImage**	see 3.9	1D texture array image at row i	3.9	–
TEXTURE_2D_ARRAY	$n \times I$	**GetTexImage**	see 3.9	2D texture array image at slice i	3.9	–
TEXTURE_RECTANGLE	$n \times I$	**GetTexImage**	see 3.9	Rectangular texture image at l.o.d. zero	3.9	–
TEXTURE_CUBE_MAP_POSITIVE_X	$n \times I$	**GetTexImage**	see 3.9.1	$+x$ face cube map texture image at l.o.d. i	3.9.1	–
TEXTURE_CUBE_MAP_NEGATIVE_X	$n \times I$	**GetTexImage**	see 3.9.1	$-x$ face cube map texture image at l.o.d. i	3.9.1	–
TEXTURE_CUBE_MAP_POSITIVE_Y	$n \times I$	**GetTexImage**	see 3.9.1	$+y$ face cube map texture image at l.o.d. i	3.9.1	–
TEXTURE_CUBE_MAP_NEGATIVE_Y	$n \times I$	**GetTexImage**	see 3.9.1	$-y$ face cube map texture image at l.o.d. i	3.9.1	–
TEXTURE_CUBE_MAP_POSITIVE_Z	$n \times I$	**GetTexImage**	see 3.9.1	$+z$ face cube map texture image at l.o.d. i	3.9.1	–
TEXTURE_CUBE_MAP_NEGATIVE_Z	$n \times I$	**GetTexImage**	see 3.9.1	$-z$ face cube map texture image at l.o.d. i	3.9.1	–

Table 6.21. Textures (state per texture unit and binding point)(cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_BORDER_COLOR	$n \times C$	**GetTexParameter**	0,0,0,0	Border color	3.9	texture
TEXTURE_MIN_FILTER	$n \times Z_6$	**GetTexParameter**	see sec. 3.9.13	Minification function	3.9.9	texture
TEXTURE_MAG_FILTER	$n \times Z_2$	**GetTexParameter**	`LINEAR`	Magnification function	3.9.10	texture
TEXTURE_WRAP_S	$n \times Z_5$	**GetTexParameter**	see sec. 3.9.13	Texcoord s wrap mode	3.9.9	texture
TEXTURE_WRAP_T	$n \times Z_5$	**GetTexParameter**	see sec. 3.9.13	Texcoord t wrap mode (2D, 3D, cube map textures only)	3.9.9	texture
TEXTURE_WRAP_R	$n \times Z_5$	**GetTexParameter**	see sec. 3.9.13	Texcoord r wrap mode (3D textures only)	3.9.9	texture
TEXTURE_PRIORITY	$n \times R^{[0,1]}$	**GetTexParameterfv**	1	Texture object priority	3.9.14	texture
TEXTURE_RESIDENT	$n \times B$	**GetTexParameteriv**	see 3.9.14	Texture residency	3.9.14	texture
TEXTURE_MIN_LOD	$n \times R$	**GetTexParameterfv**	-1000	Minimum level of detail	3.9	texture
TEXTURE_MAX_LOD	$n \times R$	**GetTexParameterfv**	1000	Maximum level of detail	3.9	texture
TEXTURE_BASE_LEVEL	$n \times Z^+$	**GetTexParameterfv**	0	Base texture array	3.9	texture
TEXTURE_MAX_LEVEL	$n \times Z^+$	**GetTexParameterfv**	1000	Max. texture array level	3.9	texture
TEXTURE_LOD_BIAS	$n \times R$	**GetTexParameterfv**	0.0	Texture level of detail bias ($bias_{texobj}$)	3.9.9	texture
DEPTH_TEXTURE_MODE	$n \times Z_3$	**GetTexParameteriv**	`LUMINANCE`	Depth texture mode	3.9.7	texture
TEXTURE_COMPARE_MODE	$n \times Z_2$	**GetTexParameteriv**	`NONE`	Comparison mode	3.9.16	texture
TEXTURE_COMPARE_FUNC	$n \times Z_8$	**GetTexParameteriv**	`LEQUAL`	Comparison function	3.9.16	texture
GENERATE_MIPMAP	$n \times B$	**GetTexParameter**	`FALSE`	Automatic mipmap generation enabled	3.9.9	texture

Table 6.22. Textures (state per texture object)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_WIDTH	$n \times Z^+$	**GetTexLevelParameter**	0	Specified width	3.9	–
TEXTURE_HEIGHT	$n \times Z^+$	**GetTexLevelParameter**	0	Specified height (2D/3D)	3.9	–
TEXTURE_DEPTH	$n \times Z^+$	**GetTexLevelParameter**	0	Specified depth (3D)	3.9	–
TEXTURE_BORDER	$n \times Z^+$	**GetTexLevelParameter**	0	Specified border width	3.9	–
TEXTURE_SAMPLES	Z^+	**GetTexLevelParameter**	0	Number of samples per texel	3.9.4	–
TEXTURE_FIXED_SAMPLE_LOCATIONS	B	**GetTexLevelParameter**	`TRUE`	Whether the image uses a fixed sample pattern	3.9.4	–
TEXTURE_INTERNAL_FORMAT (TEXTURE_COMPONENTS)	$n \times Z_{68*}$	**GetTexLevelParameter**	1 or `R8`	Internal format (see section 3.9.13	3.9	–
TEXTURE_x_SIZE	$n \times 8 \times Z^+$	**GetTexLevelParameter**	0	Component resolution (x is `RED`, `GREEN`, `BLUE`, `ALPHA`, `LUMINANCE`, `INTENSITY`, `DEPTH`, or `STENCIL`)	3.9	–
TEXTURE_SHARED_SIZE	$n \times Z^+$	**GetTexLevelParameter**	0	Shared exponent field resolution	3.9	–
TEXTURE_x_TYPE	$n \times Z_5$	**GetTexLevelParameter**	`NONE`	Component type (x is `RED`, `GREEN`, `BLUE`, `ALPHA`, `LUMINANCE`, `INTENSITY`, or `DEPTH`)	6.1.3	–
TEXTURE_COMPRESSED	$n \times B$	**GetTexLevelParameter**	`FALSE`	True if image has a compressed internal format	3.9.3	-
TEXTURE_COMPRESSED_IMAGE_SIZE	$n \times Z^+$	**GetTexLevelParameter**	0	Size (in `ubytes`) of compressed image	3.9.3	-
TEXTURE_BUFFER_DATA_STORE_BINDING	$n \times Z^+$	**GetTexLevelParameter**	0	Buffer object bound as the data store for the active image unit's buffer texture	3.9.14	texture

Table 6.23. Textures (state per texture image)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
ACTIVE_TEXTURE	Z_{48*}	**GetIntegerv**	`TEXTURE0`	Active texture unit selector	2.7	texture
COORD_REPLACE	$2*\times B$	**GetTexEnviv**	`FALSE`	Coordinate replacement enable	3.4	point
TEXTURE_ENV_MODE	$2*\times Z_6$	**GetTexEnviv**	`MODULATE`	Texture application function	3.9.15	texture
TEXTURE_ENV_COLOR	$2*\times C$	**GetTexEnvfv**	0,0,0,0	Texture environment color	3.9.15	texture
TEXTURE_LOD_BIAS	$2*\times R$	**GetTexEnvfv**	0.0	Texture level of detail bias $bias_{texunit}$	3.9.9	texture
TEXTURE_GEN_x	$2*\times 4\times B$	**IsEnabled**	`FALSE`	Texgen enabled (x is S, T, R, or Q)	2.12.3	texture/enable
EYE_PLANE	$2*\times 4\times R^4$	**GetTexGenfv**	see 2.12.3	Texgen plane equation coefficients (for S, T, R, and Q)	2.12.3	texture
OBJECT_PLANE	$2*\times 4\times R^4$	**GetTexGenfv**	see 2.12.3	Texgen object linear coefficients (for S, T, R, and Q)	2.12.3	texture
TEXTURE_GEN_MODE	$2*\times 4\times Z_5$	**GetTexGeniv**	`EYE_LINEAR`	Function used for texgen (for S, T, R, and Q	2.12.3	texture
COMBINE_RGB	$2*\times Z_8$	**GetTexEnviv**	`MODULATE`	RGB combiner function	3.9.15	texture
COMBINE_ALPHA	$2*\times Z_6$	**GetTexEnviv**	`MODULATE`	Alpha combiner function	3.9.15	texture

Table 6.24. Texture Environment and Generation

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SRC0_RGB	$2 * \times Z_3$	**GetTexEnviv**	TEXTURE	RGB source 0	3.9.15	texture
SRC1_RGB	$2 * \times Z_3$	**GetTexEnviv**	PREVIOUS	RGB source 1	3.9.15	texture
SRC2_RGB	$2 * \times Z_3$	**GetTexEnviv**	CONSTANT	RGB source 2	3.9.15	texture
SRC0_ALPHA	$2 * \times Z_3$	**GetTexEnviv**	TEXTURE	Alpha source 0	3.9.15	texture
SRC1_ALPHA	$2 * \times Z_3$	**GetTexEnviv**	PREVIOUS	Alpha source 1	3.9.15	texture
SRC2_ALPHA	$2 * \times Z_3$	**GetTexEnviv**	CONSTANT	Alpha source 2	3.9.15	texture
OPERAND0_RGB	$2 * \times Z_4$	**GetTexEnviv**	SRC_COLOR	RGB operand 0	3.9.15	texture
OPERAND1_RGB	$2 * \times Z_4$	**GetTexEnviv**	SRC_COLOR	RGB operand 1	3.9.15	texture
OPERAND2_RGB	$2 * \times Z_4$	**GetTexEnviv**	SRC_ALPHA	RGB operand 2	3.9.15	texture
OPERAND0_ALPHA	$2 * \times Z_2$	**GetTexEnviv**	SRC_ALPHA	Alpha operand 0	3.9.15	texture
OPERAND1_ALPHA	$2 * \times Z_2$	**GetTexEnviv**	SRC_ALPHA	Alpha operand 1	3.9.15	texture
OPERAND2_ALPHA	$2 * \times Z_2$	**GetTexEnviv**	SRC_ALPHA	Alpha operand 2	3.9.15	texture
RGB_SCALE	$2 * \times R^3$	**GetTexEnvfv**	1.0	RGB post-combiner scaling	3.9.15	texture
ALPHA_SCALE	$2 * \times R^3$	**GetTexEnvfv**	1.0	Alpha post-combiner scaling	3.9.15	texture

Table 6.25. Texture Environment and Generation (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SCISSOR_TEST	B	**IsEnabled**	`FALSE`	Scissoring enabled	4.1.2	scissor/enable
SCISSOR_BOX	$4 \times Z$	**GetIntegerv**	see 4.1.2	Scissor box	4.1.2	scissor
ALPHA_TEST	B	**IsEnabled**	`FALSE`	Alpha test enabled	4.1.4	color-buffer/enable
ALPHA_TEST_FUNC	Z_8	**GetIntegerv**	`ALWAYS`	Alpha test function	4.1.4	color-buffer
ALPHA_TEST_REF	R^+	**GetIntegerv**	0	Alpha test reference value	4.1.4	color-buffer
STENCIL_TEST	B	**IsEnabled**	`FALSE`	Stenciling enabled	4.1.5	stencil-buffer/enable
STENCIL_FUNC	Z_8	**GetIntegerv**	`ALWAYS`	Front stencil function	4.1.5	stencil-buffer
STENCIL_VALUE_MASK	Z^+	**GetIntegerv**	see 4.1.5	Front stencil mask	4.1.5	stencil-buffer
STENCIL_REF	Z^+	**GetIntegerv**	0	Front stencil reference value	4.1.5	stencil-buffer
STENCIL_FAIL	Z_8	**GetIntegerv**	`KEEP`	Front stencil fail action	4.1.5	stencil-buffer
STENCIL_PASS_DEPTH_FAIL	Z_8	**GetIntegerv**	`KEEP`	Front stencil depth buffer fail action	4.1.5	stencil-buffer
STENCIL_PASS_DEPTH_PASS	Z_8	**GetIntegerv**	`KEEP`	Front stencil depth buffer pass action	4.1.5	stencil-buffer
STENCIL_BACK_FUNC	Z_8	**GetIntegerv**	`ALWAYS`	Back stencil function	4.1.5	stencil-buffer
STENCIL_BACK_VALUE_MASK	Z^+	**GetIntegerv**	see 4.1.5	Back stencil mask	4.1.5	stencil-buffer
STENCIL_BACK_REF	Z^+	**GetIntegerv**	0	Back stencil reference value	4.1.5	stencil-buffer
STENCIL_BACK_FAIL	Z_8	**GetIntegerv**	`KEEP`	Back stencil fail action	4.1.5	stencil-buffer
STENCIL_BACK_PASS_DEPTH_FAIL	Z_8	**GetIntegerv**	`KEEP`	Back stencil depth buffer fail action	4.1.5	stencil-buffer
STENCIL_BACK_PASS_DEPTH_PASS	Z_8	**GetIntegerv**	`KEEP`	Back stencil depth buffer pass action	4.1.5	stencil-buffer

Table 6.26. Pixel Operations

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
DEPTH_TEST	B	**IsEnabled**	`FALSE`	Depth buffer enabled	4.1.6	depth-buffer/enable
DEPTH_FUNC	Z_8	**GetIntegerv**	`LESS`	Depth buffer test function	4.1.6	depth-buffer
BLEND	$1 * \times B$	**IsEnabledi**	`FALSE`	Blending enabled for draw buffer i	4.1.8	color-buffer/enable
BLEND_SRC_RGB (v1.3:BLEND_SRC)	Z_{15}	**GetIntegerv**	`ONE`	Blending source RGB function	4.1.8	color-buffer
BLEND_SRC_ALPHA	Z_{15}	**GetIntegerv**	`ONE`	Blending source A function	4.1.8	color-buffer
BLEND_DST_RGB (v1.3:BLEND_DST)	Z_{14}	**GetIntegerv**	`ZERO`	Blending dest. RGB function	4.1.8	color-buffer
BLEND_DST_ALPHA	Z_{14}	**GetIntegerv**	`ZERO`	Blending dest. A function	4.1.8	color-buffer
BLEND_EQUATION_RGB (v1.5: BLEND_EQUATION)	Z_5	**GetIntegerv**	`FUNC_ADD`	RGB blending equation	4.1.8	color-buffer
BLEND_EQUATION_ALPHA	Z_5	**GetIntegerv**	`FUNC_ADD`	Alpha blending equation	4.1.8	color-buffer
BLEND_COLOR	C	**GetFloatv**	0,0,0,0	Constant blend color	4.1.8	color-buffer
FRAMEBUFFER_SRGB	B	**IsEnabled**	`FALSE`	sRGB update and blending enable	4.1.8	color-buffer/enable
DITHER	B	**IsEnabled**	`TRUE`	Dithering enabled	4.1.10	color-buffer/enable
INDEX_LOGIC_OP (v1.0:LOGIC_OP)	B	**IsEnabled**	`FALSE`	Index logic op enabled	4.1.11	color-buffer/enable
COLOR_LOGIC_OP	B	**IsEnabled**	`FALSE`	Color logic op enabled	4.1.11	color-buffer/enable
LOGIC_OP_MODE	Z_{16}	**GetIntegerv**	`COPY`	Logic op function	4.1.11	color-buffer

Table 6.27. Pixel Operations (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
INDEX_WRITEMASK	Z^+	**GetIntegerv**	1's	Color index writemask	4.2.2	color-buffer
COLOR_WRITEMASK	$1 * \times 4 \times B$	**GetBooleani_v**	(TRUE,TRUE,TRUE,TRUE)	Color write enables (R,G,B,A) for draw buffer i	4.2.2	color-buffer
DEPTH_WRITEMASK	B	**GetBooleanv**	TRUE	Depth buffer enabled for writing	4.2.2	depth-buffer
STENCIL_WRITEMASK	Z^+	**GetIntegerv**	1's	Front stencil buffer writemask	4.2.2	stencil-buffer
STENCIL_BACK_WRITEMASK	Z^+	**GetIntegerv**	1's	Back stencil buffer writemask	4.2.2	stencil-buffer
COLOR_CLEAR_VALUE	C	**GetFloatv**	0,0,0,0	Color buffer clear value (RGBA mode)	4.2.3	color-buffer
INDEX_CLEAR_VALUE	CI	**GetFloatv**	0	Color buffer clear value (color index mode)	4.2.3	color-buffer
DEPTH_CLEAR_VALUE	R^+	**GetIntegerv**	1	Depth buffer clear value	4.2.3	depth-buffer
STENCIL_CLEAR_VALUE	Z^+	**GetIntegerv**	0	Stencil clear value	4.2.3	stencil-buffer
ACCUM_CLEAR_VALUE	$4 \times R^+$	**GetFloatv**	0	Accumulation buffer clear value	4.2.3	accum-buffer

Table 6.28. Framebuffer Control

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
DRAW_FRAMEBUFFER_BINDING	Z^+	**GetIntegerv**	0	Framebuffer object bound to `DRAW_FRAMEBUFFER`	4.4.1	–
READ_FRAMEBUFFER_BINDING	Z^+	**GetIntegerv**	0	Framebuffer object bound to `READ_FRAMEBUFFER`	4.4.1	–

Table 6.29. Framebuffer (state per target binding point)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
DRAW_BUFFERi	$1 * \times Z_{11*}$	**GetIntegerv**	see 4.2.1	Draw buffer selected for color output i	4.2.1	color-buffer
READ_BUFFER	Z_{11*}	**GetIntegerv**	see 4.3.2	Read source buffer	4.3.2	pixel

Table 6.30. Framebuffer (state per framebuffer object)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE	Z	**GetFramebufferAttachmentParameteriv**	NONE	Type of image attached to framebuffer attachment point	4.4.2	–
FRAMEBUFFER_ATTACHMENT_OBJECT_NAME	Z	**GetFramebufferAttachmentParameteriv**	0	Name of object attached to framebuffer attachment point	4.4.2	–
FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL	Z	**GetFramebufferAttachmentParameteriv**	0	Mipmap level of texture image attached, if object attached is texture	4.4.2	–
FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE	Z^+	**GetFramebufferAttachmentParameteriv**	NONE	Cubemap face of texture image attached, if object attached is cubemap texture	4.4.2	–
FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER	Z	**GetFramebufferAttachmentParameteriv**	0	Layer of texture image attached, if object attached is 3D texture	4.4.2	–
FRAMEBUFFER_ATTACHMENT_LAYERED	$n \times B$	**GetFramebufferAttachmentParameteriv**	FALSE	Framebuffer attachment is layered	4.4.7	–
FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING	Z_2	**GetFramebufferAttachmentParameteriv**	-	Encoding of components in the attached image	6.1.3	–
FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE	Z_4	**GetFramebufferAttachmentParameteriv**	-	Data type of components in the attached image	6.1.3	–
FRAMEBUFFER_ATTACHMENT_x_SIZE	Z^+	**GetFramebufferAttachmentParameteriv**	-	Size in bits of attached image's x component; x is RED, GREEN, BLUE, ALPHA, DEPTH, or STENCIL	6.1.3	–

Table 6.31. Framebuffer (state per attachment point)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
RENDERBUFFER_BINDING	Z	**GetIntegerv**	0	Renderbuffer object bound to `RENDERBUFFER`	4.4.2	–

Table 6.32. Renderbuffer (state per target and binding point)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
RENDERBUFFER_WIDTH	Z^+	**GetRenderbufferParameteriv**	0	Width of renderbuffer	4.4.2	–
RENDERBUFFER_HEIGHT	Z^+	**GetRenderbufferParameteriv**	0	Height of renderbuffer	4.4.2	–
RENDERBUFFER_INTERNAL_FORMAT	Z^+	**GetRenderbufferParameteriv**	RGBA	Internal format of renderbuffer	4.4.2	–
RENDERBUFFER_RED_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's red component	4.4.2	–
RENDERBUFFER_GREEN_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's green component	4.4.2	–
RENDERBUFFER_BLUE_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's blue component	4.4.2	–
RENDERBUFFER_ALPHA_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's alpha component	4.4.2	–
RENDERBUFFER_DEPTH_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's depth component	4.4.2	–
RENDERBUFFER_STENCIL_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Size in bits of renderbuffer image's stencil component	4.4.2	–
RENDERBUFFER_SAMPLES	Z^+	**GetRenderbufferParameteriv**	0	Number of samples	4.4.2	–

Table 6.33. Renderbuffer (state per renderbuffer object)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
UNPACK_SWAP_BYTES	B	**GetBooleanv**	`FALSE`	Value of `UNPACK_SWAP_BYTES`	3.7.1	pixel-store
UNPACK_LSB_FIRST	B	**GetBooleanv**	`FALSE`	Value of `UNPACK_LSB_FIRST`	3.7.1	pixel-store
UNPACK_IMAGE_HEIGHT	Z^+	**GetIntegerv**	0	Value of `UNPACK_IMAGE_HEIGHT`	3.7.1	pixel-store
UNPACK_SKIP_IMAGES	Z^+	**GetIntegerv**	0	Value of `UNPACK_SKIP_IMAGES`	3.7.1	pixel-store
UNPACK_ROW_LENGTH	Z^+	**GetIntegerv**	0	Value of `UNPACK_ROW_LENGTH`	3.7.1	pixel-store
UNPACK_SKIP_ROWS	Z^+	**GetIntegerv**	0	Value of `UNPACK_SKIP_ROWS`	3.7.1	pixel-store
UNPACK_SKIP_PIXELS	Z^+	**GetIntegerv**	0	Value of `UNPACK_SKIP_PIXELS`	3.7.1	pixel-store
UNPACK_ALIGNMENT	Z^+	**GetIntegerv**	4	Value of `UNPACK_ALIGNMENT`	3.7.1	pixel-store
PACK_SWAP_BYTES	B	**GetBooleanv**	`FALSE`	Value of `PACK_SWAP_BYTES`	4.3.2	pixel-store
PACK_LSB_FIRST	B	**GetBooleanv**	`FALSE`	Value of `PACK_LSB_FIRST`	4.3.2	pixel-store
PACK_IMAGE_HEIGHT	Z^+	**GetIntegerv**	0	Value of `PACK_IMAGE_HEIGHT`	4.3.2	pixel-store
PACK_SKIP_IMAGES	Z^+	**GetIntegerv**	0	Value of `PACK_SKIP_IMAGES`	4.3.2	pixel-store
PACK_ROW_LENGTH	Z^+	**GetIntegerv**	0	Value of `PACK_ROW_LENGTH`	4.3.2	pixel-store
PACK_SKIP_ROWS	Z^+	**GetIntegerv**	0	Value of `PACK_SKIP_ROWS`	4.3.2	pixel-store
PACK_SKIP_PIXELS	Z^+	**GetIntegerv**	0	Value of `PACK_SKIP_PIXELS`	4.3.2	pixel-store
PACK_ALIGNMENT	Z^+	**GetIntegerv**	4	Value of `PACK_ALIGNMENT`	4.3.2	pixel-store
PIXEL_PACK_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Pixel pack buffer binding	4.3.2	pixel-store
PIXEL_UNPACK_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Pixel unpack buffer binding	6.1.14	pixel-store

Table 6.34. Pixels

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
MAP_COLOR	B	**GetBooleanv**	`FALSE`	True if colors are mapped	3.7.3	pixel
MAP_STENCIL	B	**GetBooleanv**	`FALSE`	True if stencil values are mapped	3.7.3	pixel
INDEX_SHIFT	Z	**GetIntegerv**	0	Value of `INDEX_SHIFT`	3.7.3	pixel
INDEX_OFFSET	Z	**GetIntegerv**	0	Value of `INDEX_OFFSET`	3.7.3	pixel
x_SCALE	R	**GetFloatv**	1	Value of x`_SCALE`; x is `RED`, `GREEN`, `BLUE`, `ALPHA`, or `DEPTH`	3.7.3	pixel
x_BIAS	R	**GetFloatv**	0	Value of x`_BIAS`	3.7.3	pixel

Table 6.35. Pixels (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
COLOR_TABLE	B	**IsEnabled**	`FALSE`	True if color table lookup is done	3.7.3	pixel/enable
POST_CONVOLUTION_COLOR_TABLE	B	**IsEnabled**	`FALSE`	True if post convolution color table lookup is done	3.7.3	pixel/enable
POST_COLOR_MATRIX_COLOR_TABLE	B	**IsEnabled**	`FALSE`	True if post color matrix color table lookup is done	3.7.3	pixel/enable
COLOR_TABLE	I	**GetColorTable**	*empty*	Color table	3.7.3	–
POST_CONVOLUTION_COLOR_TABLE	I	**GetColorTable**	*empty*	Post convolution color table	3.7.3	–
POST_COLOR_MATRIX_COLOR_TABLE	I	**GetColorTable**	*empty*	Post color matrix color table	3.7.3	–
COLOR_TABLE_FORMAT	$2 \times 3 \times Z_{42}$	**GetColorTableParameteriv**	`RGBA`	Color tables' internal image format	3.7.3	–
COLOR_TABLE_WIDTH	$2 \times 3 \times Z^+$	**GetColorTableParameteriv**	0	Color tables' specified width	3.7.3	–
COLOR_TABLE_x_SIZE	$6 \times 2 \times 3 \times Z^+$	**GetColorTableParameteriv**	0	Color table component resolution; x is `RED`, `GREEN`, `BLUE`, `ALPHA`, `LUMINANCE`, or `INTENSITY`	3.7.3	–
COLOR_TABLE_SCALE	$3 \times R^4$	**GetColorTableParameterfv**	1,1,1,1	Scale factors applied to color table entries	3.7.3	pixel
COLOR_TABLE_BIAS	$3 \times R^4$	**GetColorTableParameterfv**	0,0,0,0	Bias factors applied to color table entries	3.7.3	pixel

Table 6.36. Pixels (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CONVOLUTION_1D	B	**IsEnabled**	`FALSE`	True if 1D convolution is done	3.7.3	pixel/enable
CONVOLUTION_2D	B	**IsEnabled**	`FALSE`	True if 2D convolution is done	3.7.3	pixel/enable
SEPARABLE_2D	B	**IsEnabled**	`FALSE`	True if separable 2D convolution is done	3.7.3	pixel/enable
CONVOLUTION_xD	$2 \times I$	**GetConvolutionFilter**	*empty*	Convolution filters; x is 1 or 2	3.7.3	–
SEPARABLE_2D	$2 \times I$	**GetSeparable- Filter**	*empty*	Separable convolution filter	3.7.3	–
CONVOLUTION_BORDER_COLOR	$3 \times C$	**GetConvolutionParameterfv**	0,0,0,0	Convolution border color	3.7.6	pixel
CONVOLUTION_BORDER_MODE	$3 \times Z_4$	**GetConvolutionParameteriv**	`REDUCE`	Convolution border mode	3.7.6	pixel
CONVOLUTION_FILTER_SCALE	$3 \times R^4$	**GetConvolutionParameterfv**	1,1,1,1	Scale factors applied to convolution filter entries	3.7.3	pixel
CONVOLUTION_FILTER_BIAS	$3 \times R^4$	**GetConvolutionParameterfv**	0,0,0,0	Bias factors applied to convolution filter entries	3.7.3	pixel
CONVOLUTION_FORMAT	$3 \times Z_{42}$	**GetConvolutionParameteriv**	`RGBA`	Convolution filter internal format	3.7.6	–
CONVOLUTION_WIDTH	$3 \times Z^+$	**GetConvolutionParameteriv**	0	Convolution filter width	3.7.6	–
CONVOLUTION_HEIGHT	$2 \times Z^+$	**GetConvolutionParameteriv**	0	Convolution filter height	3.7.6	–

Table 6.37. Pixels (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
POST_CONVOLUTION_x_SCALE	R	**GetFloatv**	1	Component scale factors after convolution; x is `RED`, `GREEN`, `BLUE`, or `ALPHA`	3.7.3	pixel
POST_CONVOLUTION_x_BIAS	R	**GetFloatv**	0	Component bias factors after convolution	3.7.3	pixel
POST_COLOR_MATRIX_x_SCALE	R	**GetFloatv**	1	Component scale factors after color matrix	3.7.3	pixel
POST_COLOR_MATRIX_x_BIAS	R	**GetFloatv**	0	Component bias factors after color matrix	3.7.3	pixel
HISTOGRAM	B	**IsEnabled**	`FALSE`	True if histogramming is enabled	3.7.3	pixel/enable
HISTOGRAM	I	**GetHistogram**	*empty*	Histogram table	3.7.3	–
HISTOGRAM_WIDTH	$2 \times Z^+$	**GetHistogramParameteriv**	0	Histogram table width	3.7.3	–
HISTOGRAM_FORMAT	$2 \times Z_{42}$	**GetHistogramParameteriv**	`RGBA`	Histogram table internal format	3.7.3	–
HISTOGRAM_x_SIZE	$5 \times 2 \times Z^+$	**GetHistogramParameteriv**	0	Histogram table component resolution; x is `RED`, `GREEN`, `BLUE`, `ALPHA`, or `LUMINANCE`	3.7.3	–
HISTOGRAM_SINK	B	**GetHistogramParameteriv**	`FALSE`	True if histogramming consumes pixel groups	3.7.3	–

Table 6.38. Pixels (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
MINMAX	B	**IsEnabled**	FALSE	True if minmax is enabled	3.7.3	pixel/enable
MINMAX	R^n	**GetMinmax**	(M,M,M,M),(m,m,m,m)	Minmax table	3.7.3	–
MINMAX_FORMAT	Z_{42}	**GetMinmaxParameteriv**	RGBA	Minmax table internal format	3.7.3	–
MINMAX_SINK	B	**GetMinmaxParameteriv**	FALSE	True if minmax consumes pixel groups	3.7.3	–
ZOOM_X	R	**GetFloatv**	1.0	x zoom factor	3.7.5	pixel
ZOOM_Y	R	**GetFloatv**	1.0	y zoom factor	3.7.5	pixel
x	$8 \times 32 * \times R$	**GetPixelMap**	0's	RGBA **PixelMap** translation tables; x is a map name from table 3.3	3.7.3	–
x	$2 \times 32 * \times Z$	**GetPixelMap**	0's	Index **PixelMap** translation tables; x is a map name from table 3.3	3.7.3	–
x_SIZE	Z^+	**GetIntegerv**	1	Size of table x	3.7.3	–

Table 6.39. Pixels (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
ORDER	$9 \times Z_{8*}$	**GetMapiv**	1	1d map order	5.1	–
ORDER	$9 \times 2 \times Z_{8*}$	**GetMapiv**	1,1	2d map orders	5.1	–
COEFF	$9 \times 8* \times R^n$	**GetMapfv**	see 5.1	1d control points	5.1	–
COEFF	$9 \times 8* \times 8* \times R^n$	**GetMapfv**	see 5.1	2d control points	5.1	–
DOMAIN	$9 \times 2 \times R$	**GetMapfv**	see 5.1	1d domain endpoints	5.1	–
DOMAIN	$9 \times 4 \times R$	**GetMapfv**	see 5.1	2d domain endpoints	5.1	–
MAP1_x	$9 \times B$	**IsEnabled**	`FALSE`	1d map enables: x is map type	5.1	eval/enable
MAP2_x	$9 \times B$	**IsEnabled**	`FALSE`	2d map enables: x is map type	5.1	eval/enable
MAP1_GRID_DOMAIN	$2 \times R$	**GetFloatv**	0,1	1d grid endpoints	5.1	eval
MAP2_GRID_DOMAIN	$4 \times R$	**GetFloatv**	0,1;0,1	2d grid endpoints	5.1	eval
MAP1_GRID_SEGMENTS	Z^+	**GetFloatv**	1	1d grid divisions	5.1	eval
MAP2_GRID_SEGMENTS	$2 \times Z^+$	**GetFloatv**	1,1	2d grid divisions	5.1	eval
AUTO_NORMAL	B	**IsEnabled**	`FALSE`	True if automatic normal generation enabled	5.1	eval/enable

Table 6.40. Evaluators (**GetMap** takes a map name)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SHADER_TYPE	Z_3	**GetShaderiv**	-	Type of shader (vertex, geometry, or fragment)	2.14.1	–
DELETE_STATUS	B	**GetShaderiv**	`FALSE`	Shader flagged for deletion	2.14.1	–
COMPILE_STATUS	B	**GetShaderiv**	`FALSE`	Last compile succeeded	2.14.1	–
-	S	**GetShaderInfoLog**	empty string	Info log for shader objects	6.1.16	–
INFO_LOG_LENGTH	Z^+	**GetShaderiv**	0	Length of info log	6.1.16	–
-	S	**GetShaderSource**	empty string	Source code for a shader	2.14.1	–
SHADER_SOURCE_LENGTH	Z^+	**GetShaderiv**	0	Length of source code	6.1.16	–

Table 6.41. Shader Object State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CURRENT_PROGRAM	Z^+	**GetIntegerv**	0	Name of current program object	2.14.2	–
DELETE_STATUS	B	**GetProgramiv**	`FALSE`	Program object deleted	2.14.2	–
LINK_STATUS	B	**GetProgramiv**	`FALSE`	Last link attempt succeeded	2.14.2	–
VALIDATE_STATUS	B	**GetProgramiv**	`FALSE`	Last validate attempt succeeded	2.14.2	–
ATTACHED_SHADERS	Z^+	**GetProgramiv**	0	Number of attached shader objects	6.1.16	–
-	$0 * \times Z^+$	**GetAttachedShaders**	empty	Shader objects attached	6.1.16	–
-	S	**GetProgramInfoLog**	empty	Info log for program object	6.1.16	–
INFO_LOG_LENGTH	Z^+	**GetProgramiv**	0	Length of info log	2.14.4	–
ACTIVE_UNIFORMS	Z^+	**GetProgramiv**	0	Number of active uniforms	2.14.4	–
-	$0 * \times Z$	**GetUniformLocation**	–	Location of active uniforms	6.1.16	–
-	$0 * \times Z^+$	**GetActiveUniform**	–	Size of active uniform	2.14.4	–
-	$0 * \times Z^+$	**GetActiveUniform**	–	Type of active uniform	2.14.4	–
-	$0 * \times \texttt{char}$	**GetActiveUniform**	empty	Name of active uniform	2.14.4	–
ACTIVE_UNIFORM_MAX_LENGTH	Z^+	**GetProgramiv**	0	Maximum active uniform name length	6.1.16	–
	$512 * \times R$	**GetUniform**	0	Uniform value	2.14.4	–
ACTIVE_ATTRIBUTES	Z^+	**GetProgramiv**	0	Number of active attributes	2.14.3	–

Table 6.42. Program Object State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
-	$0 * \times Z$	**GetAttribLocation**	–	Location of active generic attribute	2.14.3	–
-	$0 * \times Z^+$	**GetActiveAttrib**	–	Size of active attribute	2.14.3	–
-	$0 * \times Z^+$	**GetActiveAttrib**	–	Type of active attribute	2.14.3	–
-	$0 * \times$ `char`	**GetActiveAttrib**	empty	Name of active attribute	2.14.3	–
ACTIVE_ATTRIBUTE_MAX_LENGTH	Z^+	**GetProgramiv**	0	Maximum active attribute name length	6.1.16	–
GEOMETRY_VERTICES_OUT	Z^+	**GetProgramiv**	0	Maximum number of output vertices	2.15.4	-
GEOMETRY_INPUT_TYPE	Z_5	**GetProgramiv**	`TRIANGLES`	Primitive input type	2.15.1	-
GEOMETRY_OUTPUT_TYPE	Z_3	**GetProgramiv**	`TRIANGLE_STRIP`	Primitive output type	2.15.2	-
TRANSFORM_FEEDBACK_BUFFER_MODE	Z_2	**GetProgramiv**	`INTERLEAVED_ATTRIBS`	Transform feedback mode for the program	6.1.16	–
TRANSFORM_FEEDBACK_VARYINGS	Z^+	**GetProgramiv**	0	Number of varyings to stream to buffer object(s)	6.1.16	–
TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH	Z^+	**GetProgramiv**	0	Maximum transform feedback varying name length	6.1.16	–
-	Z^+	**GetTransformFeedbackVarying**	-	Size of each transform feedback varying variable	2.14.6	–
-	Z^+	**GetTransformFeedbackVarying**	-	Type of each transform feedback varying variable	2.14.6	–
-	$0^+ \times$ `char`	**GetTransformFeedbackVarying**	-	Name of each transform feedback varying variable	2.14.6	–

Table 6.43. Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
UNIFORM_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Uniform buffer object bound to the context for buffer object manipulation	2.14.4	–
UNIFORM_BUFFER_BINDING	$n \times Z^+$	**GetIntegeri_v**	0	Uniform buffer object bound to the specified context binding point	2.14.4	–
UNIFORM_BUFFER_START	$n \times Z^+$	**GetInteger64i_v**	0	Start of bound uniform buffer region	6.1.14	–
UNIFORM_BUFFER_SIZE	$n \times Z^+$	**GetInteger64i_v**	0	Size of bound uniform buffer region	6.1.14	–
ACTIVE_UNIFORM_BLOCKS	Z^+	**GetProgramiv**	0	Number of active uniform blocks in a program	2.14.4	–
ACTIVE_UNIFORM_BLOCK_MAX_NAME_LENGTH	Z^+	**GetProgramiv**	0	Length of longest active uniform block name	2.14.4	–
UNIFORM_TYPE	$0 * \times Z_{27}$	**GetActiveUniformsiv**	-	Type of active uniform	2.14.4	–
UNIFORM_SIZE	$0 * \times Z^+$	**GetActiveUniformsiv**	-	Size of active uniform	2.14.4	–
UNIFORM_NAME_LENGTH	$0 * \times Z^+$	**GetActiveUniformsiv**	-	Uniform name length	2.14.4	–
UNIFORM_BLOCK_INDEX	$0 * \times Z$	**GetActiveUniformsiv**	-	Uniform block index	2.14.4	–
UNIFORM_OFFSET	$0 * \times Z$	**GetActiveUniformsiv**	-	Uniform buffer offset	2.14.4	–

Table 6.44. Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
UNIFORM_ARRAY_STRIDE	$0 * \times Z$	**GetActiveUniformsiv**	-	Uniform buffer array stride	2.14.4	–
UNIFORM_MATRIX_STRIDE	$0 * \times Z$	**GetActiveUniformsiv**	-	Uniform buffer intra-matrix stride	2.14.4	–
UNIFORM_IS_ROW_MAJOR	$0 * \times Z^+$	**GetActiveUniformsiv**	-	Whether uniform is a row-major matrix	2.14.4	–
UNIFORM_BLOCK_BINDING	Z^+	**GetActiveUniformBlockiv**	0	Uniform buffer binding points associated with the specified uniform block	2.14.4	–
UNIFORM_BLOCK_DATA_SIZE	Z^+	**GetActiveUniformBlockiv**	-	Size of the storage needed to hold this uniform block's data	2.14.4	–
UNIFORM_BLOCK_ACTIVE_UNIFORMS	Z^+	**GetActiveUniformBlockiv**	-	Count of active uniforms in the specified uniform block	2.14.4	–
UNIFORM_BLOCK_ACTIVE_UNIFORM_INDICES	$n \times Z^+$	**GetActiveUniformBlockiv**	-	Array of active uniform indices of the specified uniform block	2.14.4	–
UNIFORM_BLOCK_REFERENCED_BY_VERTEX_SHADER	B	**GetActiveUniformBlockiv**	0	True if uniform block is actively referenced by the vertex stage	2.14.4	–
UNIFORM_BLOCK_REFERENCED_BY_GEOMETRY_SHADER	B	**GetActiveUniformBlockiv**	0	True if uniform block is actively referenced by the geometry stage	2.14.4	–
UNIFORM_BLOCK_REFERENCED_BY_FRAGMENT_SHADER	B	**GetActiveUniformBlockiv**	0	True if uniform block is actively referenced by the fragment stage	2.14.4	–

Table 6.45. Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_PROGRAM_TWO_SIDE	B	**IsEnabled**	`FALSE`	Two-sided color mode	2.13.1	enable
CURRENT_VERTEX_ATTRIB	$16 * \times R^4$	**GetVertexAttribfv**	0.0,0.0,0.0,1.0	Current generic vertex attribute values	2.7	current
PROGRAM_POINT_SIZE	B	**IsEnabled**	`FALSE`	Point size mode	3.4	enable

Table 6.46. Vertex and Geometry Shader State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
QUERY_RESULT	Z^+	**GetQueryObjectuiv**	0	Query object result	6.1.12	–
QUERY_RESULT_AVAILABLE	B	**GetQueryObjectiv**	`FALSE`	Is the query object result available?	6.1.12	–

Table 6.47. Query Object State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TRANSFORM_FEEDBACK_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Buffer object bound to generic bind point for transform feedback	6.1.14	–
TRANSFORM_FEEDBACK_BUFFER_BINDING	$n \times Z^+$	**GetIntegeri_v**	0	Buffer object bound to each transform feedback attribute stream	6.1.14	–
TRANSFORM_FEEDBACK_BUFFER_START	$n \times Z^+$	**GetInteger64i_v**	0	Start offset of binding range for each transform feedback attrib. stream	6.1.14	–
TRANSFORM_FEEDBACK_BUFFER_SIZE	$n \times Z^+$	**GetInteger64i_v**	0	Size of binding range for each transform feedback attrib. stream	6.1.14	–
MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS	Z^+	**GetIntegerv**	64	Max number of components to write to a single buffer in interleaved mode	2.19	–
MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS	Z^+	**GetIntegerv**	4	Max number of separate attributes or vayings that can be captured in transform feedback	2.19	–
MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS	Z^+	**GetIntegerv**	4	Max number of components per attribute or varying in separate mode	2.19	–

Table 6.48. Transform Feedback State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
OBJECT_TYPE	Z_1	**GetSynciv**	`SYNC_FENCE`	Type of sync object	5.6	–
SYNC_STATUS	Z_2	**GetSynciv**	`UNSIGNALED`	Sync object status	5.6	–
SYNC_CONDITION	Z_1	**GetSynciv**	`SYNC_GPU_COMMANDS_COMPLETE`	Sync object condition	5.6	–
SYNC_FLAGS	Z	**GetSynciv**	0	Sync object flags	5.6	–

Table 6.49. Sync (state per sync object)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
PERSPECTIVE_CORRECTION_HINT	Z_3	**GetIntegerv**	`DONT_CARE`	Perspective correction hint	5.7	hint
POINT_SMOOTH_HINT	Z_3	**GetIntegerv**	`DONT_CARE`	Point smooth hint	5.7	hint
LINE_SMOOTH_HINT	Z_3	**GetIntegerv**	`DONT_CARE`	Line smooth hint	5.7	hint
POLYGON_SMOOTH_HINT	Z_3	**GetIntegerv**	`DONT_CARE`	Polygon smooth hint	5.7	hint
FOG_HINT	Z_3	**GetIntegerv**	`DONT_CARE`	Fog hint	5.7	hint
GENERATE_MIPMAP_HINT	Z_3	**GetIntegerv**	`DONT_CARE`	Mipmap generation hint	5.7	hint
TEXTURE_COMPRESSION_HINT	Z_3	**GetIntegerv**	`DONT_CARE`	Texture compression quality hint	5.7	hint
FRAGMENT_SHADER_DERIVATIVE_HINT	Z_3	**GetIntegerv**	`DONT_CARE`	Fragment shader derivative accuracy hint	5.7	hint

Table 6.50. Hints

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_LIGHTS	Z^+	**GetIntegerv**	8	Maximum number of lights	2.13.1	–
MAX_CLIP_DISTANCES	Z^+	**GetIntegerv**	8	Maximum number of user clipping planes	2.22	–
MAX_COLOR_MATRIX_STACK_DEPTH	Z^+	**GetIntegerv**	2	Maximum color matrix stack depth	3.7.3	–
MAX_MODELVIEW_STACK_DEPTH	Z^+	**GetIntegerv**	32	Maximum model-view stack depth	2.12.1	–
MAX_PROJECTION_STACK_DEPTH	Z^+	**GetIntegerv**	2	Maximum projection matrix stack depth	2.12.1	–
MAX_TEXTURE_STACK_DEPTH	Z^+	**GetIntegerv**	2	Maximum number depth of texture matrix stack	2.12.1	–
SUBPIXEL_BITS	Z^+	**GetIntegerv**	4	Number of bits of subpixel precision in screen x_w and y_w	3	–
MAX_3D_TEXTURE_SIZE	Z^+	**GetIntegerv**	256	Maximum 3D texture image dimension	3.9.1	–
MAX_TEXTURE_SIZE	Z^+	**GetIntegerv**	1024	Maximum 2D/1D texture image dimension	3.9.1	–
MAX_ARRAY_TEXTURE_LAYERS	Z^+	**GetIntegerv**	256	Maximum number of layers for texture arrays	3.9.1	–
MAX_TEXTURE_LOD_BIAS	R^+	**GetFloatv**	2.0	Maximum absolute texture level of detail bias	3.9.9	–
MAX_CUBE_MAP_TEXTURE_SIZE	Z^+	**GetIntegerv**	1024	Maximum cube map texture image dimension	3.9.1	–
MAX_RENDERBUFFER_SIZE	Z^+	**GetIntegerv**	1024	Maximum width and height of renderbuffers	4.4.2	–

Table 6.51. Implementation Dependent Values

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_PIXEL_MAP_TABLE	Z^+	**GetIntegerv**	32	Maximum size of a **PixelMap** translation table	3.7.3	–
MAX_NAME_STACK_DEPTH	Z^+	**GetIntegerv**	64	Maximum selection name stack depth	5.2	–
MAX_LIST_NESTING	Z^+	**GetIntegerv**	64	Maximum display list call nesting	5.4	–
MAX_EVAL_ORDER	Z^+	**GetIntegerv**	8	Maximum evaluator polynomial order	5.1	–
MAX_VIEWPORT_DIMS	$2 \times Z^+$	**GetIntegerv**	see 2.16.1	Maximum viewport dimensions	2.16.1	–
MAX_ATTRIB_STACK_DEPTH	Z^+	**GetIntegerv**	16	Maximum depth of the server attribute stack	6	–
MAX_CLIENT_ATTRIB_STACK_DEPTH	Z^+	**GetIntegerv**	16	Maximum depth of the client attribute stack	6	–
–	$3 \times Z^+$	**-**	32	Max. size of a color table	3.7.3	–
–	Z^+	**-**	32	Max. size of the histogram table	3.7.3	–
ALIASED_POINT_SIZE_RANGE	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of aliased point sizes	3.4	–
POINT_SIZE_RANGE	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of point sprite sizes	3.4	–
POINT_SIZE_GRANULARITY	R^+	**GetFloatv**	–	Point sprite size granularity	3.4	–

Table 6.52. Implementation Dependent Values (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
ALIASED_LINE_WIDTH_RANGE	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of aliased line widths	3.5	–
SMOOTH_LINE_WIDTH_RANGE (v1.1: LINE_WIDTH_RANGE)	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of antialiased line widths	3.5	–
SMOOTH_LINE_WIDTH_GRANULARITY (v1.1: LINE_WIDTH_GRANULARITY)	R^+	**GetFloatv**	–	Antialiased line width granularity	3.5	–
MAX_CONVOLUTION_WIDTH	$3 \times Z^+$	**GetConvolutionParameteriv**	3	Maximum width of convolution filter	4.3	–
MAX_CONVOLUTION_HEIGHT	$2 \times Z^+$	**GetConvolutionParameteriv**	3	Maximum height of convolution filter	4.3	–
MAX_ELEMENTS_INDICES	Z^+	**GetIntegerv**	–	Recommended max. number of **DrawRangeElements** indices	2.8	–
MAX_ELEMENTS_VERTICES	Z^+	**GetIntegerv**	–	Recommended max. number of **DrawRangeElements** vertices	2.8	–
COMPRESSED_TEXTURE_FORMATS	$4 * \times Z^+$	**GetIntegerv**	-	Enumerated compressed texture formats	3.9.3	–
NUM_COMPRESSED_TEXTURE_FORMATS	Z	**GetIntegerv**	4	Number of compressed texture formats	3.9.3	–
MAX_TEXTURE_BUFFER_SIZE	Z^+	**GetIntegerv**	65536	No. of addressable texels for buffer textures	3.9.5	–
MAX_RECTANGLE_TEXTURE_SIZE	Z^+	**GetIntegerv**	1024	Max. width & height of rectangular textures	3.9.1	–

Table 6.53. Implementation Dependent Values (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
EXTENSIONS	$0 * \times S$	**GetStringi**	–	Supported individual extension names	6.1.4	–
NUM_EXTENSIONS	Z^+	**GetIntegerv**	–	Number of individual extension names	6.1.4	–
MAJOR_VERSION	Z^+	**GetIntegerv**	3	Major version number supported	6.1.4	–
MINOR_VERSION	Z^+	**GetIntegerv**	–	Minor version number supported	6.1.4	–
CONTEXT_FLAGS	Z^+	**GetIntegerv**	–	Context full/forward-compatible flag	6.1.4	–
EXTENSIONS	S	**GetString**	–	Supported extension names	6.1.4	–
RENDERER	S	**GetString**	–	Renderer string	6.1.4	–
SHADING_LANGUAGE_VERSION	S	**GetString**	–	Shading Language version supported	6.1.4	–
VENDOR	S	**GetString**	–	Vendor string	6.1.4	–
VERSION	S	**GetString**	–	OpenGL version supported	6.1.4	–

Table 6.54. Implementation Dependent Version and Extension Support

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_VERTEX_ATTRIBS	Z^+	**GetIntegerv**	16	Number of active vertex attributes	2.7	–
MAX_VERTEX_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1024	Number of components for vertex shader uniform variables	2.14.4	–
MAX_VERTEX_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	12	Max number of vertex uniform buffers per program	2.14.4	–
MAX_VERTEX_OUTPUT_COMPONENTS	Z^+	**GetIntegerv**	64	Max number of components of outputs written by a vertex shader	2.14.6	-
MAX_VERTEX_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	16	Number of texture image units accessible by a vertex shader	2.14.4	–

Table 6.55. Implementation Dependent Vertex Shader Limits

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_GEOMETRY_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	12	Max number of geometry uniform buffers per program	2.14.4	–
MAX_GEOMETRY_INPUT_COMPONENTS	Z^+	**GetIntegerv**	64	Max number of components of inputs read by a geometry shader	2.15.4	-
MAX_GEOMETRY_OUTPUT_COMPONENTS	Z^+	**GetIntegerv**	128	Max number of components of outputs written by a geometry shader	2.15.4	-
MAX_GEOMETRY_OUTPUT_VERTICES	Z^+	**GetIntegerv**	256	Maximum number of vertices that any geometry shader can can emit	2.15.4	-
MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS	Z^+	**GetIntegerv**	1024	Maximum number of total components (all vertices) of active varyings that a geometry shader can emit	2.15.4	-
MAX_GEOMETRY_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	16	Number of texture image units accessible by a geometry shader	2.15.4	-

Table 6.56. Implementation Dependent Geometry Shader Limits

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_FRAGMENT_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1024	Number of components for frag. shader uniform variables	3.12.1	–
MAX_FRAGMENT_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	12	Max number of fragment uniform buffers per program	2.14.4	–
MAX_FRAGMENT_INPUT_COMPONENTS	Z^+	**GetIntegerv**	128	Max number of components of inputs read by a fragment shader	3.12.2	-
MAX_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	16	Number of texture image units accessible by a fragment shader	2.14.4	–
MIN_PROGRAM_TEXEL_OFFSET	Z	**GetIntegerv**	-8	Minimum texel offset allowed in lookup	2.14.4	–
MAX_PROGRAM_TEXEL_OFFSET	Z	**GetIntegerv**	7	Maximum texel offset allowed in lookup	2.14.4	–
MAX_TEXTURE_UNITS	Z^+	**GetIntegerv**	2	Number of fixed-function texture units	2.5	–
MAX_TEXTURE_COORDS	Z^+	**GetIntegerv**	8	Number of texture coordinate sets	2.7	–

Table 6.57. Implementation Dependent Fragment Processing Limits

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_UNIFORM_BUFFER_BINDINGS	Z^+	**GetIntegerv**	36	Max number of uniform buffer binding points on the context	2.14.4	–
MAX_UNIFORM_BLOCK_SIZE	Z^+	**GetIntegerv**	16384	Max size in basic machine units of a uniform block	2.14.4	–
UNIFORM_BUFFER_OFFSET_ALIGNMENT	Z^+	**GetIntegerv**	1	Minimum required alignment for uniform buffer sizes and offsets	2.14.4	–
MAX_COMBINED_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	36	Max number of uniform buffers per program	2.14.4	–
MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	†	Number of words for vertex shader uniform variables in all uniform blocks (including default)	2.14.4	–
MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	†	Number of words for fragment shader uniform variables in all uniform blocks (including default)	2.14.4	–
MAX_COMBINED_GEOMETRY_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	†	Number of words for geometry shader uniform variables in all uniform blocks (including default)	2.14.4	–
MAX_VARYING_COMPONENTS	Z^+	**GetIntegerv**	60	Number of components for varying variables	2.14.6	–
MAX_COMBINED_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	48	Total number of texture units accessible by the GL	2.14.4	–

Table 6.58. Implementation Dependent Aggregate Shader Limits
† The minimum value for each stage is `MAX_`*`stage`*`_UNIFORM_BLOCKS` × `MAX_`*`stage`*`_UNIFORM_BLOCK_SIZE` + `MAX_`*`stage`*`_UNIFORM_COMPONENTS`

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_SAMPLE_MASK_WORDS	Z^+	**GetIntegerv**	1	Maximum number of sample mask words	4.1.3	–
MAX_COLOR_TEXTURE_SAMPLES	Z^+	**GetIntegerv**	1	Maximum number of samples in a color multisample texture	4.1.3	–
MAX_DEPTH_TEXTURE_SAMPLES	Z^+	**GetIntegerv**	1	Maximum number of samples in a depth/stencil multisample texture	4.1.3	–
MAX_INTEGER_SAMPLES	Z^+	**GetIntegerv**	1	Maximum number of samples in integer format multisample buffers	4.4.2	–
QUADS_FOLLOW_PROVOKING_VERTEX	B	**GetBooleanv**	–	Whether quads follow provoking vertex convention	2.21	–
QUERY_COUNTER_BITS	$3 \times Z^+$	**GetQueryiv**	see 6.1.12	Asynchronous query counter bits	6.1.12	–
MAX_SERVER_WAIT_TIMEOUT	Z^+	**GetInteger64v**	0	Maximum **WaitSync** timeout interval	5.6.1	–

Table 6.59. Implementation Dependent Values (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
AUX_BUFFERS	Z^+	**GetIntegerv**	0	Number of auxiliary buffers	4.2.1	–
MAX_DRAW_BUFFERS	Z^+	**GetIntegerv**	8	Maximum number of active draw buffers	4.2.1	–
RGBA_MODE	B	**GetBooleanv**	–	True if color buffers store RGBA	2.7	–
INDEX_MODE	B	**GetBooleanv**	–	True if color buffers store indexes	2.7	–
DOUBLEBUFFER	B	**GetBooleanv**	–	True if front & back buffers exist	4.2.1	–
STEREO	B	**GetBooleanv**	–	True if left & right buffers exist	6	–
SAMPLE_BUFFERS	Z^+	**GetIntegerv**	0	Number of multisample buffers	3.3.1	–
SAMPLES	Z^+	**GetIntegerv**	0	Coverage mask size	3.3.1	–
SAMPLE_POSITION	$n \times 2 \times R^{[0,1]}$	**GetMultisamplefv**	–	Explicit sample positions	3.3.1	–
MAX_COLOR_ATTACHMENTS	Z^+	**GetIntegerv**	8	Maximum number of FBO attachment points for color buffers	4.4.2	–
MAX_SAMPLES	Z^+	**GetIntegerv**	4	Maximum number of samples supported for multisampling	4.4.2	–
x_BITS	Z^+	**GetIntegerv**	-	Number of bits in x color buffer component. x is one of `RED`, `GREEN`, `BLUE`, `ALPHA`, or `INDEX`	4	–
DEPTH_BITS	Z^+	**GetIntegerv**	-	Number of depth buffer planes	4	–
STENCIL_BITS	Z^+	**GetIntegerv**	-	Number of stencil planes	4	–
ACCUM_x_BITS	Z^+	**GetIntegerv**	-	Number of bits in x accumulation buffer component (x is `RED`, `GREEN`, `BLUE`, or `ALPHA`	4	–

Table 6.60. Framebuffer Dependent Values

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
LIST_BASE	Z^+	**GetIntegerv**	0	Setting of **ListBase**	5.4	list
LIST_INDEX	Z^+	**GetIntegerv**	0	Number of display list under construction; 0 if none	5.4	–
LIST_MODE	Z^+	**GetIntegerv**	0	Mode of display list under construction; undefined if none	5.4	–
–	$16 * \times A$	**–**	empty	Server attribute stack	6	–
ATTRIB_STACK_DEPTH	Z^+	**GetIntegerv**	0	Server attribute stack pointer	6	–
–	$16 * \times A$	**–**	empty	Client attribute stack	6	–
CLIENT_ATTRIB_STACK_DEPTH	Z^+	**GetIntegerv**	0	Client attribute stack pointer	6	–
NAME_STACK_DEPTH	Z^+	**GetIntegerv**	0	Name stack depth	5.2	–
RENDER_MODE	Z_3	**GetIntegerv**	`RENDER`	**RenderMode** setting	5.2	–
SELECTION_BUFFER_POINTER	Y	**GetPointerv**	0	Selection buffer pointer	5.2	select
SELECTION_BUFFER_SIZE	Z^+	**GetIntegerv**	0	Selection buffer size	5.2	select
FEEDBACK_BUFFER_POINTER	Y	**GetPointerv**	0	Feedback buffer pointer	5.3	feedback
FEEDBACK_BUFFER_SIZE	Z^+	**GetIntegerv**	0	Feedback buffer size	5.3	feedback
FEEDBACK_BUFFER_TYPE	Z_5	**GetIntegerv**	`2D`	Feedback type	5.3	feedback
–	$n \times Z_8$	**GetError**	0	Current error code(s)	2.5	–
–	$n \times B$	**–**	`FALSE`	True if there is a corresponding error	2.5	–
	B	**–**	`FALSE`	Occlusion query active	4.1.7	–
CURRENT_QUERY	$3 \times Z^+$	**GetQueryiv**	0	Active query object names	6.1.12	–
COPY_READ_BUFFER	Z^+	**GetIntegerv**	0	Buffer object bound to copy buffer “read” bind point	2.9.5	–
COPY_WRITE_BUFFER	Z^+	**GetIntegerv**	0	Buffer object bound to copy buffer “write” bind point	2.9.5	–
TEXTURE_CUBE_MAP_SEAMLESS	B	**IsEnabled**	`FALSE`	Seamless cube map filtering enable	3.9.8	–

Table 6.61. Miscellaneous

Appendix A

Invariance

The OpenGL specification is not pixel exact. It therefore does not guarantee an exact match between images produced by different GL implementations. However, the specification does specify exact matches, in some cases, for images produced by the same implementation. The purpose of this appendix is to identify and provide justification for those cases that require exact matches.

A.1 Repeatability

The obvious and most fundamental case is repeated issuance of a series of GL commands. For any given GL and framebuffer state *vector*, and for any GL command, the resulting GL and framebuffer state must be identical whenever the command is executed on that initial GL and framebuffer state.

One purpose of repeatability is avoidance of visual artifacts when a double-buffered scene is redrawn. If rendering is not repeatable, swapping between two buffers rendered with the same command sequence may result in visible changes in the image. Such false motion is distracting to the viewer. Another reason for repeatability is testability.

Repeatability, while important, is a weak requirement. Given only repeatability as a requirement, two scenes rendered with one (small) polygon changed in position might differ at every pixel. Such a difference, while within the law of repeatability, is certainly not within its spirit. Additional invariance rules are desirable to ensure useful operation.

A.2 Multi-pass Algorithms

Invariance is necessary for a whole set of useful multi-pass algorithms. Such algorithms render multiple times, each time with a different GL mode vector, to eventually produce a result in the framebuffer. Examples of these algorithms include:

- “Erasing” a primitive from the framebuffer by redrawing it, either in a different color or using the XOR logical operation.
- Using stencil operations to compute capping planes.

On the other hand, invariance rules can greatly increase the complexity of high-performance implementations of the GL. Even the weak repeatability requirement significantly constrains a parallel implementation of the GL. Because GL implementations are required to implement ALL GL capabilities, not just a convenient subset, those that utilize hardware acceleration are expected to alternate between hardware and software modules based on the current GL mode vector. A strong invariance requirement forces the behavior of the hardware and software modules to be identical, something that may be very difficult to achieve (for example, if the hardware does floating-point operations with different precision than the software).

What is desired is a compromise that results in many compliant, high-performance implementations, and in many software vendors choosing to port to OpenGL.

A.3 Invariance Rules

For a given instantiation of an OpenGL rendering context:

Rule 1 *For any given GL and framebuffer state vector, and for any given GL command, the resulting GL and framebuffer state must be identical each time the command is executed on that initial GL and framebuffer state.*

Rule 2 *Changes to the following state values have no side effects (the use of any other state value is not affected by the change):*

Required:

- *Framebuffer contents (all bitplanes)*
- *The color buffers enabled for writing*
- *The values of matrices other than the top-of-stack matrices*

- *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).*

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* `gl_FragCoord.z` *to* `gl_FragDepth` *are depth-invariant with respect to each other, for those fragments where the assignment to* `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, and features deprecated in a previous revision may be removed. 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.

6. Bitmaps and pixel transfers do not cause selection hits.

7. The mask specified as the third argument to **StencilFunc** affects the operands of the stencil comparison function, but has no direct effect on the update of the stencil buffer. The mask specified by **StencilMask** has no effect on the

stencil comparison function; it limits the effect of the update of the stencil buffer.

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; the positions of vertex A and C are identical; and the positions of vertex B and D are identical. Vertex positions are identical for the fixed-function pipeline if they are specified with the same input values and the state of coordinate transformations is identical when the vertices are processed; otherwise they are identical if the `gl_Position` values output by the vertex (or if active, geometry) shader are identical. Then, when the fragments produced by rasterization of both polygons are taken together, each fragment intersecting the interior of the shared edge is produced exactly once.

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 `LIGHT`i, 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 4×4 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 4×4 block is treated as a single pixel. If an RGTC image has a width or height that is not a multiple of four, the data corresponding to texels outside the image are irrelevant and undefined.

When an RGTC image with a width of *w*, height of *h*, and block size of *blocksize* (8 or 16 bytes) is decoded, the corresponding image size (in bytes) is:

$$\lceil \frac{w}{4} \rceil \times \lceil \frac{h}{4} \rceil \times blocksize.$$

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:

$$blocksize \times \left(\lceil \frac{w}{4} \rceil \times \lfloor \frac{y}{4} \rfloor + \lfloor \frac{x}{4} \rfloor \right).$$

The data corresponding to a specific texel (x, y) are extracted from a 4×4 texel block using a relative (x, y) value of

$$(x \mod 4, y \mod 4).$$

There are four distinct RGTC image formats:

C.1.1 Format `COMPRESSED_RED_RGTC1`

Each 4×4 block of texels consists of 64 bits of unsigned red image data.

Each red image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

$$red_0, red_1, bits_0, bits_1, bits_2, bits_3, bits_4, bits_5$$

The 6 $bits_*$ bytes of the block are decoded into a 48-bit bit vector:

$$bits = bits_0 + 256 \times (bits_1 + 256 \times (bits_2 + 256 \times (bits_3 + 256 \times (bits_4 + 256 \times bits_5))))$$

red_0 and red_1 are 8-bit unsigned integers that are unpacked to red values RED_0 and RED_1 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:

$$R = \begin{cases} RED_0, & red_0 > red_1, code(x,y) = 0 \\ RED_1, & red_0 > red_1, code(x,y) = 1 \\ \frac{6RED_0+RED_1}{7,} & red_0 > red_1, code(x,y) = 2 \\ \frac{5RED_0+2RED_1}{7,} & red_0 > red_1, code(x,y) = 3 \\ \frac{4RED_0+3RED_1}{7,} & red_0 > red_1, code(x,y) = 4 \\ \frac{3RED_0+4RED_1}{7,} & red_0 > red_1, code(x,y) = 5 \\ \frac{2RED_0+5RED_1}{7,} & red_0 > red_1, code(x,y) = 6 \\ \frac{RED_0+6RED_1}{7,} & red_0 > red_1, code(x,y) = 7 \\ RED_0, & red_0 \leq red_1, code(x,y) = 0 \\ RED_1, & red_0 \leq red_1, code(x,y) = 1 \\ \frac{4RED_0+RED_1}{5,} & red_0 \leq red_1, code(x,y) = 2 \\ \frac{3RED_0+2RED_1}{5,} & red_0 \leq red_1, code(x,y) = 3 \\ \frac{2RED_0+3RED_1}{5,} & red_0 \leq red_1, code(x,y) = 4 \\ \frac{RED_0+4RED_1}{5,} & red_0 \leq red_1, code(x,y) = 5 \\ RED_{min}, & red_0 \leq red_1, code(x,y) = 6 \\ RED_{max}, & red_0 \leq red_1, code(x,y) = 7 \end{cases}$$

RED_{min} and RED_{max} are 0.0 and 1.0 respectively.

Since the decoded texel has a red format, the resulting RGBA value for the texel is $(R, 0, 0, 1)$.

C.1.2 Format `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 red_0, red_1, RED_0, RED_1, RED_{min}, and RED_{max} are signed values defined as follows:

red_0 and red_1 are 8-bit signed (twos complement) integers.

$$RED_0 = \begin{cases} \frac{red_0}{127.0,} & red_0 > -128 \\ -1.0, & red_0 = -128 \end{cases}$$

$$RED_1 = \begin{cases} \frac{red_1}{127.0,} & red_1 > -128 \\ -1.0, & red_1 = -128 \end{cases}$$

$$RED_{min} = -1.0$$

$$RED_{max} = 1.0$$

CAVEAT for signed red_0 and red_1 values: the expressions $red_0 > red_1$ and $red_0 \leq red_1$ above are considered undefined (read: may vary by implementation) when $red_0 = -127$ and $red_1 = -128$. This is because if red_0 were remapped to -127 prior to the comparison to reduce the latency of a hardware decompressor, the expressions would reverse their logic. Encoders for the signed red-green formats should avoid encoding blocks where $red_0 = -127$ and $red_1 = -128$.

C.1.3 Format `COMPRESSED_RG_RGTC2`

Each 4×4 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 4×4 block of texels consists of 64 bits of compressed signed red image data followed by 64 bits of compressed signed green image data.

The first 64 bits of compressed red are decoded exactly like `COMPRESSED_SIGNED_RED_RGTC1` above.

The second 64 bits of compressed green are decoded exactly like `COMPRESSED_SIGNED_RED_RGTC1` above except the decoded value R for this second block is considered the resulting green value G.

Since this image has a red-green format, the resulting RGBA value is $(R, G, 0, 1)$.

Appendix D

Shared Objects and Multiple Contexts

This appendix describes special considerations for objects shared between multiple OpenGL context, including deletion behavior and how changes to shared objects are propagated between contexts.

Objects that can be shared between contexts include pixel and vertex buffer objects, display lists, program and shader objects, renderbuffer objects, sync objects, and texture objects (except for the texture objects named zero).

Framebuffer, query, and vertex array objects are not shared.

Implementations may allow sharing between contexts implementing different OpenGL versions or different profiles of the same OpenGL version (see appendix E). However, implementation-dependent behavior may result when aspects and/or behaviors of such shared objects do not apply to, and/or are not described by more than one version or profile.

D.1 Object Deletion Behavior

D.1.1 Automatic Unbinding of Deleted Objects

When a buffer, texture, or renderbuffer object is deleted, it is unbound from any bind points it is bound to in the current context, as described for **DeleteBuffers**, **DeleteTextures**, and **DeleteRenderbuffers**. Bind points in other contexts are not affected.

D.1.2 Deleted Object and Object Name Lifetimes

When a buffer, texture, renderbuffer, or sync object is deleted, its name immediately becomes invalid (e.g. is marked unused), but the underlying object will not be deleted until it is no longer *in use*. A buffer, texture, or renderbuffer object is in use while it is attached to any container object or bound to a context bind point in any context. A sync object is in use while there is a corresponding fence command which has not yet completed and signaled the sync object, or while there are any GL clients and/or servers blocked on the sync object as a result of **ClientWaitSync** or **WaitSync** commands.

When a program or shader object is deleted, it is flagged for deletion, but its name remains valid until the underlying object can be deleted because it is no longer in use. A program object is in use while it is the current program in any context. A shader object is in use while it is attached to any program object.

Caution should be taken when deleting an object attached to a container object (such as a buffer object attached to a vertex array object, or a renderbuffer or texture attached to a framebuffer object), or a shared object bound in multiple contexts. Following its deletion, the object's name may be used by any context to create a new object or returned by **Gen*** commands, even though the underlying object state and data may still be referred to by container objects, or in use by contexts other than the one in which the object was deleted. Such a container or other context may continue using the object, and may still contain state identifying its name as being currently bound, until such time as the container object is deleted, the attachment point of the container object is changed to refer to another object, or another attempt to bind or attach the name is made in that context. Since the name is marked unused, binding the name will create a new object with the same name, and attaching the name will generate an error. The underlying storage backing a deleted object will not be reclaimed by the GL until all references to the object from container object attachment points or context binding points are removed.

D.2 Sync Objects and Multiple Contexts

When multiple GL clients and/or servers are blocked on a single sync object and that sync object is signalled, all such blocks are released. The order in which blocks are released is implementation-dependent.

D.3 Propagating State Changes

Data is information the GL implementation does not have to inspect, and does not have an operational effect. Currently, data consists of:

- Pixels in the framebuffer.
- The contents of textures and renderbuffers.
- The contents of buffer objects.

State determines the configuration of the rendering pipeline and the driver does have to inspect.

In hardware-accelerated GL implementations, state typically lives in GPU registers, while data typically lives in GPU memory.

When the state of an object T is changed, such changes are not always immediately visible, and do not always immediately affect GL operations involving that object. Changes to an object may occur via any of the following means:

- State-setting commands, such as **TexParameter**.
- Data-setting commands, such as **TexSubImage*** or **BufferSubData**.
- Data-setting through rendering to attached renderbuffers or transform feedback operations.
- Commands that affect both state and data, such as **TexImage*** and **BufferData**.
- Changes to mapped buffer data followed by a command such as **UnmapBuffer** or **FlushMappedBufferRange**.

D.3.1 Determining Completion of Changes to an object

The object T is considered to have been changed once a command such as described in section D.3 has completed. Completion of a command [1] may be determined either by calling **Finish**, or by calling **FenceSync** and executing a **WaitSync** command on the associated sync object. The second method does not require a round trip to the GL server and may be more efficient, particularly when changes to T in one context must be known to have completed before executing commands dependent on those changes in another context.

---

[1] The GL already specifies that a single context processes commands in the order they are received. This means that a change to an object in a context at time t must be completed by the time a command issued in the same context at time $t + 1$ uses the result of that change.

D.3.2 Definitions

In the remainder of this section, the following terminology is used:

- An object *T* is *directly attached* to the current context if it has been bound to one of the context binding points. Examples include but are not limited to bound textures, bound framebuffers, bound vertex arrays, and current programs.
- *T* is *indirectly attached* to the current context if it is attached to another object *C*, referred to as a *container object*, and *C* is itself directly or indirectly attached. Examples include but are not limited to renderbuffers or textures attached to framebuffers; buffers attached to vertex arrays; and shaders attached to programs.
- An object *T* which is directly attached to the current context may be *re-attached* by re-binding *T* at the same bind point. An object *T* which is indirectly attached to the current context may be re-attached by re-attaching the container object *C* to which *T* is attached.

  Corollary: re-binding *C* to the current context re-attaches *C* and its hierarchy of contained objects.

D.3.3 Rules

The following rules must be obeyed by all GL implementations:

Rule 1 *If the state of object* T *is changed in the current context while* T *is directly or indirectly attached, then all operations on* T *will use that new state in the current context.*

Note: *The intent of this rule is to address state changes in a single context only. The multi-context case is handled by the other rules.*

Note: *"Updates" via rendering or transform feedback are treated consistently with update via GL commands. Once* **EndTransformFeedback** *has been issued, any command in the same context that uses the results of the transform feedback operation will see the results. If a feedback loop is setup between rendering and transform feedback (see above), results will be undefined.*

Rule 2 *While a container object* C *is bound, any changes made to* C*'s attachments in the current context are guaranteed to be seen. To guarantee seeing changes made in another context to objects attached to* C*, such changes must be completed in that other context (see section D.3.1) prior to* C *being bound. Changes* *made in another context but not determined to have completed as described in section D.3.1, or after* C *is bound in the current context, are not guaranteed to be seen.*

Rule 3 *State changes to shared objects are not automatically propagated between contexts. If the state of a shared object* T *is changed in a context other than the current context, and* T *is already directly or indirectly attached to the current context, any operations on the current context involving* T *via those attachments are not guaranteed to use its new state.*

Rule 4 *If the state of a shared object* T *is changed in a context other than the current context,* T *must be attached or re-attached to at least one binding point in the current context in order to guarantee that the new state of* T *is visible in the current context.*

Note: "Attached or re-attached" *means either attaching an object to a binding point it wasn't already attached to, or attaching an object again to a binding point it was already attached.*

Note: *This rule also applies to the pointer to the data store of an object. The pointer itself is state, while the content of the data store are data, not state. To guarantee that another context sees data updates to an object, you should attach or re-attach the object in that context, since the pointer to the data store could have changed.*

Note: *To be sure that a data update resulting from a transform-feedback operation in another context is visible in the current context, the app needs to make sure that the command* **EndTransformFeedback** *has completed (see section D.3.1).*

Example: *If a texture image is bound to multiple texture bind points and the texture is modified in another context, re-binding the texture at any one of the texture bind points is sufficient to cause the modifications to be visible at all texture bind points.*

Appendix E

Profiles and the Deprecation Model

OpenGL 3.0 introduces a deprecation model in which certain features may be marked as *deprecated*. Deprecated features are expected to be completely removed from a future version of OpenGL. Deprecated features are summarized in section E.2.

To aid developers in writing applications which will run on such future versions, it is possible to create an OpenGL 3.0 context which does not support deprecated features. Such a context is called a *forward compatible* context, while a context supporting all OpenGL 3.0 features is called a *full* context. Forward compatible contexts cannot restore deprecated functionality through extensions, but they may support additional, non-deprecated functionality through extensions.

Profiles define subsets of OpenGL functionality targeted to specific application domains. OpenGL 3.2 defines two profiles (see below), and future versions may introduce additional profiles addressing embedded systems or other domains. OpenGL 3.2 implementations are not required to support all defined profiles, but must support the *core* profile described below.

To enable application control of deprecation and profiles, new *context creation APIs* have been defined as extensions to GLX and WGL. These APIs allow specifying a particular version, profile, and full or forward compatible status, and will either create a context compatible with the request, or fail (if, for example, requesting an OpenGL version or profile not supported by the implementation),

Only the ARB may define OpenGL profiles and deprecated features.

E.1 Core and Compatibility Profiles

OpenGL 3.2 is the first version of OpenGL to define multiple profiles. The *core profile* builds on OpenGL 3.1 by adding features described in section H.1. The *compatibility profile* builds on the combination of OpenGL 3.1 with the special `GL_ARB_compatibility` extension defined together with OpenGL 3.1, adding the same new features and in some cases extending their definition to interact with existing features of OpenGL 3.1 only found in `GL_ARB_compatibility`.

It is not possible to implement both core and compatibility profiles in a single GL context, since the core profile mandates functional restrictions not present in the compatibility profile. Refer to the `WGL_ARB_create_context_profile` and `GLX_ARB_create_context_profile` extensions (see appendix I.3.68) for information on creating a context implementing a specific profile.

E.2 Deprecated and Removed Features

OpenGL 3.0 defined a set of *deprecated features*. OpenGL 3.1 removed most of the deprecated features and moved them into the optional `GL_ARB_compatibility` extension. The OpenGL 3.2 core profile removes the same features as OpenGL 3.1, while the optional compatibility profile supports all those features.

Deprecated and removed features are summarized below in two groups: features which are marked deprecated by the core profile, but have not yet been removed, and features actually removed from the core profile of the current version of OpenGL (no features have been removed from or deprecated in the compatibility profile).

Functions which have been removed will generate an `INVALID_OPERATION` error if called in the core profile or in a forward-compatible context. Functions which are partially removed (e.g. no longer accept some parameter values) will generate the errors appropriate for any other unrecognized value of that parameter when a removed parameter value is passed in the core profile or a forward-compatible context. Functions which are deprecated but have not yet been removed from the core profile continue to operate normally except in a forward-compatible context, where they are also removed.

E.2.1 Deprecated But Still Supported Features

The following features are deprecated, but still present in the core profile. They may be removed from a future version of OpenGL, and are removed in a forward-compatible context implementing the core profile.

- Wide lines - **LineWidth** values greater than 1.0 will generate an `INVALID_VALUE` error.

- Global component limit query - the implementation-dependent values `MAX_VARYING_COMPONENTS` and `MAX_VARYING_FLOATS`.

E.2.2 Removed Features

- Application-generated object names - the names of all object types, such as buffer, query, and texture objects, must be generated using the corresponding **Gen*** commands. Trying to bind an object name not returned by a **Gen*** command will result in an `INVALID_OPERATION` error. This behavior is already the case for framebuffer, renderbuffer, and vertex array objects. Object types which have default objects (objects named zero), such as vertex array, framebuffer, and texture objects, may also bind the default object, even though it is not returned by **Gen***.

- Color index mode - No color index visuals are supplied by the window system-binding APIs such as GLX and WGL, so the default framebuffer is always in RGBA mode. All language and state related to color index mode vertex, rasterization, and fragment processing behavior is removed. `COLOR_INDEX` formats are also deprecated.

- OpenGL Shading Language versions 1.10 and 1.20. These versions of the shading language depend on many API features that have also been deprecated.

- **Begin / End** primitive specification - **Begin**, **End**, and **EdgeFlag*** (section 2.6.1); **Color***, **FogCoord***, **Index***, **Normal3***, **SecondaryColor3***, **TexCoord***, **Vertex*** **Vertex*** (section 2.7); and all associated state in tables 6.5 and 6.6. 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 **InterleavedArrays**, **ClientActiveTexture** (section 2.8); **Frustum**, **LoadIdentity**, **LoadMatrix**, **LoadTransposeMatrix**, **MatrixMode**, **MultMatrix**, **MultTransposeMatrix**, **Ortho**, **PopMatrix**, **PushMatrix**, **Rotate**, **Scale**, and **Translate** (section 2.12.1; **En-**

able/Disable targets `RESCALE_NORMAL` and `NORMALIZE` (section 2.12.2); **TexGen*** and **Enable/Disable** targets `TEXTURE_GEN_*` (section 2.12.3, **Material***, **Light***, **LightModel***, and **ColorMaterial**, **ShadeModel**, and **Enable/Disable** targets `LIGHTING`. `VERTEX_PROGRAM_TWO_SIDE`, `LIGHT`i, and `COLOR_MATERIAL` (sections 2.13.2 and 2.13.3; **ClipPlane**; and all associated fixed-function vertex array, multitexture, matrix and matrix stack, normal and texture coordinate, lighting, and clipping state. A vertex shader must be defined in order to draw primitives.

Language referring to edge flags in the current specification is modified as though all edge flags are `TRUE`.

Note that the **FrontFace** and **ClampColor** commands in section 2.13 are **not** deprecated, as they still affect other non-deprecated functionality; however, the **ClampColor** *target*s `CLAMP_VERTEX_COLOR` and `CLAMP_FRAGMENT_COLOR` are deprecated.

- Client vertex and index arrays - all vertex array attribute and element array index pointers must refer to buffer objects (section 2.9.6). 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 2.24), and all associated state.
- Two-sided color selection (section 2.13.1) - **Enable** target `VERTEX_PROGRAM_TWO_SIDE`; OpenGL Shading Language builtins `gl_BackColor` and `gl_BackSecondaryColor`; and all associated state.
- 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.

- Separate polygon draw mode - **PolygonMode** *face* values of `FRONT` and `BACK`; polygons are always drawn in the same mode, no matter which face is being rasterized.

- Polygon Stipple - **PolygonStipple** and **Enable/Disable** target `POLYGON_STIPPLE` (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 3.7.6), and all associated state and commands defining that state.

- Pixel drawing - **DrawPixels** and **PixelZoom** (section 3.7.5). 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) and the `BITMAP` external format.

- Legacy OpenGL 1.0 pixel formats - the values 1, 2, 3, and 4 are no longer accepted as internal formats by **TexImage*** or any other command taking an internal format argument. The initial internal format of a texel array is `RGBA` instead of 1 (see section 3.9.13). `TEXTURE_COMPONENTS` is deprecated; always use `TEXTURE_INTERNAL_FORMAT`.

- Legacy pixel formats - all `ALPHA`, `LUMINANCE`, `LUMINANCE_ALPHA`, and `INTENSITY` external and internal formats, including compressed, floating-point, and integer variants (see tables 3.6, 3.16, 3.18, 3.20, 3.25, 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 3.9.16 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; and all associated state.

- Automatic mipmap generation - **TexParameter*** *target* `GENERATE_MIPMAP` (section 3.9.9), and all associated state.

- Fixed-function fragment processing - **AreTexturesResident**, **PrioritizeTextures**, and **TexParameter** *target* `TEXTURE_PRIORITY`; **TexEnv** *target* `TEXTURE_ENV`, and all associated parameters; **TexEnv** *target* `TEXTURE_FILTER_CONTROL`, and parameter name `TEXTURE_LOD_BIAS`; **Enable** *targets* of all dimensionalities (`TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`, and `TEXTURE_CUBE_MAP`); **Enable** target `COLOR_SUM`; **Enable** *target* `FOG`, **Fog**, and all associated parameters; the implementation-dependent values `MAX_TEXTURE_UNITS` and `MAX_TEXTURE_COORDS`; and all associated state.

- Alpha test - **AlphaFunc** and **Enable/Disable** target `ALPHA_TEST` (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 6.60); and all associated state.

  Window system-binding APIs such as GLX and WGL may choose to either not expose window configs containing accumulation buffers, or to ignore accumulation buffers when the default framebuffer bound to a GL context contains them.

- Pixel copying - **CopyPixels** (the comments also applying to **CopyTexImage** will be moved to section 3.9.2).

- Auxiliary color buffers, including `AUX`i targets of the default framebuffer.

- Context framebuffer size queries - `RED_BITS`, `GREEN_BITS`, `BLUE_BITS`, `ALPHA_BITS`, `DEPTH_BITS`, and `STENCIL_BITS`.

- Evaluators - **Map***, **EvalCoord***, **MapGrid***, **EvalMesh***, **EvalPoint***, and all evaluator map enables 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`, `POINT_SMOOTH_HINT`, `FOG_HINT`, and `GENERATE_MIPMAP_HINT` targets to **Hint** (section 5.7).

- Attribute stacks - **PushAttrib**, **PushClientAttrib**, **PopAttrib**, **PopClientAttrib**, the `MAX_ATTRIB_STACK_DEPTH`, `MAX_CLIENT_ATTRIB_STACK_DEPTH`, `ATTRIB_STACK_DEPTH`, and `CLIENT_ATTRIB_STACK_DEPTH` state, the client and server attribute stacks, and the values `ALL_ATTRIB_BITS` and `CLIENT_ALL_ATTRIB_BITS` (section 6.1.19).

- Unified extension string - `EXTENSIONS` target to **GetString** (section 6.1.4).

- Token names and queries - all token names and queries not otherwise mentioned above for deprecated state, as well as all query entry points where all valid targets of that query are deprecated state (chapter 6 and the state tables)

Appendix F

Version 3.0 and Before

OpenGL version 3.0, released on August 11, 2008, is the eighth revision since the original version 1.0. When using a *full* 3.0 context, OpenGL 3.0 is upward compatible with earlier versions, meaning that any program that runs with a 2.1 or earlier GL implementation will also run unchanged with a 3.0 GL implementation. OpenGL 3.0 context creation is done using a window system binding API, and on most platforms a new command, defined by extensions introduced along with OpenGL 3.0, must be called to create a 3.0 context. Calling the older context creation commands will return an OpenGL 2.1 context. When using a *forward compatible* context, many OpenGL 2.1 features are not supported.

Following are brief descriptions of changes and additions to OpenGL 3.0. Descriptions of changes and additions in earlier versions of OpenGL (versions 1.1, 1.2, 1.2.1, 1.3, 1.4, 1.5, 2.0, and 2.1) are omitted in this specification, but may be found in the OpenGL 3.0 Specification, available on the World Wide Web at URL

http://www.opengl.org/registry/

F.1 New Features

New features in OpenGL 3.0, including the extension or extensions if any on which they were based, include:

- API support for the new texture lookup, texture format, and integer and unsigned integer capabilities of the OpenGL Shading Language 1.30 specification (`GL_EXT_gpu_shader4`).
- Conditional rendering (`GL_NV_conditional_render`).

- Fine control over mapping buffer subranges into client space and flushing modified data (`GL_APPLE_flush_buffer_range`).
- Floating-point color and depth internal formats for textures and renderbuffers (`GL_ARB_color_buffer_float`, `GL_NV_depth_buffer_float`, `GL_ARB_texture_float`, `GL_EXT_packed_float`, and `GL_EXT_texture_shared_exponent`).
- Framebuffer objects (`GL_EXT_framebuffer_object`).
- Half-float (16-bit) vertex array and pixel data formats (`GL_NV_half_float` and `GL_ARB_half_float_pixel`).
- Multisample stretch blit functionality (`GL_EXT_framebuffer_multisample` and `GL_EXT_framebuffer_blit`).
- Non-normalized integer color internal formats for textures and renderbuffers (`GL_EXT_texture_integer`).
- One- and two-dimensional layered texture targets (`GL_EXT_texture_array`).
- Packed depth/stencil internal formats for combined depth+stencil textures and renderbuffers (`GL_EXT_packed_depth_stencil`).
- Per-color-attachment blend enables and color writemasks (`GL_EXT_draw_buffers2`).
- RGTC specific internal compressed formats (`GL_EXT_texture_compression_rgtc`).
- Single- and double-channel (`R` and `RG`) internal formats for textures and renderbuffers.
- Transform feedback (`GL_EXT_transform_feedback`).
- Vertex array objects (`GL_APPLE_vertex_array_object`).
- sRGB framebuffer mode (`GL_EXT_framebuffer_sRGB`)

F.2 Deprecation Model

OpenGL 3.0 introduces a *deprecation model* in which certain features may be marked as *deprecated*. The deprecation model is described in detail in appendix E, together with a summary of features deprecated in OpenGL 3.0.

New Token Name	Old Token Name
`COMPARE_REF_TO_TEXTURE`	`COMPARE_R_TO_TEXTURE`
`MAX_VARYING_COMPONENTS`	`MAX_VARYING_FLOATS`
`MAX_CLIP_DISTANCES`	`MAX_CLIP_PLANES`
`CLIP_DISTANCE`i	`CLIP_PLANE`i

Table F.1: New token names and the old names they replace.

F.3 Changed Tokens

New token names are introduced to be used in place of old, inconsistent names. However, the old token names continue to be supported, for backwards compatibility with code written for previous versions of OpenGL. The new names, and the old names they replace, are shown in table F.1.

F.4 Change Log

Minor corrections to the OpenGL 3.0 Specification were made after its initial release.

Changes in the draft of September 23, 2008:

- Changed **ClearBuffer*** in section 4.2.3 to use `DEPTH` and `STENCIL` buffer names. Changed **GetFramebufferAttachmentParameteriv** in section 6.1.17 to accept only `DEPTH` and `STENCIL` to identify default framebuffer depth and stencil buffers, and only `DEPTH_ATTACHMENT` and `STENCIL_ATTACMENT` to identify framebuffer object depth and stencil buffers (bug 3744).

Changes in the draft of September 18, 2008:

- Added missing close-brace to **ArrayElement** pseudocode in section 2.8 (bug 3897).
- Noted in section 2.17 that **BeginQuery** will generate an `INVALID_OPERATION` error when called with an existing query object name whose type does not match the specified *target* (bug 3712).
- Add description of `gl_ClipDistance` to shader outputs in section 2.14.4 and note that only one of `gl_ClipVertex` and `gl_ClipDistance` should be written by a shader (bug 3898).

- Changed **ClearBuffer*** in section 4.2.3 to indirect through the draw buffer state by specifying the buffer type and draw buffer number, rather than the attachment name; also changed to accept `DEPTH_BUFFER` / `DEPTH_ATTACHMENT` and `STENCIL_BUFFER` / `STENCIL_ATTACHMENT` interchangeably, to reduce inconsistency between clearing the default framebuffer and framebuffer objects. Likewise changed **GetFramebufferAttachmentParameteriv** in section 6.1.17 to accept `DEPTH_BUFFER` / `DEPTH_ATTACHMENT` and `STENCIL_BUFFER` / `STENCIL_ATTACMENT` interchangeably (bug 3744).
- Add proper type suffix to query commands in tables 6.9 and 6.46 (Mark Kilgard).
- Update deprecation list in section E.2 to itemize deprecated state for two-sided color selection and include per-texture-unit LOD bias (bug 3735).

Changes in the draft of August 28, 2008:

- Sections 2.9, 2.9.3; tables 2.8, 2.9, and 6.12 - move buffer map/unmap calls into their own subsection and rewrite **MapBuffer** in terms of **MapBufferRange**. Add buffer state `BUFFER_ACCESS_FLAGS`, `BUFFER_MAP_OFFSET`, `BUFFER_MAP_LENGTH`. Make **MapBuffer** and **MapBufferRange** errors consistent (bug 3601).
- Section 2.10 - Extend `INVALID_OPERATION` error to **any** array pointer-setting command called to specify a client array while a vertex array object is bound, not just **VertexAttrib*Pointer** (bug 3696).
- Sections 2.16.1, 4.1.2, 4.2.1, and 4.3.4 - define initial state when a context is bound with no default framebuffer - null viewport and scissor region, draw buffer = read buffer = `NONE`, max viewport dims = max(display size - if any, max renderbuffer size). Viewport/scissor language added to the GLX and WGL create context extension specs as well (bug 2941).
- Section 2.19 - define "word-aligned" to be a multiple of 4 (e.g. 32 bits) (bug 3624).
- Section 5.4.1 - add **MapBufferRange** and **FlushMappedBufferRange** to commands not compiled in display lists (bug 3704).
- Section 6.1.14 - Moved **GetBufferParameteriv** query from section 6.1.3 and changed formal argument specifying the parameter name from *value* to *pname* (side effect of bug 3697).

- Section 6.1.17 - Moved **GetFramebufferAttachmentiv** query from section 6.1.3. Querying framebuffer attachment parameters other than object type and name when no attachment is present is an `INVALID_ENUM` error. Querying texture parameters (level, cube map face, or layer) for a renderbuffer attachment is also an `INVALID_ENUM` error (note that this was allowed in previous versions of the extension but the return values were not specified; it should clearly be an error as are other parameters that don't exist for the type of attachment present). Also reorganized the description of this command quite a bit to improve readability and remove redundancy and internal inconsistencies (bug 3697).

- Section 6.1.18 - Moved **GetRenderbufferParameteriv** query from section 6.1.3 (side effect of bug 3697).

- Appendix D.1 - add language to clarify that attachments to an object affect its reference count, and that object storage doesn't go away until there are no references remaining (bug 3725).

- Appendix E.2 - remove `TEXTURE_BORDER_COLOR` and `CLAMP_TO_BORDER` mode from the deprecated feature list; they were put in by accident (bug 3750).

- Appendix F - Cite `GL_EXT_texture_array` instead of `GL_EXT_geometry_shader4` as the source of 1D/2D array texture functionality. Fix a typo. Add change log relative to initial 3.0 spec release.

F.5 Credits and Acknowledgements

OpenGL 3.0 is the result of the contributions of many people and companies. Members of the Khronos OpenGL ARB Working Group during the development of OpenGL 3.0, including the company that they represented at the time of their contributions, follow. Some major contributions made by individuals are listed 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
Brian Paul, Tungsten Graphics
Bruce Merry, ARM (Detailed specification review)
Cass Everitt, NVIDIA
Chris Dodd, NVIDIA
Daniel Horowitz, NVIDIA
Daniel Koch, TransGaming (Framebuffer objects, half float vertex formats, and instanced rendering)
Daniel Omachi, Apple
Dave Shreiner, ARM
Eric Boumaour, AMD
Eskil Steenberg, Obsession
Evan Hart, NVIDIA
Folker Schamel, Spinor GMBH
Gavriel State, TransGaming
Geoff Stahl, Apple
Georg Kolling, Imagination Technologies
Gregory Prisament, NVIDIA
Guillaume Portier, HI Corp
Ian Romanick, IBM / Intel (Vertex array objects; GLX protocol)
James Helferty, TransGaming (Instanced rendering)
James Jones, NVIDIA
Jamie Gennis, NVIDIA
Jason Green, TransGaming
Jeff Bolz, NVIDIA
Jeff Juliano, NVIDIA
Jeremy Sandmel, Apple (Chair, ARB Nextgen (OpenGL 3.0) TSG)
John Kessenich, Intel (OpenGL Shading Language Specification Editor; deprecation model)
John Rosasco, Apple
Jon Leech, Independent (Chair, ARB Ecosystem TSG; OpenGL API Specification Editor; R/RG image formats and new context creation APIs)
Marc Olano, U. Maryland

Mark Callow, HI Corp
Mark Kilgard, NVIDIA (Many extensions on which OpenGL 3.0 features were based)
Matti Paavola, Nokia
Michael Gold, NVIDIA (Framebuffer objects and instanced rendering)
Neil Trevett, NVIDIA (President, Khronos Group)
Nick Burns, Apple
Nick Haemel, AMD
Pat Brown, NVIDIA (Many extensions on which OpenGL 3.0 features were based; detailed specification review)
Paul Martz, SimAuthor
Paul Ramsey, Sun
Pierre Boudier, AMD (Floating-point depth buffers)
Rob Barris, Blizzard (Framebuffer object and map buffer range)
Robert Palmer, Symbian
Robert Simpson, AMD
Steve Demlow, Vital Images
Thomas Roell, NVIDIA
Timo Suoranta, Futuremark
Tom Longo, AMD
Tom Olson, TI (Chair, Khronos OpenGL ES Working Group)
Travis Bryson, Sun
Yaki Tebeka, Graphic Remedy
Yanjun Zhang, S3 Graphics
Zack Rusin, Tungsten Graphics

The ARB gratefully acknowledges administrative support by the members of Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, and Michelle Clark, and technical support from James Riordon, webmaster of Khronos.org and OpenGL.org.

Appendix G

Version 3.1

OpenGL version 3.1, released on March 24, 2009, is the ninth revision since the original version 1.0.

Unlike earlier versions of OpenGL, OpenGL 3.1 is not upward compatible with earlier versions. The commands and interfaces identified as *deprecated* in OpenGL 3.0 (see appendix F) have been **removed** from OpenGL 3.1 entirely, with the following exception:

- Wide lines have not been removed, and calling **LineWidth** with values greater than 1.0 is not an error.

Implementations may restore such removed features using the `GL_ARB_compatibility` extension discussed in section G.2.

Following are brief descriptions of changes and additions to OpenGL 3.1.

G.1 New Features

New features in OpenGL 3.1, including the extension or extensions if any on which they were based, include:

- Support for OpenGL Shading Language 1.30 and 1.40.
- Instanced rendering with a per-instance counter accessible to vertex shaders (`GL_ARB_draw_instanced`).
- Data copying between buffer objects (`GL_ARB_copy_buffer`).
- Primitive restart (`GL_NV_primitive_restart`). Because client enable/disable no longer exists in OpenGL 3.1, the `PRIMITIVE_RESTART`

state has become server state, unlike the `NV` extension where it is client state. As a result, the numeric values assigned to `PRIMITIVE_RESTART` and `PRIMITIVE_RESTART_INDEX` differ from the `NV` versions of those tokens.

- At least 16 texture image units must be accessible to vertex shaders, in addition to the 16 already guaranteed to be accessible to fragment shaders.
- Texture buffer objects (`GL_ARB_texture_buffer_object`).
- Rectangular textures (`GL_ARB_texture_rectangle`).
- Uniform buffer objects (`GL_ARB_uniform_buffer_object`).
- Signed normalized texture component formats.

G.2 Deprecation Model

The features marked as deprecated in OpenGL 3.0 (see section E) have been removed from OpenGL 3.1 (with the exception of line widths greater than one, which are retained).

As described by the deprecation model, features removed from OpenGL 3.0 have been moved into the new extension `GL_ARB_compatibility`. If an implementation chooses to provide this extension, it restores all features deprecated by OpenGL 3.0 and removed from OpenGL 3.1. This extension may only be provided in an OpenGL 3.1 or later context version.

Because of the complexity of describing this extension relative to the OpenGL 3.1 core specification, it is not written up as a separate document, unlike other extensions in the extension registry. Instead, an alternate version of this specification document has been generated with the deprecated material still present, but marked in a distinct color.

No additional features are deprecated in OpenGL 3.1.

G.3 Change Log

Changes in the specification update of May 28, 2009:

- Update `MAX_CLIP_DISTANCES` from 6 to 8 in section 2.22 and table 6.51, to match GLSL (bug 4803).
- Accept null pointers in **CompressedTexImage*** (section 3.9.3) and treat them the same as for the corresponding **TexImage*** commands (bug 4863).

- Relax error conditions when specifying RGTC format texture images (section 3.9.2) and subimages (section 3.9.3) so that non-power-of-two RGTC images may be specified (also see section C.1), and edits to partial tiles at the edge of such an image made (bug 4856).

- Relaxed texture magnification switch-over point calculation in section 3.9.10 (bug 4392).

- Clarify initial value of stencil value masks in section 4.1.5 and table 6.26 (bug 4378).

- Change **FramebufferTextureLayer** in section 4.4.2 to generate `INVALID_VALUE` for negative *layer* only if *texture* is non-zero (bug 4084).

- Clarify **RenderbufferStorageMultisample** language in section 4.4.2 to allow, but not require creation of multisampled integer renderbuffers with more one sample (bug 4396).

- Added language to section 6.1.4 disallowing data-type format mismatches between internal and external texture formats in **GetTexImage** (bug 4163).

- Change initial value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE` in table 6.31 to `NONE` (bug 4407).

- Brought extension list in appendix I.3 up to date and correctly described extensions introduced along with OpenGL 3.0 and OpenGL 3.1 which implement subsets of new functionality in those versions to enable older hardware.

- Added missing contributors to the OpenGL 3.1 contributor list.

G.4 Credits and Acknowledgements

OpenGL 3.1 is the result of the contributions of many people and companies. Members of the Khronos OpenGL ARB Working Group during the development of OpenGL 3.1, including the company that they represented at the time of their contributions, follow. Some major contributions made by individuals are listed together with their name, including specific functionality developed in the form of new ARB extensions together with OpenGL 3.1. In addition, many people participated in developing earlier vendor and `EXT` extensions on which the OpenGL 3.1 functionality is based in part; those individuals are listed in the respective extension specifications in the OpenGL Extension Registry.

Alexis Mather, AMD (Chair, ARB Marketing TSG)
Avi Shapira, Graphic Remedy
Barthold Lichtenbelt, NVIDIA (Chair, Khronos OpenGL ARB Working Group)
Benjamin Lipchak, Apple (Uniform buffer objects)
Bill Licea-Kane, AMD (Chair, ARB Shading Language TSG; signed normalized texture formats)
Brent Insko, Intel
Brian Paul, Tungsten Graphics
Bruce Merry, ARM (Detailed specification review)
Christopher Webb, NVIDIA
Daniel Koch, TransGaming
Daniel Omachi, Apple
Eric Werness, NVIDIA
Gavriel State, TransGaming
Geoff Stahl, Apple
Gregory Roth, NVIDIA
Ian Romanick, Intel
James Helferty, TransGaming
James Jones, NVIDIA
Jeff Bolz, NVIDIA (Buffer to buffer copies)
Jeremy Sandmel, Apple (Chair, ARB Nextgen (OpenGL 3.1) TSG; uniform buffer objects)
John Kessenich, Intel (OpenGL Shading Language Specification Editor)
John Rosasco, Apple (Uniform buffer objects)
Jon Leech, Independent (OpenGL API Specification Editor)
Mark Callow, HI Corp
Mark Kilgard, NVIDIA (Many extensions on which OpenGL 3.0 features were based)
Matt Craighead, NVIDIA
Michael Gold, NVIDIA
Neil Trevett, NVIDIA (President, Khronos Group)
Nick Haemel, AMD
Pat Brown, NVIDIA (Many extensions on which OpenGL 3.0 features were based; detailed specification review)
Paul Martz, SimAuthor
Pierre Boudier, AMD
Rob Barris, Blizzard
Tom Olson, TI (Chair, Khronos OpenGL ES Working Group)
Yaki Tebeka, Graphic Remedy
Yanjun Zhang, S3 Graphics

The ARB gratefully acknowledges administrative support by the members of Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, and Michelle Clark, and technical support from James Riordon, webmaster of Khronos.org and OpenGL.org.

Appendix H

Version 3.2

OpenGL version 3.2, released on August 3, 2009, is the tenth revision since the original version 1.0.

Separate versions of the OpenGL 3.2 Specification exist for the *core* and *compatibility* profiles described in appendix E, respectively subtitled the "Core Profile" and the "Compatibility Profile". This document describes the Compatibility Profile. An OpenGL 3.2 implementation *must* be able to create a context supporting the core profile, and may also be able to create a context supporting the compatibility profile.

Material specific to the compatibility profile specification is marked in a distinct color to clearly call out differences between the two profiles.

The OpenGL 3.2 core profile is upward compatible with OpenGL 3.1, but not with earlier versions (see appendices G and F).

The OpenGL 3.2 compatibility profile is upward compatible with the combination of OpenGL 3.1 and the `GL_ARB_compatibility` extension, as well as with all earlier versions of OpenGL.

Following are brief descriptions of changes and additions to OpenGL 3.2.

H.1 New Features

New features in OpenGL 3.2, including the extension or extensions if any on which they were based, include:

- Introduction of core and compatibility profiles, superseding the `GL_ARB_compatibility` extension introduced with OpenGL 3.1.
- Support for OpenGL Shading Language 1.50.

- BGRA vertex component ordering (`GL_ARB_vertex_array_bgra`).
- Drawing commands allowing modification of the base vertex index (`GL_ARB_draw_elements_base_vertex`).
- Shader fragment coordinate convention control (`GL_ARB_fragment_coord_conventions`).
- Provoking vertex control (`GL_ARB_provoking_vertex`).
- Seamless cube map filtering (`GL_ARB_seamless_cube_map`).
- Multisampled textures and texture samplers for specific sample locations (`GL_ARB_texture_multisample`).
- Fragment depth clamping (`GL_ARB_depth_clamp`).
- Geometry shaders (`GL_ARB_geometry_shader4`).
- Fence sync objects (`GL_ARB_sync`).

H.2 Deprecation Model

In addition to restoring features removed from OpenGL 3.1 core, the compatibility profile defines some additional interactions between those features and new features introduced by OpenGL 3.2.

The following features are newly deprecated by the OpenGL 3.2 core profile:

- Global component limit query - the implementation-dependent values `MAX_VARYING_COMPONENTS` and `MAX_VARYING_FLOATS`.

No features are deprecated by the OpenGL 3.2 compatibility profile.

H.3 Changed Tokens

New token names are introduced to be used in place of old, less general names. However, the old token names continue to be supported, for backwards compatibility with code written for previous versions of OpenGL. The new names, and the old names they replace, are shown in table H.1.

New Token Name	Old Token Name
`PROGRAM_POINT_SIZE`	`VERTEX_PROGRAM_POINT_SIZE`

Table H.1: New token names and the old names they replace.

H.4 Change Log

Minor corrections to the OpenGL 3.2 Specification were made after its initial release in the update of December 7, 2009:

- Clean up description of GL command syntax in section 2.3, generalize the list of object types in the introduction to section 2.5 instead of enumerating them all redundantly, add half float to the **ArrayElement** pseudocode in section 2.8, fix **BindBuffer** (section 2.9.1), **BeginQuery** (section 2.17), and **BindTexture** (section 3.9.14) to only generate errors for user generated names in the core profile, remove P from computation of normalized device coordinates in section 2.16, increase minimum number of clip half-spaces to 8 in section 2.22, correct labelling of fragment processing selection in figure 3.1 for the compatibility profile, improve formatting and correct column heading in tables 6.20 and 6.21, and a variety of minor editorial corrections not enumerated here (Bug 5761).

- Remove “just as if they were unused” from description of names generated by **GenVertexArrays**, **GenFramebuffers**, and **GenRenderbuffers** in sections 2.10, 4.4.1, and 4.4.2 (Bug 5201).

- Fix duplicate label formerly applied to sections 2.12 and 2.16 (Bug 5455).

- Moved error language resulting from trying to put client pointers into VAOs from section 2.10 to section 2.8 (Bug 3975). This results in a forward reference, unfortunately.

- Remove reference to borders from texel fetch language in section 2.14.4 of the core specification (Bug 5343).

- Remove `INVALID_VALUE` error when passing a program object to **TransformFeedbackVaryings** in section 2.14.6 (Bug 5661).

- Changed number of query types from two to three in section 2.17 (Bug 5624).

- Change flat-shading source value description from "generic attribute" to "varying" in sections 3.5.1 and 3.6.1 (Bug 5359).
- Remove leftover references in core spec sections 3.9.3 and 6.1.3 to deprecated texture border state (Bug 5579). Still need to fix gl3.h accordingly.
- Fix typo in second paragraph of section 3.9.6 (Bug 5625).
- Simplify and clean up equations in the coordinate wrapping and mipmapping calculations of section 3.9.9, especially in the core profile where wrap mode `CLAMP` does not exist (Bug 5615).
- Fix computation of $u(x, y)$ and $v(x, y)$ in scale factor calculations of section 3.9.9 for rectangular textures (Bug 5700).
- Restructure definition of texture completeness in section 3.9.12 to separate mipmap consistency from filter requirements and cover new texture target types, and simplify how completness applies to texture fetches (section 2.14.4) and lookups (sections 2.14.4 and 3.12.2) (Bugs 4264, 5749).
- Update sampling language in sections 3.9.12, 2.14.4, and 3.12.2 to not require texture completeness when non-mipmapped access to the texture is being done (Bug 4264, based on ES bugs 4282 and 3499).
- Add fixed sample location state for multisample textures to section 3.9.13 (Bug 5454).
- Don't use the sign of the input component in the description of dithering in section 4.1.10 (Bug 5594).
- Change error condition for certain invalid buffers to **DrawBuffers** in section 4.2.1 from `INVALID_OPERATION` to `INVALID_ENUM` (Bug 5576).
- Clarify error conditions in section 4.2.3 when the clear mask is zero or contains invalid bits (Bug 5567).
- Change **BlitFramebuffer** in section 4.3.3 so format conversion is supported only within the three equivalence classes of fixed-point and floating point buffers, unsigned integer buffers, and signed integer buffers (Bug 5577).
- Include **ClientWaitSync**, **FenceSync**, and **PrimitiveRestartIndex** in the commands not compiled into display lists in section 5.4.1 (for the compatibility spec only) (Bug 5091).

- Remove a reference to unreachable `INVALID_OPERATION` errors from the core profile only in section 6.1.2 (Bug 5365).
- Specify that compressed texture component type queries in section 6.1.3 return how components are interpreted after decompression (Bug 5453).
- Increase value of `MAX_UNIFORM_BUFFER_BINDINGS` and `MAX_COMBINED_UNIFORM_BLOCKS` in table 6.58 from 24 to 36 to reflect addition of geometry shaders (Bug 5607).
- Update sharing rule 2 in appendix D.3.3 to read sensibly (Bug 5397).
- Update sharing rule 4 in appendix D.3.3 to cover the case where an object is only attached or bound in a single place, clarify comments about transform feedback, and state that reattachment is required to guarantee seeing changes made in other contexts, but does not preclude implementations from making changes visible without reattachment (Bugs 5546, 5777).

Changes in the specification for public release on August 3, 2009:

- Public release of OpenGL 3.2.

H.5 Credits and Acknowledgements

OpenGL 3.2 is the result of the contributions of many people and companies. Members of the Khronos OpenGL ARB Working Group during the development of OpenGL 3.2, including the company that they represented at the time of their contributions, follow. Some major contributions made by individuals are listed together with their name, including specific functionality developed in the form of new ARB extensions together with OpenGL 3.1. In addition, many people participated in developing earlier vendor and `EXT` extensions on which the OpenGL 3.1 functionality is based in part; those individuals are listed in the respective extension specifications in the OpenGL Extension Registry.

Aaftab Munshi, Apple
Acorn Pooley, NVIDIA
Barthold Lichtenbelt, NVIDIA (Chair, Khronos OpenGL ARB Working Group)
Benjamin Lipchak, Apple
Bill Licea-Kane, AMD (Chair, ARB Shading Language TSG)
Bruce Merry, ARM (Detailed specification review)
Cynthia Allison, NVIDIA

Daniel Koch, TransGaming (base vertex offset drawing, fragment coordinate conventions, provoking vertex control, BGRA attribute component ordering)
Dave Shreiner, ARM
David Garcia, AMD
Gavriel State, TransGaming
Geoff Stahl, Apple
Graham Sellers, AMD (seamless cube maps)
Gregory Roth, NVIDIA
Henri Verbeet, CodeWeavers
Ian Romanick, Intel
Jason Green, TransGaming
Jeff Bolz, NVIDIA (multisample textures)
Jeff Juliano, NVIDIA
Jeremy Sandmel, Apple (Chair, ARB Nextgen (OpenGL 3.2) TSG)
John Kessenich, Intel (OpenGL Shading Language Specification Editor)
Jon Leech, Independent (OpenGL API Specification Editor, fence sync objects)
Marcus Steyer, NVIDIA
Mark Callow, HI Corp
Mark Kilgard, NVIDIA (Many extensions on which OpenGL 3.2 features were based, including depth clamp, fragment coordinate conventions, provoking vertex control, and BGRA attribute component ordering)
Mark Krenek, Aspyr
Michael Gold, NVIDIA
Neil Trevett, NVIDIA (President, Khronos Group)
Nicholas Vining, Destineer
Nick Haemel, AMD
Pat Brown, NVIDIA (Many extensions on which OpenGL 3.0 features were based; detailed specification review)
Patrick Doane, Blizzard
Paul Martz, Skew Matrix
Pierre Boudier, AMD
Rob Barris, Blizzard
Ryan Gordon, Destineer
Stefan Dosinger, CodeWeavers
Yanjun Zhang, S3 Graphics

The ARB gratefully acknowledges administrative support by the members of Gold Standard Group, including Andrew Riegel, Elizabeth Riegel, Glenn Fredericks, and Michelle Clark, and technical support from James Riordon, webmaster of Khronos.org and OpenGL.org.

Appendix I

Extension Registry, Header Files, and ARB Extensions

I.1 Extension Registry

Many extensions to the OpenGL API have been defined by vendors, groups of vendors, and the OpenGL ARB. In order not to compromise the readability of the GL Specification, such extensions are not integrated into the core language; instead, they are made available online in the *OpenGL Extension Registry*, together with extensions to window system binding APIs, such as GLX and WGL, and with specifications for OpenGL, GLX, and related APIs.

Extensions are documented as changes to a particular version of the Specification. The Registry is available on the World Wide Web at URL

http://www.opengl.org/registry/

I.2 Header Files

Historically, C and C++ source code calling OpenGL was to `#include` a single header file, `<GL/gl.h>`. In addition to the core OpenGL API, the APIs for all extensions provided by an implementation were defined in this header.

When platforms became common where the OpenGL SDK (library and header files) were not necessarily obtained from the same source as the OpenGL driver, such as Microsoft Windows and Linux, `<GL/gl.h>` could not always be kept in sync with new core API versions and extensions supported by drivers. At this time the OpenGL ARB defined a new header, `<GL/glext.h>`, which could be obtained directly from the OpenGL Extension Registry (see section I.1). The combination of `<GL/gl.h>` and `<GL/glext.h>` always defines all APIs for all profiles of the latest OpenGL version, as well as for all extensions defined in the Registry.

`<GL3/gl3.h>` defines APIs for the core profile of OpenGL 3.2, together with ARB extensions compatible with the core profile. It does not include APIs for features only in the compatibility profile or for other extensions.

`<GL3/gl3ext.h>` defines APIs for additional ARB, EXT, and vendor extensions compatible with the core profile, but not defined in `<GL3/gl3.h>`. Most older extensions are not compatible with the core profile.

Applications using the OpenGL 3.2 compatibility profile (see appendices H and E) should `#include` the traditional `<GL/gl.h>` and `<GL/glext.h>` headers.

Applications using the core profile should `#include` the new `<GL3/gl3.h>` and `<GL3/gl3ext.h>` headers introduced with OpenGL 3.1.

By using `<GL3/gl3.h>` and `<GL3/gl3ext.h>`, instead of the legacy `<GL/gl.h>` and `<GL/glext.h>`, newly developed applications are given increased protection against accidentally using a “legacy” feature that has been removed from the core profile This can assist in developing applications on a GL implementation that supports the compatibility profile when the application is also intended to run on other platforms supporting only the core profile.

Developers should always be able to download `<GL3/gl3.h>` and `<GL3/gl3ext.h>` from the Registry, with these headers replacing, or being used in place of older versions that may be provided by a platform SDK.

I.3 ARB Extensions

OpenGL extensions that have been approved by the OpenGL Architectural Review Board (ARB) are summarized in this section. ARB extensions are not required to be supported by a conformant OpenGL implementation, but are expected to be widely available; they define functionality that is likely to move into the required feature set in a future revision of the specification.

I.3.1 Naming Conventions

To distinguish ARB extensions from core OpenGL features and from vendor-specific extensions, the following naming conventions are used:

- A unique *name string* of the form ”`GL_ARB_`*`name`*” is associated with each extension. If the extension is supported by an implementation, this string will be present in the `EXTENSIONS` string returned by **GetString**, and will be among the `EXTENSIONS` strings returned by **GetStringi**, as described in section 6.1.4.

- All functions defined by the extension will have names of the form ***Function*ARB**

- All enumerants defined by the extension will have names of the form *NAME*`_ARB`.

- In additional to OpenGL extensions, there are also ARB extensions to the related GLX and WGL APIs. Such extensions have name strings prefixed by `"GLX_"` and `"WGL_"` respectively. Not all GLX and WGL ARB extensions are described here, but all such extensions are included in the registry.

I.3.2 Promoting Extensions to Core Features

ARB extensions can be *promoted* to required core features in later revisions of OpenGL. When this occurs, the extension specifications are merged into the core specification. Functions and enumerants that are part of such promoted extensions will have the **ARB** affix removed.

GL implementations of such later revisions should continue to export the name strings of promoted extensions in the `EXTENSIONS` 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 the corresponding version of the OpenGL specification, or the descriptions of that version in version-specific appendices to later versions of the specification.

I.3.3 Multitexture

The name string for multitexture is `GL_ARB_multitexture`. It was promoted to a core feature in OpenGL 1.3.

I.3.4 Transpose Matrix

The name string for transpose matrix is `GL_ARB_transpose_matrix`. It was promoted to a core feature in OpenGL 1.3.

I.3.5 Multisample

The name string for multisample is `GL_ARB_multisample`. It was promoted to a core feature in OpenGL 1.3.

I.3.6 Texture Add Environment Mode

The name string for texture add mode is `GL_ARB_texture_env_add`. It was promoted to a core feature in OpenGL 1.3.

I.3.7 Cube Map Textures

The name string for cube mapping is `GL_ARB_texture_cube_map`. It was promoted to a core feature in OpenGL 1.3.

I.3.8 Compressed Textures

The name string for compressed textures is `GL_ARB_texture_compression`. It was promoted to a core feature in OpenGL 1.3.

I.3.9 Texture Border Clamp

The name string for texture border clamp is `GL_ARB_texture_border_clamp`. It was promoted to a core feature in OpenGL 1.3.

I.3.10 Point Parameters

The name string for point parameters is `GL_ARB_point_parameters`. It was promoted to a core features in OpenGL 1.4.

I.3.11 Vertex Blend

Vertex blending replaces the single model-view transformation with multiple vertex units. Each unit has its own transform matrix and an associated current weight. Vertices are transformed by all the enabled units, scaled by their respective weights, and summed to create the eye-space vertex. Normals are similarly transformed by the inverse transpose of the model-view matrices.

The name string for vertex blend is `GL_ARB_vertex_blend`.

I.3.12 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`.

I.3.13 Texture Combine Environment Mode

The name string for texture combine mode is `GL_ARB_texture_env_combine`. It was promoted to a core feature in OpenGL 1.3.

I.3.14 Texture Crossbar Environment Mode

The name string for texture crossbar is `GL_ARB_texture_env_crossbar`. It was promoted to a core features in OpenGL 1.4.

I.3.15 Texture Dot3 Environment Mode

The name string for DOT3 is `GL_ARB_texture_env_dot3`. It was promoted to a core feature in OpenGL 1.3.

I.3.16 Texture Mirrored Repeat

The name string for texture mirrored repeat is `GL_ARB_texture_mirrored_repeat`. It was promoted to a core feature in OpenGL 1.4.

I.3.17 Depth Texture

The name string for depth texture is `GL_ARB_depth_texture`. It was promoted to a core feature in OpenGL 1.4.

I.3.18 Shadow

The name string for shadow is `GL_ARB_shadow`. It was promoted to a core feature in OpenGL 1.4.

I.3.19 Shadow Ambient

Shadow ambient extends the basic image-based shadow functionality by allowing a texture value specified by the `TEXTURE_COMPARE_FAIL_VALUE_ARB` texture parameter to be returned when the texture comparison fails. This may be used for ambient lighting of shadowed fragments and other advanced lighting effects.

The name string for shadow ambient is `GL_ARB_shadow_ambient`.

I.3.20 Window Raster Position

The name string for window raster position is `GL_ARB_window_pos`. It was promoted to a core feature in OpenGL 1.4.

I.3.21 Low-Level Vertex Programming

Application-defined *vertex programs* may be specified in a new low-level programming language, replacing the standard fixed-function vertex transformation, lighting, and texture coordinate generation pipeline. Vertex programs enable many new effects and are an important first step towards future graphics pipelines that will be fully programmable in an unrestricted, high-level shading language.

The name string for low-level vertex programming is `GL_ARB_vertex_program`.

I.3.22 Low-Level Fragment Programming

Application-defined *fragment programs* may be specified in the same low-level language as `GL_ARB_vertex_program`, replacing the standard fixed-function vertex texturing, fog, and color sum operations.

The name string for low-level fragment programming is `GL_ARB_fragment_program`.

I.3.23 Buffer Objects

The name string for buffer objects is `GL_ARB_vertex_buffer_object`. It was promoted to a core feature in OpenGL 1.5.

I.3.24 Occlusion Queries

The name string for occlusion queries is `GL_ARB_occlusion_query`. It was promoted to a core feature in OpenGL 1.5.

I.3.25 Shader Objects

The name string for shader objects is `GL_ARB_shader_objects`. It was promoted to a core feature in OpenGL 2.0.

I.3.26 High-Level Vertex Programming

The name string for high-level vertex programming is `GL_ARB_vertex_shader`. It was promoted to a core feature in OpenGL 2.0.

I.3.27 High-Level Fragment Programming

The name string for high-level fragment programming is `GL_ARB_fragment_shader`. It was promoted to a core feature in OpenGL 2.0.

I.3.28 OpenGL Shading Language

The name string for the OpenGL Shading Language is `GL_ARB_shading_language_100`. The presence of this extension string indicates that programs written in version 1 of the Shading Language are accepted by OpenGL. It was promoted to a core feature in OpenGL 2.0.

I.3.29 Non-Power-Of-Two Textures

The name string for non-power-of-two textures is `GL_ARB_texture_non_power_of_two`. It was promoted to a core feature in OpenGL 2.0.

I.3.30 Point Sprites

The name string for point sprites is `GL_ARB_point_sprite`. It was promoted to a core feature in OpenGL 2.0.

I.3.31 Fragment Program Shadow

Fragment program shadow extends low-level fragment programs defined with `GL_ARB_fragment_program` to add shadow 1D, 2D, and 3D texture targets, and remove the interaction with `GL_ARB_shadow`.

The name string for fragment program shadow is `GL_ARB_fragment_program_shadow`.

I.3.32 Multiple Render Targets

The name string for multiple render targets is `GL_ARB_draw_buffers`. It was promoted to a core feature in OpenGL 2.0.

I.3.33 Rectangular Textures

Rectangular textures define a new texture target `TEXTURE_RECTANGLE_ARB` that supports 2D textures without requiring power-of-two dimensions. Rectangular textures are useful for storing video images that do not have power-of-two sizes (POTS). Resampling artifacts are avoided and less texture memory may be required. They are are also useful for shadow maps and window-space texturing. These textures are accessed by dimension-dependent (aka non-normalized) texture coordinates.

Rectangular textures are a restricted version of non-power-of-two textures. The differences are that rectangular textures are supported only for 2D; they require a new texture target; and the new target uses non-normalized texture coordinates.

The name string for texture rectangles is `GL_ARB_texture_rectangle`. It was promoted to a core feature in OpenGL 3.1.

I.3.34 Floating-Point Color Buffers

Floating-point color buffers can represent values outside the normal $[0, 1]$ range of colors in the fixed-function OpenGL pipeline. This group of related extensions enables controlling clamping of vertex colors, fragment colors throughout the pipeline, and pixel data read back to client memory, and also includes WGL and GLX extensions for creating frame buffers with floating-point color components (referred to in GLX as *framebuffer configurations*, and in WGL as *pixel formats*).

The name strings for floating-point color buffers are `GL_ARB_color_buffer_float`, `GLX_ARB_fbconfig_float`, and `WGL_ARB_pixel_format_float`. `GL_ARB_color_buffer_float` was promoted to a core feature in OpenGL 3.0.

I.3.35 Half-Precision Floating Point

This extension defines the representation of a 16-bit floating point data format, and a corresponding *type* argument which may be used to specify and read back pixel and texture images stored in this format in client memory. Half-precision floats are smaller than full precision floats, but provide a larger dynamic range than similarly sized (`short`) data types.

The name string for half-precision floating point is `GL_ARB_half_float_pixel`. It was promoted to a core feature in OpenGL 3.0.

I.3.36 Floating-Point Textures

Floating-point textures stored in both 32- and 16-bit formats may be defined using new *internalformat* arguments to commands which specify and read back texture images.

The name string for floating-point textures is `GL_ARB_texture_float`. It was promoted to a core feature in OpenGL 3.0.

I.3.37 Pixel Buffer Objects

The buffer object interface is expanded by adding two new binding targets for buffer objects, the pixel pack and unpack buffers. This permits buffer objects to be used to store pixel data as well as vertex array data. Pixel-drawing and -reading commands using data in pixel buffer objects may operate at greatly improved performance compared to data in client memory.

The name string for pixel buffer objects is `GL_ARB_pixel_buffer_object`. It was promoted to a core feature in OpenGL 2.1.

I.3.38 Floating-Point Depth Buffers

The name string for floating-point depth buffers is `GL_ARB_depth_buffer_float`. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier `GL_NV_depth_buffer_float` extension, and is provided to enable this functionality in older drivers.

I.3.39 Instanced Rendering

The name string for instanced rendering is `GL_ARB_draw_instanced`. It was promoted to a core feature in OpenGL 3.1.

I.3.40 Framebuffer Objects

The name string for framebuffer objects is `GL_ARB_framebuffer_object`. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier `GL_EXT_framebuffer_object`, `GL_EXT_framebuffer_multisample`, and `GL_EXT_framebuffer_blit` extensions, and is provided to enable this functionality in older drivers.

I.3.41 sRGB Framebuffers

The name string for sRGB framebuffers is `GL_ARB_framebuffer_sRGB`. It was promoted to a core feature in OpenGL 3.0. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier `GL_EXT_framebuffer_sRGB` extension, and is provided to enable this functionality in older drivers.

To create sRGB format surface for use on display devices, an additional pixel format (config) attribute is required in the window system integration layer. The name strings for the GLX and WGL sRGB pixel format interfaces are `GLX_ARB_framebuffer_sRGB` and `WGL_ARB_framebuffer_sRGB` respectively.

I.3.42 Geometry Shaders

This extension defines a new shader type called a *geometry shader*. Geometry shaders are run after vertices are transformed, but prior to the remaining fixed-function vertex processing, and may generate new vertices for, or remove vertices from the primitive assembly process.

The name string for geometry shaders is `GL_ARB_geometry_shader4`.

I.3.43 Half-Precision Vertex Data

The name string for half-precision vertex data `GL_ARB_half_float_vertex`. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier `GL_NV_half_float` extension, and is provided to enable this functionality in older drivers.

I.3.44 Instanced Rendering

.

This instanced rendering interface is a less-capable form of `GL_ARB_draw_instanced` which can be supported on older hardware.

The name string for instance rendering is `GL_ARB_instanced_arrays`.

I.3.45 Flexible Buffer Mapping

The name string for flexible buffer mapping is `GL_ARB_map_buffer_range`. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier `GL_APPLE_flush_buffer_range` extension, and is provided to enable this functionality in older drivers.

I.3.46 Texture Buffer Objects

The name string for texture buffer objects is `GL_ARB_texture_buffer_object`. It was promoted to a core feature in OpenGL 3.1.

I.3.47 RGTC Texture Compression Formats

The name string for RGTC texture compression formats is `GL_ARB_texture_compression_rgtc`. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier `GL_EXT_texture_compression_rgtc` extension, and is provided to enable this functionality in older drivers.

It was promoted to a core feature in OpenGL 3.0.

I.3.48 One- and Two-Component Texture Formats

The name string for one- and two-component texture formats is `GL_ARB_texture_rg`. It was promoted to a core feature in OpenGL 3.0. This extension is equivalent to new core functionality introduced in OpenGL 3.0, and is provided to enable this functionality in older drivers.

I.3.49 Vertex Array Objects

The name string for vertex array objects is `GL_ARB_vertex_array_object`. This extension is equivalent to new core functionality introduced in OpenGL 3.0, based on the earlier `GL_APPLE_vertex_array_object` extension, and is provided to enable this functionality in older drivers.

It was promoted to a core feature in OpenGL 3.0.

I.3.50 Versioned Context Creation

Starting with OpenGL 3.0, a new context creation interface is required in the window system integration layer. This interface specifies the context version required as well as other attributes of the context.

The name strings for the GLX and WGL context creation interfaces are `GLX_ARB_create_context` and `WGL_ARB_create_context` respectively.

I.3.51 Uniform Buffer Objects

The name string for uniform buffer objects is `GL_ARB_uniform_buffer_object`. This extension is equivalent to new core functionality introduced in OpenGL 3.1 and is provided to enable this functionality in older drivers.

I.3.52 Restoration of features removed from OpenGL 3.0

OpenGL 3.1 removes a large number of features that were marked deprecated in OpenGL 3.0 (see appendix G.2). GL implementations needing to maintain these features to support existing applications may do so, following the deprecation model, by exporting an extension string indicating those features are present. Applications written for OpenGL 3.1 should not depend on any of the features corresponding to this extension, since they will not be available on all platforms with 3.1 implementations.

The name string for restoration of features deprecated by OpenGL 3.0 is `GL_ARB_compatibility`.

The profile terminology introduced with OpenGL 3.2 eliminates the necessity for evolving this extension. Instead, interactions between features removed by OpenGL 3.1 and new features introduced in later OpenGL versions are defined by the compatibility profile corresponding to those versions.

I.3.53 Fast Buffer-to-Buffer Copies

The name string for fast buffer-to-buffer copies is `GL_ARB_copy_buffer`. This extension is equivalent to new core functionality introduced in OpenGL 3.1 and is provided to enable this functionality in older drivers.

I.3.54 Shader Texture Level of Detail Control

The name string for shader texture level of detail control is `GL_ARB_shader_texture_lod`. This extension is equivalent to new core functions introduced in OpenGL Shading Language 1.30 and is provided to enable this functionality in older versions of the shading language.

I.3.55 Depth Clamp Control

The name string for depth clamp control is `GL_ARB_depth_clamp`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

I.3.56 Base Vertex Offset Drawing Commands

The name string for base vertex offset drawing commands is `GL_ARB_draw_elements_base_vertex`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

I.3.57 Fragment Coordinate Convention Control

The name string for fragment coordinate convention control is `GL_ARB_fragment_coord_conventions`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

I.3.58 Provoking Vertex Control

The name string for provoking vertex control is `GL_ARB_provoking_vertex`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

I.3.59 Seamless Cube Maps

The name string for seamless cube maps is `GL_ARB_seamless_cube_map`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

I.3.60 Fence Sync Objects

The name string for fence sync objects is `GL_ARB_sync`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

I.3.61 Multisample Textures

The name string for multisample textures is `GL_ARB_texture_multisample`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

I.3.62 BGRA Attribute Component Ordering

The name string for BGRA attribute component ordering is `GL_ARB_vertex_array_bgra`. This extension is equivalent to new core functionality introduced in OpenGL 3.2 and is provided to enable this functionality in older drivers.

I.3.63 Per-Buffer Blend Control

The blending interface is extended to specify blend equation and blend function on a per-draw-buffer basis.

The name string for per-buffer blend control is `GL_ARB_draw_buffers_blend`.

I.3.64 Sample Shading Control

Sample shading control adds the ability to request that an implementation use a minimum number of unique sets of fragment computation inputs when multisampling a pixel.

The name string for sample shading control is `GL_ARB_sample_shading`.

I.3.65 Cube Map Array Textures

A cube map array texture is a two-dimensional array texture that may contain many cube map layers. Each cube map layer is a unique cube map image set.

The name string for cube map array textures is `GL_ARB_texture_cube_map_array`.

I.3.66 Texture Gather

Texture gather adds a new set of texture functions (`textureGather`) to the OpenGL Shading Language that determine the 2×2 footprint used for linear filtering in a texture lookup, and return a vector consisting of the first component from each of the four texels in the footprint.

The name string for texture gather is `GL_ARB_texture_gather`.

I.3.67 Texture Level-Of-Detail Queries

Texture level-of-detail queries adds a new set of fragment shader texture functions (`textureLOD`) to the OpenGL Shading Language that return the results of automatic level-of-detail computations that would be performed if a texture lookup were to be done.

The name string for texture level-of-detail queries is `GL_ARB_texture_query_lod`.

I.3.68 Profiled Context Creation

Starting with OpenGL 3.2, API profiles are defined. Profiled context creation extends the versioned context creation interface to specify a profile which must be implemnented by the context.

The name strings for the GLX and WGL profiled context creation interfaces are `GLX_ARB_create_context_profile` and `WGL_ARB_create_context_profile` respectively.

Index

x, 431
x_BIAS, 172, 427
x_BITS, 452
x_SCALE, 172, 427
x_SIZE, 431
*BaseVertex, 40
*CopyBufferSubData, 56
*GetString, 379
*GetStringi, 379
*MapBuffer, 54
*MapBufferRange, 52
*Pointer, 35
*WaitSync, 382
-, 433–435, 444
–, 397, 398, 453
453
2D, 354, 355, 453
2_BYTES, 357
3D, 354, 355
3D_COLOR, 354, 355
3D_COLOR_TEXTURE, 354, 355
3_BYTES, 357
4D_COLOR_TEXTURE, 354, 355
4_BYTES, 357

1, 412
2, 412
3, 412

ACCUM, 307
Accum, 129, 146, 307, 308, 475
ACCUM_x_BITS, 452
ACCUM_*_BITS, 475
ACCUM_ALPHA_BITS, 323
ACCUM_BLUE_BITS, 323
ACCUM_BUFFER_BIT, 304, 394, 475
ACCUM_CLEAR_VALUE, 420
ACCUM_GREEN_BITS, 323
ACCUM_RED_BITS, 323
ACTIVE_ATTRIBUTE_MAX_LENGTH, 87, 385, 435
ACTIVE_ATTRIBUTES, 87, 385, 434
ACTIVE_TEXTURE, 31, 66, 140, 211, 212, 260, 346, 368, 416
ACTIVE_UNIFORM_BLOCK_MAX_NAME_LENGTH, 386, 436
ACTIVE_UNIFORM_BLOCKS, 92, 386, 436
ACTIVE_UNIFORM_MAX_LENGTH, 94, 95, 385, 434
ACTIVE_UNIFORMS, 94, 95, 385, 434
ActiveTexture, 106, 211, 212, 269
ADD, 261, 263, 264, 307
ADD_SIGNED, 264
ALIASED_LINE_WIDTH_RANGE, 445
ALIASED_POINT_SIZE_RANGE, 444
ALL_ATTRIB_BITS, 393, 394, 476
ALPHA, 172, 186, 201, 202, 214, 218, 221, 241, 242, 261–263, 266, 293, 312, 316, 336, 374, 415, 423, 427, 428, 430, 452, 474

ALPHA12, 218
ALPHA16, 218
ALPHA4, 218
ALPHA8, 216, 218
ALPHA_BIAS, 199
ALPHA_BITS, 341, 475
ALPHA_INTEGER, 186
ALPHA_SCALE, 198, 261, 417
ALPHA_TEST, 286, 418, 475
ALPHA_TEST_FUNC, 418
ALPHA_TEST_REF, 418
AlphaFunc, 286, 475
ALREADY_SIGNALED, 362
ALWAYS, 241, 267, 286–288, 418
AMBIENT, 77, 78, 80, 407, 408
AMBIENT_AND_DIFFUSE, 77, 78, 80
AND, 296
AND_INVERTED, 296
AND_REVERSE, 296
Antialiasing, 162
AreTexturesResident, 259, 260, 358, 475
ARRAY_BUFFER, 35, 48, 57
ARRAY_BUFFER_BINDING, 57, 403
ArrayElement, 30, 38–41, 57, 113, 114, 356, 479, 491
ATTACHED_SHADERS, 385, 386, 434
AttachShader, 84, 359
ATTRIB_STACK_DEPTH, 453, 476
AUTO_NORMAL, 110, 347, 432
AUXi, 298–300, 389, 475
AUX0, 298
AUX_BUFFERS, 298, 323, 452

BACK, 76, 77, 79, 163, 164, 166, 287, 290, 298, 299, 301–303, 305, 311, 321, 369, 410, 474
BACK_LEFT, 299, 300, 389
BACK_RIGHT, 299, 300, 389
Begin, 18, 22–25, 27–30, 39–41, 60, 77, 80, 105, 115, 118, 123, 125, 129, 131, 135, 156, 159, 162, 166, 277, 339, 340, 348, 349, 354, 472
BeginConditionalRender, 129, 130
BeginQuery, 128, 129, 133, 289, 479, 491
BeginTransformFeedback, 130–132
BGR, 186, 312, 316
BGR_INTEGER, 186
BGRA, 35–37, 186, 189, 194, 312
BGRA_INTEGER, 186
BindAttribLocation, 88, 89, 359
BindBuffer, 47–49, 58, 239, 358, 491
BindBufferBase, 49, 105, 131, 133, 358
BindBufferRange, 49, 105, 131–133, 358
BindFragDataLocation, 278, 279, 359
BindFramebuffer, 322, 324, 338, 358
BindRenderbuffer, 325, 326, 359
BindTexture, 106, 212, 257–259, 491
BindVertexArray, 59, 358
BITMAP, 165, 174, 177, 183, 185, 196, 208, 314, 375, 474
Bitmap, 105, 129, 146, 208, 209, 277, 342, 474
BITMAP_TOKEN, 355
BLEND, 261, 263, 290, 294, 296, 419
BLEND_COLOR, 419
BLEND_DST_ALPHA, 419
BLEND_DST_RGB (v1.3:BLEND_DST), 419
BLEND_EQUATION_ALPHA, 419
BLEND_EQUATION_RGB (v1.5: BLEND_EQUATION), 419
BLEND_SRC_ALPHA, 419
BLEND_SRC_RGB (v1.3:BLEND_SRC), 419
BlendColor, 294

BlendEquation, 290, 291
BlendEquationSeparate, 290, 291
BlendFunc, 292
BlendFuncSeparate, 292
BlitFramebuffer, 308, 318, 320, 335, 359, 492
BLUE, 172, 186, 312, 316, 415, 423, 427, 428, 430, 452
BLUE_BIAS, 199
BLUE_BITS, 341, 475
BLUE_INTEGER, 186
BLUE_SCALE, 198
BOOL, 96
bool, 96, 102
BOOL_VEC2, 96
BOOL_VEC3, 96
BOOL_VEC4, 96
BUFFER_ACCESS, 49, 51, 54, 404
BUFFER_ACCESS_FLAGS, 49, 51, 54, 55, 404, 480
BUFFER_MAP_LENGTH, 49, 51, 54, 55, 404, 480
BUFFER_MAP_OFFSET, 49, 51, 54, 55, 404, 480
BUFFER_MAP_POINTER, 49, 51, 54, 55, 383, 404
BUFFER_MAPPED, 49, 51, 54, 55, 404
BUFFER_SIZE, 49, 51, 53, 54, 404
BUFFER_USAGE, 49, 51, 52, 404
BufferData, 50, 51, 90, 358, 467
BufferSubData, 51, 90, 358, 467
bvec2, 96, 100
bvec3, 96
bvec4, 96
BYTE, 35, 185, 314, 315, 357

C3F_V3F, 44, 45
C4F_N3F_V3F, 44, 45
C4UB_V2F, 44, 45
C4UB_V3F, 44, 45
CallList, 30, 356, 357, 476
CallLists, 30, 357, 476
CCW, 163, 410
CheckFramebufferStatus, 338, 339, 358
CLAMP, 241, 247, 248, 474, 492
CLAMP_FRAGMENT_COLOR, 197, 320, 406, 473
CLAMP_READ_COLOR, 313, 406
CLAMP_TO_BORDER, 241, 243, 248, 481
CLAMP_TO_EDGE, 241, 243, 248, 256, 319
CLAMP_VERTEX_COLOR, 81, 406, 473
ClampColor, 81, 197, 313, 473
CLEAR, 296
Clear, 129, 146, 303, 305, 306, 343, 475
ClearAccum, 304, 475
ClearBuffer, 306
ClearBuffer*, 129, 146, 343, 479, 480
ClearBuffer{if ui}v, 305, 306
ClearBufferfi, 306
ClearBufferfv, 305, 306
ClearBufferiv, 305, 306
ClearBufferuiv, 305
ClearColor, 304, 305
ClearDepth, 304–306
ClearIndex, 304
ClearStencil, 304–306
CLIENT_ACTIVE_TEXTURE, 37, 368, 403
CLIENT_ALL_ATTRIB_BITS, 393, 394, 476
CLIENT_ATTRIB_STACK_DEPTH, 453, 476
CLIENT_PIXEL_STORE_BIT, 394
CLIENT_VERTEX_ARRAY_BIT, 394
ClientActiveTexture, 30, 37, 358, 472
ClientWaitSync, 359, 361–364, 466, 492

CLIP_DISTANCEi, 137, 405, 479
CLIP_DISTANCE0, 137
CLIP_PLANEi, 136, 405, 479
ClipPlane, 136, 473
COEFF, 370, 432
COLOR, 62, 66, 176, 180, 181, 227, 305, 306, 316
Color, 29, 32, 70, 80, 86, 129, 140
Color*, 472
Color3, 32
Color4, 32
Color[size][type]v, 38
COLOR_ARRAY, 37, 46, 399
COLOR_ARRAY_BUFFER_BINDING, 402
COLOR_ARRAY_POINTER, 378, 399
COLOR_ARRAY_SIZE, 399
COLOR_ARRAY_STRIDE, 399
COLOR_ARRAY_TYPE, 399
COLOR_ATTACHMENTi, 298, 299, 311, 312, 329, 337
COLOR_ATTACHMENTm, 299, 301
COLOR_ATTACHMENTn, 323
COLOR_ATTACHMENT0, 298, 301, 312, 323
COLOR_BUFFER_BIT, 304, 306, 318, 320, 394
COLOR_CLEAR_VALUE, 420
COLOR_INDEX, 165, 174, 177, 183, 186, 196, 199, 208, 312, 316, 373, 375, 472
COLOR_INDEXES, 78, 81, 408
COLOR_LOGIC_OP, 296, 419
COLOR_MATERIAL, 77, 80, 407, 473
COLOR_MATERIAL_FACE, 407
COLOR_MATERIAL_PARAMETER, 407
COLOR_MATRIX, 375
COLOR_MATRIX (TRANSPOSE_COLOR_MATRIX), 405
COLOR_MATRIX_STACK_DEPTH, 375, 405
COLOR_SUM, 271, 406, 475
COLOR_TABLE, 174, 177, 200, 428
COLOR_TABLE_x_SIZE, 428
COLOR_TABLE_ALPHA_SIZE, 375
COLOR_TABLE_BIAS, 174, 175, 375, 428
COLOR_TABLE_BLUE_SIZE, 375
COLOR_TABLE_FORMAT, 375, 428
COLOR_TABLE_GREEN_SIZE, 375
COLOR_TABLE_INTENSITY_SIZE, 375
COLOR_TABLE_LUMINANCE_SIZE, 375
COLOR_TABLE_RED_SIZE, 375
COLOR_TABLE_SCALE, 174–176, 375, 428
COLOR_TABLE_WIDTH, 375, 428
COLOR_WRITEMASK, 302, 420
ColorMask, 302, 303
ColorMaski, 302
ColorMaterial, 77, 79, 80, 347, 459, 473
ColorPointer, 30, 34, 36, 46, 358, 472
ColorSubTable, 170, 176
ColorTable, 170, 174–178, 181, 182, 205, 206, 359
ColorTableParameter, 175
ColorTableParameterfv, 174
Colorub, 139
Colorui, 139
Colorus, 139
COMBINE, 261, 264, 269
COMBINE_ALPHA, 261, 264, 265, 416
COMBINE_RGB, 261, 264, 265, 416
COMPARE_R_TO_TEXTURE, 479
COMPARE_REF_TO_TEXTURE, 241, 266, 479

COMPILE, 356, 459
COMPILE_AND_EXECUTE, 356, 357
COMPILE_STATUS, 83, 385, 433
CompileShader, 83, 359
COMPRESSED_ALPHA, 221
COMPRESSED_INTENSITY, 221
COMPRESSED_LUMINANCE, 221
COMPRESSED_LUMINANCE_ALPHA, 221
COMPRESSED_RED, 221
COMPRESSED_RED_RGTC1, 216, 221, 462–464
COMPRESSED_RG, 221
COMPRESSED_RG_RGTC2, 216, 221, 464
COMPRESSED_RGB, 221
COMPRESSED_RGBA, 221
COMPRESSED_SIGNED_RED_RGTC1, 216, 221, 463, 464
COMPRESSED_SIGNED_RG_RGTC2, 216, 221, 464
COMPRESSED_SLUMINANCE, 221, 267
COMPRESSED_SLUMINANCE_ALPHA, 221, 267
COMPRESSED_SRGB, 221, 267
COMPRESSED_SRGB_ALPHA, 221, 267
COMPRESSED_TEXTURE_FORMATS, 215, 445
CompressedTexImage, 235
CompressedTexImage$n$D, 233
CompressedTexImage*, 339, 485
CompressedTexImage1D, 232–235
CompressedTexImage2D, 232–235
CompressedTexImage3D, 232–235
CompressedTexSubImage$n$D, 235
CompressedTexSubImage1D, 234–236
CompressedTexSubImage2D, 234–236
CompressedTexSubImage3D, 234–236
CONDITION_SATISFIED, 362
CONSTANT, 263, 265, 417
CONSTANT_ALPHA, 293
CONSTANT_ATTENUATION, 78, 408
CONSTANT_BORDER, 202–204
CONSTANT_COLOR, 293
CONTEXT_COMPATIBILITY_PROFILE_BIT, 379
CONTEXT_CORE_PROFILE_BIT, 379
CONTEXT_FLAG_FORWARD_COMPATIBLE_BIT, 379
CONTEXT_FLAGS, 379, 446
CONTEXT_PROFILE_MASK, 379
CONVOLUTION_xD, 429
CONVOLUTION_1D, 178–180, 200, 225, 376, 429
CONVOLUTION_2D, 177–179, 201, 224, 376, 429
CONVOLUTION_BORDER_COLOR, 203, 376, 429
CONVOLUTION_BORDER_MODE, 202, 376, 429
CONVOLUTION_FILTER_BIAS, 177–179, 376, 429
CONVOLUTION_FILTER_SCALE, 177–180, 376, 429
CONVOLUTION_FORMAT, 376, 429
CONVOLUTION_HEIGHT, 376, 429
CONVOLUTION_WIDTH, 376, 429
ConvolutionFilter1D, 170, 178–180
ConvolutionFilter2D, 170, 177–180
ConvolutionParameter, 178, 202
ConvolutionParameterfv, 178, 203
ConvolutionParameteriv, 179, 203
COORD_REPLACE, 151, 154, 416
COPY, 296, 297, 419
COPY_INVERTED, 296
COPY_PIXEL_TOKEN, 355

COPY_READ_BUFFER, 48, 56, 453
COPY_WRITE_BUFFER, 48, 56, 453
CopyColorSubTable, 176, 318
CopyColorTable, 175, 176, 318
CopyConvolutionFilter*, 318
CopyConvolutionFilter1D, 180
CopyConvolutionFilter2D, 179, 180
CopyPixels, 129, 146, 169, 172, 175, 176, 179, 180, 201, 308, 316–320, 342, 343, 353, 475
CopyTexImage, 340, 475
CopyTexImage*, 318, 329, 335, 339
CopyTexImage1D, 201, 228, 229, 232, 251
CopyTexImage2D, 201, 227–229, 232, 251
CopyTexImage3D, 229
CopyTexSubImage, 340
CopyTexSubImage*, 232, 236, 318, 329
CopyTexSubImage1D, 201, 228, 229, 231, 232
CopyTexSubImage2D, 201, 228–232
CopyTexSubImage3D, 201, 228, 229, 231, 232
CreateProgram, 84, 359
CreateShader, 83, 359
CULL_FACE, 163, 410
CULL_FACE_MODE, 410
CullFace, 163, 164, 168
CURRENT_BIT, 394
CURRENT_COLOR, 398
CURRENT_FOG_COORD, 398
CURRENT_INDEX, 398
CURRENT_NORMAL, 398
CURRENT_PROGRAM, 434
CURRENT_QUERY, 380, 453
CURRENT_RASTER_COLOR, 398
CURRENT_RASTER_DISTANCE, 398
CURRENT_RASTER_INDEX, 398
CURRENT_RASTER_POSITION, 398
CURRENT_RASTER_POSITION_VALID, 398
CURRENT_RASTER_SECONDARY_COLOR, 398
CURRENT_RASTER_TEXTURE_COORDS, 140, 398, 458
CURRENT_SECONDARY_COLOR, 398
CURRENT_TEXTURE_COORDS, 31, 398
CURRENT_VERTEX_ATTRIB, 388, 438
CW, 163

DECAL, 261, 262
DECR, 287
DECR_WRAP, 287
DELETE_STATUS, 84, 385, 433, 434
DeleteBuffers, 47, 48, 358, 465
DeleteFramebuffers, 322, 324, 358
DeleteLists, 358, 476
DeleteProgram, 86, 359
DeleteQueries, 129, 358
DeleteRenderbuffers, 326, 339, 359, 465
DeleteShader, 84, 359
DeleteSync, 361, 362, 382
DeleteTextures, 259, 339, 358, 465
DeleteVertexArrays, 59, 358
DEPTH, 227, 305, 306, 316, 389, 415, 423, 427, 479
DEPTH24_STENCIL8, 216, 221
DEPTH32F_STENCIL8, 216, 221
DEPTH_ATTACHMENT, 323, 329, 337, 479, 480
DEPTH_BIAS, 172, 199
DEPTH_BITS, 318, 341, 452, 475
DEPTH_BUFFER, 480

DEPTH_BUFFER_BIT, 304, 306, 318–320, 394
DEPTH_CLAMP, 137, 405
DEPTH_CLEAR_VALUE, 420
DEPTH_COMPONENT, 113, 174, 175, 177, 186, 196, 214, 221, 266, 269, 275, 310, 312, 313, 316, 318, 336, 372, 373, 375–377
DEPTH_COMPONENT16, 216, 221
DEPTH_COMPONENT24, 216, 221
DEPTH_COMPONENT32, 221
DEPTH_COMPONENT32F, 216, 221
DEPTH_FUNC, 419
DEPTH_RANGE, 405
DEPTH_SCALE, 172, 198
DEPTH_STENCIL, 113, 174, 175, 177, 183, 186, 189, 194–196, 198, 213, 214, 221, 227, 254, 266, 269, 275, 306, 308, 310, 312, 313, 316, 318, 329, 333, 336, 372, 373
DEPTH_STENCIL_ATTACHMENT, 329, 333, 389
DEPTH_TEST, 288, 419
DEPTH_TEXTURE_MODE, 241, 256, 266, 414, 474
DEPTH_WRITEMASK, 420
DepthFunc, 288, 308
DepthMask, 303, 308
DepthRange, 126, 141, 367, 459
DetachShader, 84, 359
dFdx, 365
dFdy, 365
DIFFUSE, 77, 78, 407, 408
Disable, 39, 66, 67, 70, 72, 76, 77, 137, 146, 149–151, 156, 159, 163, 166, 168, 205–207, 212, 243, 268, 271, 284, 286, 288, 290, 295, 346, 347, 473–475
DisableClientState, 30, 37, 46, 358, 472
Disablei, 290
DisableVertexAttribArray, 37, 358, 388
DITHER, 295, 419
DOMAIN, 370, 432
DONT_CARE, 364, 365, 442
DOT3_RGB, 264
DOT3_RGBA, 264
DOUBLE, 35, 38, 39
DOUBLEBUFFER, 452
DRAW_BUFFER, 298, 302, 311
DRAW_BUFFERi, 290, 302, 305, 337, 422
DRAW_BUFFER0, 302
DRAW_FRAMEBUFFER, 308, 322–324, 328, 330, 339, 389, 421
DRAW_FRAMEBUFFER_BINDING, 29, 196, 209, 250, 298, 300, 318, 320, 324, 340, 341, 421
DRAW_PIXEL_TOKEN, 355
DrawArrays, 40–43, 57, 59, 113, 356
DrawArraysInstanced, 42
DrawBuffer, 297–299, 301, 303, 306
DrawBuffers, 297, 298, 300, 301, 492
DrawElements, 41–43, 58, 59, 131, 356
DrawElementsBaseVertex, 43, 44, 58
DrawElementsInstanced, 43, 58
DrawElementsInstancedBaseVertex, 43, 58
DrawPixels, 105, 129, 133, 146, 165, 169–172, 174, 177, 188, 196–198, 201, 207, 208, 277, 308, 316, 342, 353, 474
DrawRangeElements, 42, 44, 58, 356, 445
DrawRangeElementsBaseVertex, 43, 44, 58
DST_ALPHA, 293
DST_COLOR, 293
DYNAMIC_COPY, 49, 51

DYNAMIC_DRAW, 49, 50
DYNAMIC_READ, 49, 50

EDGE_FLAG, 398
EDGE_FLAG_ARRAY, 37, 46, 401
EDGE_FLAG_ARRAY_BUFFER_BINDING, 402
EDGE_FLAG_ARRAY_POINTER, 378, 401
EDGE_FLAG_ARRAY_STRIDE, 401
EdgeFlag, 29, 30
EdgeFlag*, 472
EdgeFlagPointer, 30, 34–36, 358, 472
EdgeFlagv, 29, 38
ELEMENT_ARRAY_BUFFER, 48, 58
ELEMENT_ARRAY_BUFFER_BINDING, 402
EMISSION, 77, 78, 407
Enable, 39, 66, 67, 70, 72, 76, 77, 137, 146, 149–151, 156, 159, 163, 166, 168, 205–207, 212, 243, 268, 271, 284, 286, 288, 290, 295, 346, 347, 367, 473–475
ENABLE_BIT, 394
EnableClientState, 30, 37, 46, 358, 472
Enablei, 290
EnableVertexAttribArray, 37, 59, 358, 388
End, 18, 22, 24, 25, 27, 29, 30, 39–41, 60, 77, 80, 105, 129, 135, 156, 162, 166, 339, 348, 349, 354, 472
EndConditionalRender, 129, 130
EndList, 356, 476
EndPrimitive, 343
EndQuery, 128, 289
EndTransformFeedback, 130, 131, 468, 469
EQUAL, 241, 267, 286–288
EQUIV, 296

EVAL_BIT, 394
EvalCoord, 30, 346, 347
EvalCoord*, 475
EvalCoord1, 347–349
EvalCoord1d, 348
EvalCoord1f, 348
EvalCoord2, 347, 349, 350
EvalMesh*, 475
EvalMesh1, 129, 348
EvalMesh2, 129, 348, 349
EvalPoint, 30
EvalPoint*, 475
EvalPoint1, 349
EvalPoint2, 349
EXP, 271, 272, 406
EXP2, 271
EXTENSIONS, 172, 378–380, 446, 476, 498
EYE_LINEAR, 68–70, 369, 416
EYE_PLANE, 68, 69, 416

FALSE, 29, 35, 49, 51, 56, 73, 75, 83, 85, 86, 100, 116, 117, 134, 135, 151, 170, 172, 181, 182, 196, 197, 199, 207, 241, 256, 259, 276, 285, 308, 310, 313, 332, 367, 374, 377, 378, 380–382, 384, 385, 388, 389, 391, 399–401, 403–412, 414–416, 418, 419, 423, 426–434, 438, 439, 453
FASTEST, 364, 365
FEEDBACK, 351–353, 460
FEEDBACK_BUFFER_POINTER, 378, 453
FEEDBACK_BUFFER_SIZE, 453
FEEDBACK_BUFFER_TYPE, 453
FeedbackBuffer, 352, 353, 358, 378, 475
FenceSync, 359–361, 364, 467, 492

FILL, 166–169, 348, 410, 459
Finish, 359, 360, 458, 467
FIRST_VERTEX_CONVENTION, 134
FIXED_ONLY, 197, 313, 320, 321, 406
FLAT, 134, 459
flat, 121, 134
FLOAT, 35, 38, 39, 45–47, 88, 96, 183, 185, 214, 312, 313, 315, 357, 369, 371, 390, 399–401
float, 86, 96, 102
FLOAT_32_UNSIGNED_INT_24_8_REV, 183, 185, 187–189, 193, 311, 314, 315
FLOAT_MAT2, 88, 96
FLOAT_MAT2x3, 88, 96
FLOAT_MAT2x4, 88, 96
FLOAT_MAT3, 88, 96
FLOAT_MAT3x2, 88, 96
FLOAT_MAT3x4, 88, 96
FLOAT_MAT4, 88, 96
FLOAT_MAT4x2, 88, 96
FLOAT_MAT4x3, 88, 96
FLOAT_VEC2, 88, 96
FLOAT_VEC3, 88, 96
FLOAT_VEC4, 88, 96
Flush, 359, 360, 364, 458
FlushMappedBufferRange, 53, 55, 358, 467, 480
FOG, 271, 406, 475
Fog, 271, 272, 475
FOG_BIT, 394
FOG_COLOR, 272, 406
FOG_COORD, 140, 271, 272
FOG_COORD_ARRAY, 37, 46, 399
FOG_COORD_ARRAY_BUFFER_BINDING, 402
FOG_COORD_ARRAY_POINTER, 378, 399
FOG_COORD_ARRAY_STRIDE, 399
FOG_COORD_ARRAY_TYPE, 399
FOG_COORD_SRC, 143, 271, 272, 406
FOG_DENSITY, 272, 406
FOG_END, 272, 406
FOG_HINT, 365, 442, 476
FOG_INDEX, 272, 406
FOG_MODE, 271, 272, 406
FOG_START, 272, 406
FogCoord, 30, 31
FogCoord*, 472
FogCoord[type]v, 38
FogCoordPointer, 30, 35, 36, 358, 472
FRAGMENT_DEPTH, 271, 272, 406
FRAGMENT_SHADER, 273, 385
FRAGMENT_SHADER_DERIVATIVE_HINT, 365, 442
FRAMEBUFFER, 322, 328, 330, 339, 389
FRAMEBUFFER_ATTACHMENT_x_SIZE, 423
FRAMEBUFFER_ATTACHMENT_ALPHA_SIZE, 390
FRAMEBUFFER_ATTACHMENT_BLUE_SIZE, 390
FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING, 291, 294, 390, 423
FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE, 390, 423
FRAMEBUFFER_ATTACHMENT_DEPTH_SIZE, 390
FRAMEBUFFER_ATTACHMENT_GREEN_SIZE, 390
FRAMEBUFFER_ATTACHMENT_LAYERED, 332, 391, 423
FRAMEBUFFER_ATTACHMENT_OBJECT_NAME, 329, 332, 336, 337, 390, 391, 423
FRAMEBUFFER_ATTACH-

MENT_OBJECT_TYPE, 328, 332, 336, 337, 341, 390, 391, 423
FRAMEBUFFER_ATTACHMENT_RED_SIZE, 390
FRAMEBUFFER_ATTACHMENT_STENCIL_SIZE, 390
FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE, 332, 391, 423, 486
FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER, 332, 336, 337, 342, 391, 423
FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL, 250, 332, 334, 391, 423
FRAMEBUFFER_BINDING, 324
FRAMEBUFFER_COMPLETE, 340
FRAMEBUFFER_DEFAULT, 390
FRAMEBUFFER_INCOMPLETE_ATTACHMENT, 337
FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER, 337
FRAMEBUFFER_INCOMPLETE_LAYER_TARGETS, 338
FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT, 337
FRAMEBUFFER_INCOMPLETE_MULTISAMPLE, 338
FRAMEBUFFER_INCOMPLETE_READ_BUFFER, 338
FRAMEBUFFER_SRGB, 291, 294, 419
FRAMEBUFFER_UNDEFINED, 337
FRAMEBUFFER_UNSUPPORTED, 338, 339
FramebufferRenderbuffer, 328, 329, 338, 339, 359
FramebufferTexture, 330, 332, 359
FramebufferTexture*, 332, 333, 338, 339
FramebufferTexture1D, 330, 331, 359
FramebufferTexture2D, 330–332, 359
FramebufferTexture3D, 330–332, 359
FramebufferTextureLayer, 331, 332, 359, 486
FRONT, 76, 77, 80, 163, 164, 166, 287, 290, 298, 299, 301–303, 305, 311, 321, 369, 474
FRONT_AND_BACK, 76, 77, 79, 80, 163, 166, 287, 290, 299, 301–303, 305, 311
FRONT_FACE, 410
FRONT_LEFT, 299, 300, 389
FRONT_RIGHT, 299, 300, 389
FrontFace, 76, 163, 276, 473
Frustum, 63, 64, 459, 472
ftransform, 114, 115
FUNC_ADD, 291, 292, 294, 419
FUNC_REVERSE_SUBTRACT, 291, 292
FUNC_SUBTRACT, 291, 292
fwidth, 365

Gen*, 466, 472
GenBuffers, 47, 358
GENERATE_MIPMAP, 241, 242, 253, 256, 414, 475
GENERATE_MIPMAP_HINT, 365, 442, 476
GenerateMipmap, 252, 358
GenFramebuffers, 322, 324, 358, 491
GenLists, 357, 358, 476
GenQueries, 128, 358
GenRenderbuffers, 325, 326, 358, 491
GenTextures, 259, 358, 374
GenVertexArrays, 59, 358, 491

GEOMETRY_INPUT_TYPE, 119, 386, 435
GEOMETRY_OUTPUT_TYPE, 120, 386, 435
GEOMETRY_SHADER, 118, 385
GEOMETRY_VERTICES_OUT, 120, 124, 386, 435
GEQUAL, 241, 267, 286–288
Get, 31, 127, 140, 359, 366, 367
GetActiveUniformBlockiv, 437
GetActiveAttrib, 87, 88, 109, 435
GetActiveUniform, 95, 96, 98, 101, 434
GetActiveUniformBlockiv, 92
GetActiveUniformBlockName, 92
GetActiveUniformName, 94
GetActiveUniformsiv, 95, 98, 436, 437
GetAttachedShaders, 386, 434
GetAttribLocation, 88, 89, 435
GetBooleani_v, 302, 366, 420
GetBooleanv, 134, 285, 302, 366–368, 395, 398, 407, 411, 420, 426, 427, 451, 452
GetBufferParameteri64v, 382, 404
GetBufferParameteriv, 382, 404, 480
GetBufferPointerv, 383, 404
GetBufferSubData, 383, 404
GetClipPlane, 369, 405
GetColorTable, 177, 310, 375, 428
GetColorTable- Parameterfv, 428
GetColorTable- Parameteriv, 428
GetColorTableParameter, 375
GetCompressedTexImage, 233–235, 365, 371, 373, 374
GetConvolution- Filter, 429
GetConvolution- Parameterfv, 429
GetConvolution- Parameteriv, 429, 445
GetConvolutionFilter, 310, 376
GetConvolutionParameter, 376
GetConvolutionParameteriv, 178, 179
GetDoublev, 366–368, 395
GetError, 17, 18, 453
GetFloatv, 14, 285, 366–368, 375, 395, 398, 405–407, 409–411, 419, 420, 427, 430–432, 443–445
GetFragDataLocation, 279
GetFramebufferAttachment- Parameteriv, 423
GetFramebufferAttachmentiv, 481
GetFramebufferAttachmentParameteriv, 341, 389, 390, 479, 480
GetHistogram, 182, 310, 376, 430
GetHistogram- Parameteriv, 430
GetHistogramParameter, 377
GetInteger64i_v, 367, 383, 384, 436, 440
GetInteger64v, 363, 366, 367, 451
GetIntegeri_v, 285, 366, 383, 411, 436, 440
GetIntegerv, 42, 93, 101, 102, 105, 148, 300, 302, 324, 326, 366–368, 375, 379, 395, 398–403, 405–407, 409, 410, 412, 413, 416, 418–422, 424, 426, 427, 431, 434, 436, 440, 442–453
GetLight, 369
GetLightfv, 408
GetMap, 370
GetMapfv, 432
GetMapiv, 432
GetMaterial, 369
GetMaterialfv, 407, 408
GetMinmax, 310, 377, 431
GetMinmax- Parameteriv, 431
GetMinmaxParameter, 378
GetMultisamplefv, 112, 148, 452
GetPixelMap, 369, 431
GetPixelMapuiv, 369
GetPixelMapusv, 369
GetPointerv, 378, 399–401, 453
GetPolygonStipple, 310, 374, 410

GetProgramInfoLog, 85, 386, 434
GetProgramiv, 85, 87, 92, 94, 109, 115, 119, 120, 124, 385–387, 434–436
GetQueryiv, 380, 451, 453
GetQueryObject[u]iv, 381
GetQueryObjectiv, 381, 439
GetQueryObjectuiv, 381, 439
GetRenderbufferParameteriv, 425
GetRenderbufferParameteriv, 341, 392, 481
GetSeparable- Filter, 429
GetSeparableFilter, 310, 376
GetShaderInfoLog, 84, 386, 433
GetShaderiv, 83, 84, 384, 387, 433
GetShaderSource, 387, 433
GetString, 379, 380, 446, 476, 498
GetStringi, 446, 498
GetSynciv, 361, 381, 382, 441
GetTexEnv, 369
GetTexEnvfv, 416, 417
GetTexEnviv, 416, 417
GetTexGen, 369
GetTexGenfv, 416
GetTexGeniv, 416
GetTexImage, 257, 310, 372, 373, 375–378, 413, 486
GetTexLevelParameter, 370, 371, 415
GetTexParameter, 341, 370, 414
GetTexParameterfv, 257, 259, 414
GetTexParameterI, 370
GetTexParameterIiv, 370
GetTexParameterIuiv, 370
GetTexParameteriv, 257, 259, 414
GetTransformFeedbackVarying, 435
GetTransformFeedbackVarying, 108, 109
GetUniform, 434
GetUniform*, 389
GetUniformBlockIndex, 91
GetUniformfv, 388
GetUniformIndices, 93–95
GetUniformiv, 389
GetUniformLocation, 91, 94, 95, 106, 434
GetUniformuiv, 389
GetVertexAttribPointerv, 401
GetVertexAttribdv, 387, 388
GetVertexAttribfv, 387, 388, 438
GetVertexAttribIiv, 387, 388
GetVertexAttribIuiv, 387, 388
GetVertexAttribiv, 387, 388, 401, 402
GetVertexAttribPointerv, 388
GL_APPLE_flush_buffer_range, 478, 505
GL_APPLE_vertex_array_object, 478, 506
GL_ARB_color_buffer_float, 478, 503
GL_ARB_compatibility, 471, 484, 485, 489, 507
GL_ARB_copy_buffer, 484, 507
GL_ARB_depth_buffer_float, 504
GL_ARB_depth_clamp, 507
GL_ARB_depth_texture, 500
GL_ARB_depth_clamp, 490
GL_ARB_draw_buffers, 502
GL_ARB_draw_buffers_blend, 509
GL_ARB_draw_elements_base_vertex, 490, 507
GL_ARB_draw_instanced, 484, 504, 505
GL_ARB_fragment_coord_conventions, 490, 508
GL_ARB_fragment_program, 501, 502
GL_ARB_fragment_program_shadow, 502
GL_ARB_fragment_shader, 502
GL_ARB_framebuffer_object, 504
GL_ARB_framebuffer_sRGB, 504
GL_ARB_geometry_shader4, 505

GL_ARB_geometry_shader4, 490
GL_ARB_half_float_pixel, 478, 503
GL_ARB_half_float_vertex, 505
GL_ARB_imaging, 172
GL_ARB_instanced_arrays, 505
GL_ARB_map_buffer_range, 505
GL_ARB_matrix_palette, 500
GL_ARB_multisample, 499
GL_ARB_multitexture, 498
GL_ARB_occlusion_query, 501
GL_ARB_pixel_buffer_object, 504
GL_ARB_point_parameters, 499
GL_ARB_point_sprite, 502
GL_ARB_provoking_vertex, 508
GL_ARB_provoking_vertex, 490
GL_ARB_sample_shading, 509
GL_ARB_seamless_cube_map, 508
GL_ARB_seamless_cube_map, 490
GL_ARB_shader_objects, 501
GL_ARB_shader_texture_lod, 507
GL_ARB_shading_language_100, 502
GL_ARB_shadow, 500, 502
GL_ARB_shadow_ambient, 501
GL_ARB_sync, 490, 508
GL_ARB_texture_border_clamp, 499
GL_ARB_texture_buffer_object, 485, 505
GL_ARB_texture_compression, 499
GL_ARB_texture_compression_rgtc, 506
GL_ARB_texture_cube_map, 499
GL_ARB_texture_cube_map_array, 509
GL_ARB_texture_env_add, 499
GL_ARB_texture_env_combine, 500
GL_ARB_texture_env_crossbar, 500
GL_ARB_texture_env_dot3, 500
GL_ARB_texture_float, 478, 504
GL_ARB_texture_gather, 509
GL_ARB_texture_mirrored_repeat, 500
GL_ARB_texture_multisample, 508
GL_ARB_texture_non_power_of_two, 502
GL_ARB_texture_query_lod, 509
GL_ARB_texture_rectangle, 485, 503
GL_ARB_texture_rg, 506
GL_ARB_texture_multisample, 490
GL_ARB_transpose_matrix, 498
GL_ARB_uniform_buffer_object, 485, 506
GL_ARB_vertex_array_bgra, 490, 508
GL_ARB_vertex_array_object, 506
GL_ARB_vertex_blend, 499
GL_ARB_vertex_buffer_object, 501
GL_ARB_vertex_program, 501
GL_ARB_vertex_shader, 502
GL_ARB_window_pos, 501
GL_ARB_*name*, 497
gl_BackColor, 75, 114, 122, 124, 134, 473
gl_BackSecondaryColor, 75, 114, 122, 124, 134, 473
gl_ClipDistance, 114, 479
gl_ClipDistance[], 122, 137
gl_ClipVertex, 114, 122, 124, 137, 479
gl_Color, 276
GL_EXT_draw_buffers2, 478
GL_EXT_framebuffer_blit, 478, 504
GL_EXT_framebuffer_multisample, 478, 504
GL_EXT_framebuffer_object, 478, 504
GL_EXT_framebuffer_sRGB, 478, 505
GL_EXT_geometry_shader4, 481
GL_EXT_gpu_shader4, 477
GL_EXT_packed_depth_stencil, 478
GL_EXT_packed_float, 478
GL_EXT_texture_array, 478, 481
GL_EXT_texture_compression_rgtc, 478, 506
GL_EXT_texture_integer, 478
GL_EXT_texture_shared_exponent, 478

GL_EXT_transform_feedback, 478
gl_FogFragCoord, 114, 122, 124, 140
gl_FragColor, 277, 278, 301
gl_FragCoord, 275
gl_FragCoord.z, 457
gl_FragData, 278, 301
gl_FragData[n], 277, 278
gl_FragDepth, 277, 278, 457
gl_FrontColor, 75, 114, 122, 124, 134
gl_FrontFacing, 276
gl_FrontSecondaryColor, 75, 114, 122, 124, 134
gl_in[], 122
gl_InstanceID, 42, 113, 114
gl_Layer, 124, 125, 342, 343
GL_NV_conditional_render, 477
GL_NV_depth_buffer_float, 478, 504
GL_NV_half_float, 478, 505
GL_NV_primitive_restart, 484
gl_PointCoord, 154
gl_PointSize, 114, 122, 124, 150
gl_Position, 107, 114, 123, 124, 126, 460
gl_PrimitiveID, 124, 277
gl_PrimitiveIDIn, 123
gl_SecondaryColor, 276
gl_TexCoord, 114
gl_TexCoord[], 122, 124
gl_VertexID, 113, 277
GLX_ARB_create_context, 506
GLX_ARB_create_context_profile, 471, 510
GLX_ARB_fbconfig_float, 503
GLX_ARB_framebuffer_sRGB, 505
GREATER, 241, 267, 286–288
GREEN, 172, 186, 312, 316, 415, 423, 427, 428, 430, 452
GREEN_BIAS, 199
GREEN_BITS, 341, 475
GREEN_INTEGER, 186
GREEN_SCALE, 198

HALF_FLOAT, 35, 38, 39, 185, 214, 312, 313, 315
Hint, 364, 476
HINT_BIT, 394
HISTOGRAM, 181, 182, 206, 377, 430
Histogram, 181, 182, 207, 359
HISTOGRAM_x_SIZE, 430
HISTOGRAM_ALPHA_SIZE, 377
HISTOGRAM_BLUE_SIZE, 377
HISTOGRAM_FORMAT, 377, 430
HISTOGRAM_GREEN_SIZE, 377
HISTOGRAM_LUMINANCE_SIZE, 377
HISTOGRAM_RED_SIZE, 377
HISTOGRAM_SINK, 377, 430
HISTOGRAM_WIDTH, 377, 430

INCR, 287
INCR_WRAP, 287
INDEX, 390, 452
Index, 29, 32
Index*, 472
Index[type]v, 38
INDEX_ARRAY, 37, 46, 400
INDEX_ARRAY_BUFFER_BINDING, 402
INDEX_ARRAY_POINTER, 378, 400
INDEX_ARRAY_STRIDE, 400
INDEX_ARRAY_TYPE, 400
INDEX_CLEAR_VALUE, 420
INDEX_LOGIC_OP, 295
INDEX_LOGIC_OP (v1.0:LOGIC_OP), 419
INDEX_MODE, 452
INDEX_OFFSET, 172, 199, 427
INDEX_SHIFT, 172, 199, 427
INDEX_WRITEMASK, 420
IndexMask, 302

IndexPointer, 30, 34–36, 358, 472
INFO_LOG_LENGTH, 385, 387, 433, 434
InitNames, 350, 475
INT, 35, 88, 96, 185, 314, 315, 357, 371, 390
int, 96, 102
INT_SAMPLER_1D, 97
INT_SAMPLER_1D_ARRAY, 97
INT_SAMPLER_2D, 97
INT_SAMPLER_2D_ARRAY, 97
INT_SAMPLER_2D_MULTISAMPLE, 97
INT_SAMPLER_2D_MULTISAMPLE_ARRAY, 97
INT_SAMPLER_2D_RECT, 97
INT_SAMPLER_3D, 97
INT_SAMPLER_BUFFER, 97
INT_SAMPLER_CUBE, 97
INT_VEC2, 88, 96
INT_VEC3, 88, 96
INT_VEC4, 88, 96
INTENSITY, 182, 201, 202, 214, 220, 221, 241, 242, 262, 263, 266, 374, 415, 428, 474
INTENSITY12, 220
INTENSITY16, 220
INTENSITY4, 220
INTENSITY8, 220
INTERLEAVED_ATTRIBS, 108, 132, 386, 435
InterleavedArrays, 30, 44, 45, 358, 472
INTERPOLATE, 264
INVALID_ENUM, 18, 19, 38, 55, 69, 76, 171, 177, 182, 183, 212, 227, 233–236, 242, 257, 299–301, 306, 311, 361, 373, 375–377, 382, 391, 392, 481, 492
INVALID_FRAMEBUFFER_OPERATION, 19, 29, 176, 180, 196, 209, 232, 311, 318, 320, 340
INVALID_INDEX, 91, 94
INVALID_OPERATION, 19, 30, 35, 48, 51, 53, 55–57, 59, 66, 83–86, 88, 91, 100, 101, 106, 109, 115, 116, 118, 125, 128, 130–133, 171, 173, 174, 183, 187, 188, 196, 212, 214, 223, 227, 231, 233–238, 253, 258, 279, 299, 301, 306, 308, 310–312, 316, 318–322, 326–328, 330–332, 339, 346, 351, 353, 356, 368, 369, 371–374, 381, 384, 386, 389, 390, 392, 471–473, 479, 480, 492, 493
INVALID_VALUE, 18, 19, 33, 35, 37, 39, 40, 42, 49, 51, 53, 55, 57, 65, 76, 82, 87, 88, 92, 94, 95, 105, 108, 109, 127, 130, 131, 148, 150, 151, 156, 170, 172, 173, 175, 176, 178, 179, 181, 213, 222–225, 227, 229–231, 233, 235, 237, 251, 259, 272, 278, 284, 285, 290, 299, 300, 302, 304, 306, 327, 330–332, 345, 346, 348, 356, 361–363, 367, 371, 373, 374, 380, 382, 384, 388, 472–474, 486, 491
INVERT, 287, 296
Is, 359
isampler1D, 97
isampler1DArray, 97
isampler2D, 97
isampler2DArray, 97
isampler2DMS, 97
isampler2DMSArray, 97
isampler2DRect, 97
isampler3D, 97
isamplerBuffer, 97
isamplerCube, 97

IsBuffer, 382
IsEnabled, 284, 294, 367, 395, 399–401, 403, 405–412, 416, 418, 419, 428–432, 438, 453
IsEnabledi, 294, 367, 419
IsFramebuffer, 389
IsList, 358, 476
IsProgram, 385
IsQuery, 380
IsRenderbuffer, 391
IsShader, 384
IsSync, 382
IsTexture, 374
IsVertexArray, 384
ivec2, 96
ivec3, 96
ivec4, 96

KEEP, 287, 288, 418

LAST_VERTEX_CONVENTION, 134, 136, 406
Layered images, 325
layout, 103
LEFT, 290, 298, 299, 301, 302, 305, 311
LEQUAL, 241, 256, 267, 286–288, 414
LESS, 241, 267, 286–289, 419
Light, 76–78
LIGHTi, 76, 77, 408, 460, 473
Light*, 473
LIGHT0, 76
LIGHT_MODEL_AMBIENT, 78, 407
LIGHT_MODEL_COLOR_CONTROL, 78, 407
LIGHT_MODEL_LOCAL_VIEWER, 78, 407
LIGHT_MODEL_TWO_SIDE, 78, 407
LIGHTING, 72, 407, 473
LIGHTING_BIT, 394
LightModel, 76, 78
LightModel*, 473
LINE, 166–169, 348, 349, 410
LINE_BIT, 394
LINE_LOOP, 22, 119, 125, 131
LINE_RESET_TOKEN, 355
LINE_SMOOTH, 156, 162, 409
LINE_SMOOTH_HINT, 365, 442
LINE_STIPPLE, 159, 409, 473
LINE_STIPPLE_PATTERN, 409
LINE_STIPPLE_REPEAT, 409
LINE_STRIP, 22, 119, 120, 125, 131, 348, 386
LINE_STRIP_ADJACENCY, 25, 119, 125
LINE_TOKEN, 355
LINE_WIDTH, 409
LINEAR, 111, 241–243, 248, 250, 252–256, 271, 319, 320, 334, 390, 414
LINEAR_ATTENUATION, 78, 408
LINEAR_MIPMAP_LINEAR, 241, 250, 252, 334
LINEAR_MIPMAP_NEAREST, 241, 250, 252, 334
LINES, 22, 119, 125, 131, 159, 386
lines, 119
LINES_ADJACENCY, 25, 119, 125, 386
lines_adjacency, 119
LineStipple, 159, 473
LineWidth, 156, 472, 473, 484
LINK_STATUS, 85, 385, 434
LinkProgram, 85–87, 89, 91–95, 105, 106, 108, 109, 124, 133, 279, 359
LIST_BASE, 453
LIST_BIT, 394
LIST_INDEX, 453
LIST_MODE, 453
ListBase, 357, 359, 476

LOAD, 307
LoadIdentity, 63, 472
LoadMatrix, 62, 63, 472
LoadMatrix[fd], 63
LoadName, 350, 351, 475
LoadTransposeMatrix, 62, 472
LoadTransposeMatrix[fd], 63
LOGIC_OP, 295
LOGIC_OP_MODE, 419
LogicOp, 296
LOWER_LEFT, 151, 154, 155
LUMINANCE, 186, 195, 201, 202, 213, 214, 220, 221, 241, 242, 256, 262, 263, 266, 312, 313, 316, 374, 414, 415, 428, 430, 462, 474
LUMINANCE12, 220
LUMINANCE12_ALPHA12, 220
LUMINANCE12_ALPHA4, 220
LUMINANCE16, 220
LUMINANCE16_ALPHA16, 220
LUMINANCE4, 220
LUMINANCE4_ALPHA4, 220
LUMINANCE6_ALPHA2, 220
LUMINANCE8, 220, 256
LUMINANCE8_ALPHA8, 220
LUMINANCE_ALPHA, 186, 195, 201, 202, 213, 214, 220, 221, 262, 263, 312, 313, 316, 374, 474

MAJOR_VERSION, 379, 446
Map*, 475
Map1, 344–346, 368
MAP1_x, 432
MAP1_COLOR_4, 345
MAP1_GRID_DOMAIN, 432
MAP1_GRID_SEGMENTS, 432
MAP1_INDEX, 345
MAP1_NORMAL, 345
MAP1_TEXTURE_COORD_1, 345, 347
MAP1_TEXTURE_COORD_2, 345, 347
MAP1_TEXTURE_COORD_3, 345
MAP1_TEXTURE_COORD_4, 345
MAP1_VERTEX_3, 345
MAP1_VERTEX_4, 345
Map2, 345, 346, 368
MAP2_x, 432
MAP2_GRID_DOMAIN, 432
MAP2_GRID_SEGMENTS, 432
MAP2_VERTEX_3, 347
MAP2_VERTEX_4, 347
MAP_COLOR, 172, 199, 200, 427
MAP_FLUSH_EXPLICIT_BIT, 53–55
MAP_INVALIDATE_BUFFER_BIT, 53, 54
MAP_INVALIDATE_RANGE_BIT, 53, 54
MAP_READ_BIT, 52–54
MAP_STENCIL, 172, 200, 427
MAP_UNSYNCHRONIZED_BIT, 53, 54
MAP_WRITE_BIT, 52–54
Map{12}, 346
MapBuffer, 49, 51, 54, 55, 90, 133, 358, 480
MapBufferRange, 51–55, 358, 480
MapGrid*, 475
MapGrid1, 348
MapGrid2, 348
mat$C$, 102
mat$C$xR, 102
mat2, 86, 96
mat2x3, 87, 96
mat2x4, 87, 96
mat3, 87, 96
mat3x2, 86, 96
mat3x4, 87, 96

mat4, 87, 96
mat4x2, 86, 96
mat4x3, 87, 96
Material, 30, 76–78, 81, 459
Material*, 473
MATRIX_MODE, 66, 405
MatrixMode, 62, 472
MAX, 291, 292
MAX_3D_TEXTURE_SIZE, 223, 331, 443
MAX_ARRAY_TEXTURE_LAYERS, 223, 443
MAX_ATTRIB_STACK_DEPTH, 392, 444, 476
MAX_CLIENT_ATTRIB_STACK_DEPTH, 392, 444, 476
MAX_CLIP_DISTANCES, 443, 479, 485
MAX_CLIP_PLANES, 479
MAX_COLOR_ATTACHMENTS, 299, 301, 322, 329, 340, 452
MAX_COLOR_MATRIX_STACK_DEPTH, 375, 443
MAX_COLOR_TEXTURE_SAMPLES, 237, 451
MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS, 273, 450
MAX_COMBINED_GEOMETRY_UNIFORM_COMPONENTS, 450
MAX_COMBINED_TEXTURE_IMAGE_UNITS, 112, 212, 368, 450
MAX_COMBINED_UNIFORM_BLOCKS, 102, 450, 493
MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS, 90, 450
MAX_CONVOLUTION_HEIGHT, 178, 376, 445
MAX_CONVOLUTION_WIDTH, 178, 179, 376, 445
MAX_CUBE_MAP_TEXTURE_SIZE, 223, 331, 443
MAX_DEPTH_TEXTURE_SAMPLES, 237, 451
MAX_DRAW_BUFFERS, 278, 279, 290, 294, 300, 302, 306, 452
MAX_ELEMENTS_INDICES, 42, 445
MAX_ELEMENTS_VERTICES, 42, 445
MAX_EVAL_ORDER, 345, 346, 444
MAX_FRAGMENT_INPUT_COMPONENTS, 277, 449
MAX_FRAGMENT_UNIFORM_BLOCKS, 101, 449
MAX_FRAGMENT_UNIFORM_COMPONENTS, 273, 449
MAX_GEOMETRY_INPUT_COMPONENTS, 123, 448
MAX_GEOMETRY_OUTPUT_COMPONENTS, 125, 448
MAX_GEOMETRY_OUTPUT_VERTICES, 124, 448
MAX_GEOMETRY_TEXTURE_IMAGE_UNITS, 112, 448
MAX_GEOMETRY_TOTAL_OUTPUT_COMPONENTS, 124, 448
MAX_GEOMETRY_UNIFORM_BLOCKS, 101, 448
MAX_GEOMETRY_UNIFORM_COMPONENTS, 120
MAX_INTEGER_SAMPLES, 237, 327, 328, 451
MAX_LIGHTS, 443

MAX_LIST_NESTING, 444
MAX_MODELVIEW_STACK_DEPTH, 443
MAX_NAME_STACK_DEPTH, 444
MAX_PIXEL_MAP_TABLE, 173, 199, 444
MAX_PROGRAM_TEXEL_OFFSET, 245, 449
MAX_PROJECTION_STACK_DEPTH, 443
MAX_RECTANGLE_TEXTURE_SIZE, 223, 445
MAX_RENDERBUFFER_SIZE, 327, 443
MAX_SAMPLE_MASK_WORDS, 285, 451
MAX_SAMPLES, 237, 327, 328, 452
MAX_SERVER_WAIT_TIMEOUT, 363, 451
MAX_TEXTURE_BUFFER_SIZE, 238, 445
MAX_TEXTURE_COORDS, 31, 34, 46, 66, 212, 368, 449, 475
MAX_TEXTURE_IMAGE_UNITS, 112, 275, 449
MAX_TEXTURE_LOD_BIAS, 244, 443
MAX_TEXTURE_SIZE, 223, 237, 331, 443
MAX_TEXTURE_STACK_DEPTH, 443
MAX_TEXTURE_UNITS, 19, 212, 269, 393, 449, 475
MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS, 108, 440
MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS, 108, 131, 132, 384, 440
MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS, 108, 440
MAX_UNIFORM_BLOCK_SIZE, 93, 450
MAX_UNIFORM_BUFFER_BINDINGS, 105, 383, 450, 493
MAX_VARYING_COMPONENTS, 107, 450, 472, 479, 490
MAX_VARYING_FLOATS, 472, 479, 490
MAX_VERTEX_ATTRIBS, 32–35, 37, 46, 87–89, 388, 447
MAX_VERTEX_OUTPUT_COMPONENTS, 107, 447
MAX_VERTEX_TEXTURE_IMAGE_UNITS, 112, 447
MAX_VERTEX_UNIFORM_BLOCKS, 101, 447
MAX_VERTEX_UNIFORM_COMPONENTS, 90, 447
MAX_VIEWPORT_DIMS, 380, 444
MIN, 291, 292
MIN_PROGRAM_TEXEL_OFFSET, 245, 449
MINMAX, 182, 207, 377, 378, 431
Minmax, 182, 207
MINMAX_FORMAT, 378, 431
MINMAX_SINK, 378, 431
MINOR_VERSION, 379, 446
MIRRORED_REPEAT, 241, 242, 248
MODELVIEW, 62, 66
MODELVIEW_MATRIX, 368
MODELVIEW_MATRIX (TRANSPOSE_MODELVIEW_MATRIX), 405
MODELVIEW_STACK_DEPTH, 405
MODULATE, 261–264, 416
MULT, 307
MultiDrawArrays, 40
MultiDrawElements, 41, 58

MultiDrawElementsBaseVertex, 44, 58
MULTISAMPLE, 149, 155, 162, 168, 207, 210, 284, 297, 411
MULTISAMPLE_BIT, 394
MultiTexCoord, 29, 31, 38
MultiTexCoord[size][type]v, 38
MultMatrix, 62, 63, 472
MultMatrix[fd], 63
MultTransposeMatrix, 62, 472
MultTransposeMatrix[fd], 63

N3F_V3F, 44, 45
NAME_STACK_DEPTH, 453
NAND, 296
NEAREST, 111, 112, 241–243, 247, 248, 250, 252, 253, 255, 266, 319, 334
NEAREST_MIPMAP_LINEAR, 241, 250, 252, 254, 256, 334
NEAREST_MIPMAP_NEAREST, 241, 250, 252, 254, 255, 266, 334
NEVER, 241, 267, 286–288
NewList, 356, 357, 476
NICEST, 364, 365
NO_ERROR, 17, 18
NONE, 113, 241, 256, 257, 266, 275, 297, 299–302, 306, 311, 312, 321, 336, 337, 371, 390, 414, 415, 423, 480, 486
NOOP, 296
noperspective, 139
NOR, 296
Normal, 29, 31, 86
Normal3, 31
Normal3*, 472
Normal3[type]v, 38
NORMAL_ARRAY, 37, 46, 399
NORMAL_ARRAY_BUFFER_BINDING, 402
NORMAL_ARRAY_POINTER, 378, 399
NORMAL_ARRAY_STRIDE, 399
NORMAL_ARRAY_TYPE, 399
NORMAL_MAP, 68, 69
NORMALIZE, 67, 405, 473
NormalPointer, 30, 34–36, 46, 358, 472
NOTEQUAL, 241, 267, 286–288
NULL, 35, 47, 49, 51, 53, 58, 83, 87, 92, 94, 95, 109, 382, 383, 386, 387, 396, 401, 404
NUM_COMPRESSED_TEXTURE_FORMATS, 215, 445
NUM_EXTENSIONS, 380, 446
NV, 485

OBJECT_LINEAR, 68, 70, 369
OBJECT_PLANE, 68, 69, 416
OBJECT_TYPE, 361, 382, 441
ONE, 293, 294, 419
ONE_MINUS_CONSTANT_ALPHA, 293
ONE_MINUS_CONSTANT_COLOR, 293
ONE_MINUS_DST_ALPHA, 293
ONE_MINUS_DST_COLOR, 293
ONE_MINUS_SRC_ALPHA, 265, 293
ONE_MINUS_SRC_COLOR, 265, 293
OPERAND0_ALPHA, 417
OPERAND0_RGB, 417
OPERAND1_ALPHA, 417
OPERAND1_RGB, 417
OPERAND2_ALPHA, 417
OPERAND2_RGB, 417
OPERAND*n*_ALPHA, 261, 263, 265, 269
OPERAND*n*_RGB, 261, 263, 265, 269
OR, 296
OR_INVERTED, 296
OR_REVERSE, 296

ORDER, 370, 432
Ortho, 63, 65, 459, 472
OUT_OF_MEMORY, 18, 19, 51, 54, 237, 327, 356

PACK_ALIGNMENT, 310, 426
PACK_IMAGE_HEIGHT, 310, 373, 426
PACK_LSB_FIRST, 310, 426
PACK_ROW_LENGTH, 310, 426
PACK_SKIP_IMAGES, 310, 373, 426
PACK_SKIP_PIXELS, 310, 426
PACK_SKIP_ROWS, 310, 426
PACK_SWAP_BYTES, 310, 426
PASS_THROUGH_TOKEN, 355
PassThrough, 354, 475
PERSPECTIVE_CORRECTION_HINT, 365, 442, 476
PIXEL_MAP_A_TO_A, 173, 199
PIXEL_MAP_B_TO_B, 173, 199
PIXEL_MAP_G_TO_G, 173, 199
PIXEL_MAP_I_TO_A, 173, 199
PIXEL_MAP_I_TO_B, 173, 199
PIXEL_MAP_I_TO_G, 173, 199
PIXEL_MAP_I_TO_I, 173, 200
PIXEL_MAP_I_TO_R, 173, 199
PIXEL_MAP_R_TO_R, 173, 199
PIXEL_MAP_S_TO_S, 173, 200
PIXEL_MODE_BIT, 394
PIXEL_PACK_BUFFER, 48, 171, 308
PIXEL_PACK_BUFFER_BINDING, 314, 369, 373, 426
PIXEL_UNPACK_BUFFER, 48, 171
PIXEL_UNPACK_BUFFER_BINDING, 173, 183, 233, 426
PixelMap, 170, 172–174, 321
PixelStore, 30, 169, 170, 172, 310, 321, 358
PixelTransfer, 169, 172, 205, 321
PixelZoom, 197, 208, 474
POINT, 166–169, 348, 349, 410
POINT_BIT, 394
POINT_DISTANCE_ATTENUATION, 151, 409
POINT_FADE_THRESHOLD_SIZE, 151, 409
POINT_SIZE, 409
POINT_SIZE_GRANULARITY, 444
POINT_SIZE_MAX, 151, 409
POINT_SIZE_MIN, 151, 409
POINT_SIZE_RANGE, 444
POINT_SMOOTH, 151, 155, 409, 473
POINT_SMOOTH_HINT, 365, 442, 476
POINT_SPRITE, 151, 155, 156, 260, 261, 369, 409, 473
POINT_SPRITE_COORD_ORIGIN, 151, 154, 155, 409
POINT_TOKEN, 355
PointParameter, 150
PointParameter*, 151
POINTS, 22, 119, 120, 125, 131, 166, 348, 386
points, 119
PointSize, 150
POLYGON, 23, 27, 29, 131, 277, 473
POLYGON_BIT, 394
POLYGON_MODE, 410
POLYGON_OFFSET_FACTOR, 410
POLYGON_OFFSET_FILL, 168, 410
POLYGON_OFFSET_LINE, 168, 410
POLYGON_OFFSET_POINT, 168, 410
POLYGON_OFFSET_UNITS, 410
POLYGON_SMOOTH, 163, 168, 410
POLYGON_SMOOTH_HINT, 365, 442
POLYGON_STIPPLE, 166, 410, 474
POLYGON_STIPPLE_BIT, 394
POLYGON_TOKEN, 355
PolygonMode, 162, 166, 167, 169, 351, 353, 474

PolygonOffset, 167
PolygonStipple, 165, 170, 474
PopAttrib, 392, 393, 395, 460, 476
PopClientAttrib, 30, 358, 392, 393, 395, 476
PopMatrix, 66, 472
PopName, 350, 475
POSITION, 78, 369, 408
POST_COLOR_MATRIX_x_BIAS, 172, 430
POST_COLOR_MATRIX_x_SCALE, 172, 430
POST_COLOR_MATRIX_ALPHA_BIAS, 206
POST_COLOR_MATRIX_ALPHA_SCALE, 205
POST_COLOR_MATRIX_BLUE_BIAS, 205
POST_COLOR_MATRIX_BLUE_SCALE, 205
POST_COLOR_MATRIX_COLOR_TABLE, 174, 206, 428
POST_COLOR_MATRIX_GREEN_BIAS, 205
POST_COLOR_MATRIX_GREEN_SCALE, 205
POST_COLOR_MATRIX_RED_BIAS, 205
POST_COLOR_MATRIX_RED_SCALE, 205
POST_CONVOLUTION_x_BIAS, 172, 430
POST_CONVOLUTION_x_SCALE, 172, 430
POST_CONVOLUTION_ALPHA_BIAS, 205
POST_CONVOLUTION_ALPHA_SCALE, 205
POST_CONVOLUTION_BLUE_BIAS, 205
POST_CONVOLUTION_BLUE_SCALE, 205
POST_CONVOLUTION_COLOR_TABLE, 174, 205, 428
POST_CONVOLUTION_GREEN_BIAS, 205
POST_CONVOLUTION_GREEN_SCALE, 205
POST_CONVOLUTION_RED_BIAS, 205
POST_CONVOLUTION_RED_SCALE, 205
PREVIOUS, 263, 265, 417
PRIMARY_COLOR, 265
PRIMITIVE_RESTART, 39, 403, 484, 485
PRIMITIVE_RESTART_INDEX, 403, 485
PrimitiveRestartIndex, 39, 358, 492
PRIMITIVES_GENERATED, 127, 133, 380
PrioritizeTextures, 260, 475
PROGRAM_POINT_SIZE, 122, 150, 438, 491
PROJECTION, 62, 66
PROJECTION_MATRIX (TRANSPOSE_PROJECTION_MATRIX), 405
PROJECTION_STACK_DEPTH, 405
PROVOKING_VERTEX, 406
ProvokingVertex, 134
PROXY_COLOR_TABLE, 174, 177, 359
PROXY_HISTOGRAM, 181, 182, 359, 377
PROXY_POST_COLOR_MATRIX_COLOR_TABLE, 174, 359
PROXY_POST_CONVOLUTION_COLOR_TABLE, 174, 359
PROXY_TEXTURE_1D, 214, 225, 257,

359, 370
PROXY_TEXTURE_1D_ARRAY, 214, 224, 257, 359, 370
PROXY_TEXTURE_2D, 214, 224, 257, 359, 370
PROXY_TEXTURE_2D_ARRAY, 212, 214, 257, 359, 370
PROXY_TEXTURE_2D_MULTISAMPLE, 237, 257, 370
PROXY_TEXTURE_2D_MULTISAMPLE_ARRAY, 237, 257, 370
PROXY_TEXTURE_3D, 212, 257, 359, 370
PROXY_TEXTURE_CUBE_MAP, 214, 224, 257, 359, 370
PROXY_TEXTURE_RECTANGLE, 214, 224, 233, 235, 257, 370
PushAttrib, 392, 393, 395, 476
PushClientAttrib, 30, 358, 392, 393, 395, 476
PushMatrix, 66, 472
PushName, 350, 475

Q, 68, 69, 369
QUAD_STRIP, 24, 27, 131, 277, 473
QUADRATIC_ATTENUATION, 78, 408
QUADS, 24, 27, 29, 131, 277, 473
QUADS_FOLLOW_PROVOKING_VERTEX, 134, 135, 451
QUERY_BY_REGION_NO_WAIT, 130
QUERY_BY_REGION_WAIT, 130
QUERY_COUNTER_BITS, 380, 451
QUERY_NO_WAIT, 130
QUERY_RESULT, 381, 439
QUERY_RESULT_AVAILABLE, 381, 439
QUERY_WAIT, 129

R, 68, 69, 369, 478
R11F_G11F_B10F, 216, 219
R16, 216, 218, 240
R16_SNORM, 216, 218
R16F, 216, 218, 240
R16I, 216, 219, 240
R16UI, 216, 219, 240
R32F, 216, 219, 240
R32I, 216, 219, 240
R32UI, 216, 219, 240
R3_G3_B2, 218
R8, 216, 218, 240, 415
R8_SNORM, 216, 218
R8I, 216, 219, 240
R8UI, 216, 219, 240
RASTERIZER_DISCARD, 146, 340
RasterPos, 115, 125, 140, 340, 351, 459
RasterPos*, 105, 473
RasterPos2, 140
RasterPos3, 140
RasterPos4, 140
READ_BUFFER, 312, 337, 342, 422
READ_FRAMEBUFFER, 308, 322–324, 328, 330, 339, 389, 421
READ_FRAMEBUFFER_BINDING, 176, 180, 232, 310–312, 318, 320, 324, 421
READ_ONLY, 49, 54
READ_WRITE, 49, 51, 54, 404
ReadBuffer, 299, 311, 312, 321
ReadPixels, 133, 169, 170, 172, 188, 201, 227, 308–312, 314, 316, 318, 340, 343, 358, 373–375, 474
Rect, 60, 163
Rect*, 473
RED, 172, 175, 186, 214, 218, 219, 221, 241, 242, 262, 263, 266, 312, 316, 336, 374, 415, 423, 427, 428, 430, 452

RED_BIAS, 199
RED_BITS, 341, 475
RED_INTEGER, 186
RED_SCALE, 198
REDUCE, 202, 203, 205, 429
REFLECTION_MAP, 68, 69
RENDER, 351, 352, 453
RENDER_MODE, 453
RENDERBUFFER, 325–329, 341, 390, 392, 424
RENDERBUFFER_ALPHA_SIZE, 392, 425
RENDERBUFFER_BINDING, 326, 424
RENDERBUFFER_BLUE_SIZE, 392, 425
RENDERBUFFER_DEPTH_SIZE, 392, 425
RENDERBUFFER_GREEN_SIZE, 392, 425
RENDERBUFFER_HEIGHT, 327, 392, 425
RENDERBUFFER_INTERNAL_FORMAT, 327, 392, 425
RENDERBUFFER_RED_SIZE, 392, 425
RENDERBUFFER_SAMPLES, 327, 338, 340, 392, 425
RENDERBUFFER_STENCIL_SIZE, 392, 425
RENDERBUFFER_WIDTH, 327, 392, 425
RenderbufferStorage, 327, 328, 339, 359
RenderbufferStorageMultisample, 326–328, 359, 486
RENDERER, 379, 446
RenderMode, 351–354, 358, 475
REPEAT, 241, 242, 248, 256
REPLACE, 261, 262, 264, 287
REPLICATE_BORDER, 202, 204
RESCALE_NORMAL, 67, 405, 473
ResetHistogram, 377
ResetMinmax, 378
RETURN, 307
RG, 175, 186, 214, 218, 219, 221, 262, 263, 312, 316, 336, 374, 478
RG16, 216, 218, 240
RG16_SNORM, 216, 218
RG16F, 216, 218, 240
RG16I, 216, 219, 240
RG16UI, 216, 219, 240
RG32F, 216, 219, 240
RG32I, 216, 219, 240
RG32UI, 216, 219, 240
RG8, 216, 218, 240
RG8_SNORM, 216, 218
RG8I, 216, 219, 240
RG8UI, 216, 219, 240
RG_INTEGER, 186
RGB, 186, 189, 194, 201, 202, 213, 214, 217–219, 221, 261–263, 293, 312, 314, 316, 336, 374
RGB10, 218
RGB10_A2, 216, 218
RGB12, 218
RGB16, 216, 218
RGB16_SNORM, 216, 218
RGB16F, 216, 218
RGB16I, 216, 219
RGB16UI, 216, 219
RGB32F, 216, 219
RGB32I, 216, 219
RGB32UI, 216, 219
RGB4, 218
RGB5, 218
RGB5_A1, 218
RGB8, 216, 218
RGB8_SNORM, 216, 218
RGB8I, 216, 219

RGB8UI, 216, 219
RGB9_E5, 175, 216, 219, 268, 314
RGB_INTEGER, 186
RGB_SCALE, 261, 417
RGBA, 176, 177, 180, 182, 186, 189, 194, 201, 202, 213, 214, 218, 219, 221, 262, 263, 312, 316, 336, 374, 425, 428–431, 474
RGBA12, 218
RGBA16, 216, 218, 240
RGBA16_SNORM, 216, 218
RGBA16F, 216, 219, 240
RGBA16I, 216, 219, 240
RGBA16UI, 216, 219, 240
RGBA2, 218
RGBA32F, 216, 219, 240
RGBA32I, 216, 219, 240
RGBA32UI, 216, 219, 240
RGBA4, 218
RGBA8, 216, 218, 240
RGBA8_SNORM, 216, 218
RGBA8I, 216, 219, 240
RGBA8UI, 216, 219, 240
RGBA_INTEGER, 186
RGBA_MODE, 452
RIGHT, 290, 298, 299, 301, 302, 305, 311
Rotate, 63, 459, 472

S, 68, 69, 369
SAMPLE_ALPHA_TO_COVERAGE, 284, 411
SAMPLE_ALPHA_TO_ONE, 284, 285, 411
SAMPLE_BUFFERS, 148, 155, 162, 168, 207, 210, 284, 289, 297, 303, 310, 318, 320, 340, 452
SAMPLE_COVERAGE, 284, 285, 411
SAMPLE_COVERAGE_INVERT, 284, 285, 411
SAMPLE_COVERAGE_VALUE, 284, 285, 411
SAMPLE_MASK, 284, 285, 411
SAMPLE_MASK_VALUE, 284, 285, 411
SAMPLE_POSITION, 148, 452
SampleCoverage, 285
SampleMaski, 285
sampler1D, 96
sampler1DArray, 97
sampler1DArrayShadow, 97
sampler1DShadow, 97, 113, 275
sampler2D, 96, 106
sampler2DArray, 97
sampler2DArrayShadow, 97
sampler2DMS, 97
sampler2DMSArray, 97
sampler2DRect, 97
sampler2DRectShadow, 97, 113, 275
sampler2DShadow, 97, 113, 275
sampler3D, 97
SAMPLER_1D, 96
SAMPLER_1D_ARRAY, 97
SAMPLER_1D_ARRAY_SHADOW, 97
SAMPLER_1D_SHADOW, 97
SAMPLER_2D, 96
SAMPLER_2D_ARRAY, 97
SAMPLER_2D_ARRAY_SHADOW, 97
SAMPLER_2D_MULTISAMPLE, 97
SAMPLER_2D_MULTISAMPLE_ARRAY, 97
SAMPLER_2D_RECT, 97
SAMPLER_2D_RECT_SHADOW, 97
SAMPLER_2D_SHADOW, 97
SAMPLER_3D, 97
SAMPLER_BUFFER, 97
SAMPLER_CUBE, 97
SAMPLER_CUBE_SHADOW, 97

samplerBuffer, 97
samplerCube, 97
samplerCubeShadow, 97
SAMPLES, 148, 149, 289, 320, 340, 452
SAMPLES_PASSED, 129, 130, 289, 380
Scale, 63, 64, 459, 472
Scissor, 284
SCISSOR_BIT, 394
SCISSOR_BOX, 418
SCISSOR_TEST, 284, 418
SECONDARY_COLOR_ARRAY, 37, 46, 400
SECONDARY_COLOR_ARRAY_-BUFFER_BINDING, 402
SECONDARY_COLOR_ARRAY_-POINTER, 378, 400
SECONDARY_COLOR_ARRAY_-SIZE, 400
SECONDARY_COLOR_ARRAY_-STRIDE, 400
SECONDARY_COLOR_ARRAY_-TYPE, 400
SecondaryColor, 29, 32
SecondaryColor3, 32
SecondaryColor3*, 472
SecondaryColor3[type]v, 38
SecondaryColorPointer, 30, 34, 36, 358, 472
SELECT, 351, 352, 460
SelectBuffer, 351, 352, 358, 378, 475
SELECTION_BUFFER_POINTER, 378, 453
SELECTION_BUFFER_SIZE, 453
SEPARABLE_2D, 179, 201, 224, 376, 429
SeparableFilter2D, 170, 179
SEPARATE_ATTRIBS, 108, 132, 386
SEPARATE_SPECULAR_COLOR, 74
SET, 296
SHADE_MODEL, 406
ShadeModel, 134, 473
SHADER_SOURCE_LENGTH, 385, 387, 433
SHADER_TYPE, 116, 385, 433
ShaderSource, 83, 359, 387
SHADING_LANGUAGE_VERSION, 379, 446
SHININESS, 78, 407
SHORT, 35, 185, 314, 315, 357
SIGNALED, 361, 382
SIGNED_NORMALIZED, 371, 390
SINGLE_COLOR, 73, 74, 407
SLUMINANCE, 220, 267
SLUMINANCE8, 267
SLUMINANCE8_ALPHA8, 220, 267
SLUMINANCE_ALPHA, 267
SMOOTH, 134, 136, 406
SMOOTH_LINE_WIDTH_GRANULARITY (v1.1: LINE_WIDTH_GRANULARITY), 445
SMOOTH_LINE_WIDTH_RANGE (v1.1: LINE_WIDTH_-RANGE), 445
SPECULAR, 77, 78, 407, 408
SPHERE_MAP, 68–70
SPOT_CUTOFF, 78, 408
SPOT_DIRECTION, 78, 369, 408
SPOT_EXPONENT, 78, 408
SRC0_ALPHA, 417
SRC0_RGB, 417
SRC1_ALPHA, 417
SRC1_RGB, 417
SRC2_ALPHA, 417
SRC2_RGB, 417
SRC_ALPHA, 263, 265, 293, 417
SRC_ALPHA_SATURATE, 293
SRC_COLOR, 263, 265, 293, 417

SRC*n*_ALPHA, 261, 263, 265, 269
SRC*n*_RGB, 261, 263, 265, 269
SRGB, 267, 291, 294, 390
SRGB8, 216, 218, 267
SRGB8_ALPHA8, 216, 218, 267
SRGB_ALPHA, 267
STACK_OVERFLOW, 19, 66, 351, 392
STACK_UNDERFLOW, 19, 66, 351, 393
STATIC_COPY, 49, 50
STATIC_DRAW, 49, 50, 404
STATIC_READ, 49, 50
std140, 93, 103
STENCIL, 305, 306, 316, 389, 415, 423, 479
STENCIL_ATTACHMENT, 323, 329, 337, 480
STENCIL_ATTACMENT, 479, 480
STENCIL_BACK_FAIL, 418
STENCIL_BACK_FUNC, 418
STENCIL_BACK_PASS_DEPTH_FAIL, 418
STENCIL_BACK_PASS_DEPTH_PASS, 418
STENCIL_BACK_REF, 418
STENCIL_BACK_VALUE_MASK, 418
STENCIL_BACK_WRITEMASK, 420
STENCIL_BITS, 318, 341, 452, 475
STENCIL_BUFFER, 480
STENCIL_BUFFER_BIT, 304, 306, 318–320, 394
STENCIL_CLEAR_VALUE, 420
STENCIL_FAIL, 418
STENCIL_FUNC, 418
STENCIL_INDEX, 174, 177, 183, 186, 196, 198, 212, 308, 311, 312, 316, 318, 327, 336, 373
STENCIL_INDEX1, 327
STENCIL_INDEX16, 327
STENCIL_INDEX4, 327
STENCIL_INDEX8, 327
STENCIL_PASS_DEPTH_FAIL, 418
STENCIL_PASS_DEPTH_PASS, 418
STENCIL_REF, 418
STENCIL_TEST, 286, 418
STENCIL_VALUE_MASK, 418
STENCIL_WRITEMASK, 420
StencilFunc, 286–288, 458
StencilFuncSeparate, 287, 288
StencilMask, 303, 308, 458
StencilMaskSeparate, 303, 308
StencilOp, 287, 288
StencilOpSeparate, 287, 288
STEREO, 452
STREAM_COPY, 49, 50
STREAM_DRAW, 49, 50
STREAM_READ, 49, 50
SUBPIXEL_BITS, 443
SUBTRACT, 264
SYNC_CONDITION, 361, 382, 441
SYNC_FENCE, 361, 382, 441
SYNC_FLAGS, 361, 382, 441
SYNC_FLUSH_COMMANDS_BIT, 362, 364
SYNC_GPU_COMMANDS_COMPLETE, 361, 382, 441
SYNC_STATUS, 361, 382, 441

T, 68, 369
T2F_C3F_V3F, 44, 45
T2F_C4F_N3F_V3F, 44, 45
T2F_C4UB_V3F, 44, 45
T2F_N3F_V3F, 44, 45
T2F_V3F, 44, 45
T4F_C4F_N3F_V4F, 44, 45
T4F_V4F, 44, 45
TABLE_TOO_LARGE, 19, 175, 181
TexBuffer, 238, 358
TexCoord, 29–31
TexCoord*, 472

TexCoord1, 31
TexCoord2, 31
TexCoord3, 31
TexCoord4, 31
TexCoordPointer, 30, 35–37, 46, 358, 472
TexEnv, 66, 212, 260, 269, 475
TexEnv*, 151
TexGen, 66, 68, 69, 368
TexGen*, 473
TexImage, 212, 229
TexImage*, 467, 474, 485
TexImage*D, 169, 170
TexImage1D, 170, 201, 203, 220, 225, 226, 228, 229, 232, 234, 251, 257, 359
TexImage2D, 170, 201, 203, 220, 223–227, 229, 232, 234, 251, 257, 359
TexImage2DMultisample, 236, 237, 257
TexImage3D, 170, 212, 213, 220, 222, 224, 226, 229, 232, 234, 251, 257, 359, 373
TexImage3DMultisample, 237, 257
TexParameter, 212, 239, 467, 475
TexParameter*, 475
TexParameter[if], 245, 251
TexParameterf, 260
TexParameterfv, 260
TexParameterI, 239
TexParameteri, 239, 260
TexParameterIiv, 239
TexParameterIuiv, 239
TexParameteriv, 239, 260
TexSubImage, 229
TexSubImage*, 232, 236, 467
TexSubImage*D, 170
TexSubImage1D, 170, 201, 228, 229, 231, 234
TexSubImage2D, 170, 201, 228–231, 234
TexSubImage3D, 170, 228, 229, 231, 234
TEXTURE, 62, 65, 66, 263, 265, 332, 336, 337, 341, 390, 391, 417
TEXTUREi, 31, 212
TEXTURE0, 31, 38, 47, 212, 346, 353, 393, 403, 416
TEXTURE1, 393
TEXTURE_x_SIZE, 415
TEXTURE_x_TYPE, 415
TEXTURE_xD, 412, 413
TEXTURE_1D, 214, 225, 228, 239, 252, 258, 268, 331, 370, 372, 475
TEXTURE_1D_ARRAY, 214, 224, 227, 229, 239, 252, 258, 370, 372, 412, 413, 475
TEXTURE_2D, 106, 212, 214, 224, 227, 229, 239, 252, 258, 268, 331, 370, 372, 475
TEXTURE_2D_ARRAY, 212, 214, 222, 229, 234, 236, 239, 252, 258, 259, 370, 372, 412, 413, 475
TEXTURE_2D_MULTISAMPLE, 237, 258, 259, 331, 370, 413
TEXTURE_2D_MULTISAMPLE_ARRAY, 237, 258, 259, 331, 370, 413
TEXTURE_3D, 212, 222, 229, 239, 252, 257, 258, 268, 331, 370, 372, 475
TEXTURE_ALPHA_SIZE, 371
TEXTURE_ALPHA_TYPE, 371
TEXTURE_BASE_LEVEL, 223, 239, 241, 242, 250, 251, 256, 334, 335, 414
TEXTURE_BINDING_xD, 412
TEXTURE_BINDING_1D_ARRAY,

412
TEXTURE_BINDING_2D_ARRAY, 412
TEXTURE_BINDING_2D_MULTISAMPLE, 413
TEXTURE_BINDING_2D_MULTISAMPLE_ARRAY, 413
TEXTURE_BINDING_BUFFER, 413
TEXTURE_BINDING_CUBE_MAP, 413
TEXTURE_BINDING_RECTANGLE, 413
TEXTURE_BIT, 393, 394
TEXTURE_BLUE_SIZE, 371
TEXTURE_BLUE_TYPE, 371
TEXTURE_BORDER, 234, 235, 371, 415
TEXTURE_BORDER_COLOR, 239, 241, 248, 256, 370, 414, 481
TEXTURE_BUFFER, 48, 238, 239, 258, 259, 371, 413
TEXTURE_BUFFER_DATA_STORE_BINDING, 415
TEXTURE_COMPARE_FAIL_VALUE_ARB, 500
TEXTURE_COMPARE_FUNC, 241, 256, 266, 414
TEXTURE_COMPARE_MODE, 113, 241, 256, 266, 275, 414
TEXTURE_COMPONENTS, 372, 474
TEXTURE_COMPRESSED, 415
TEXTURE_COMPRESSED_IMAGE_SIZE, 234, 235, 371, 373, 415
TEXTURE_COMPRESSION_HINT, 365, 442
TEXTURE_COORD_ARRAY, 37, 46, 400
TEXTURE_COORD_ARRAY_BUFFER_BINDING, 402
TEXTURE_COORD_ARRAY_POINTER, 378, 400
TEXTURE_COORD_ARRAY_SIZE, 400
TEXTURE_COORD_ARRAY_STRIDE, 400
TEXTURE_COORD_ARRAY_TYPE, 400
TEXTURE_CUBE_MAP, 214, 224, 239, 252, 258, 259, 268, 370, 371, 412, 413, 475
TEXTURE_CUBE_MAP_*, 224
TEXTURE_CUBE_MAP_NEGATIVE_X, 224, 227, 229, 243, 331, 343, 370, 372, 413
TEXTURE_CUBE_MAP_NEGATIVE_Y, 224, 227, 229, 243, 331, 343, 370, 372, 413
TEXTURE_CUBE_MAP_NEGATIVE_Z, 224, 227, 229, 243, 331, 343, 370, 372, 413
TEXTURE_CUBE_MAP_POSITIVE_X, 224, 227, 229, 243, 331, 332, 343, 370, 372, 413
TEXTURE_CUBE_MAP_POSITIVE_Y, 224, 227, 229, 243, 331, 343, 370, 372, 413
TEXTURE_CUBE_MAP_POSITIVE_Z, 224, 227, 229, 243, 331, 343, 370, 372, 413
TEXTURE_CUBE_MAP_SEAMLESS, 243, 453
TEXTURE_DEPTH, 233, 235, 236, 371, 415
TEXTURE_DEPTH_SIZE, 371
TEXTURE_DEPTH_TYPE, 371
TEXTURE_ENV, 260, 261, 369, 475
TEXTURE_ENV_COLOR, 261, 416

TEXTURE_ENV_MODE, 261, 269, 416
TEXTURE_FILTER_CONTROL, 260, 261, 369, 475
TEXTURE_FIXED_SAMPLE_LOCATIONS, 338, 415
TEXTURE_GEN_x, 416
TEXTURE_GEN_*, 473
TEXTURE_GEN_MODE, 68–70, 416
TEXTURE_GEN_Q, 70
TEXTURE_GEN_R, 70
TEXTURE_GEN_S, 70
TEXTURE_GEN_T, 70
TEXTURE_GREEN_SIZE, 371
TEXTURE_GREEN_TYPE, 371
TEXTURE_HEIGHT, 231, 233, 235, 236, 371, 415
TEXTURE_INTENSITY_SIZE, 371
TEXTURE_INTENSITY_TYPE, 371
TEXTURE_INTERNAL_FORMAT, 234, 235, 372, 474
TEXTURE_INTERNAL_FORMAT (TEXTURE_COMPONENTS), 415
TEXTURE_LOD_BIAS, 241, 244, 261, 414, 416, 475
TEXTURE_LUMINANCE_SIZE, 371
TEXTURE_LUMINANCE_TYPE, 371
TEXTURE_MAG_FILTER, 241, 253, 256, 266, 414
TEXTURE_MATRIX (TRANSPOSE_TEXTURE_MATRIX), 405
TEXTURE_MAX_LEVEL, 239, 241, 251, 256, 335, 414
TEXTURE_MAX_LOD, 239, 241, 245, 256, 414
TEXTURE_MIN_FILTER, 241, 242, 247, 248, 250, 253, 254, 256, 266, 334, 414
TEXTURE_MIN_LOD, 239, 241, 245, 256, 414
TEXTURE_PRIORITY, 239, 241, 256, 260, 414, 475
TEXTURE_RECTANGLE, 214, 224, 225, 227, 229, 233–235, 239, 242, 258, 259, 331, 370, 372, 373, 413
TEXTURE_RECTANGLE_ARB, 503
TEXTURE_RED_SIZE, 371
TEXTURE_RED_TYPE, 371
TEXTURE_RESIDENT, 257, 259, 370, 414
TEXTURE_SAMPLES, 338, 415
TEXTURE_SHARED_SIZE, 371, 415
TEXTURE_STACK_DEPTH, 405
TEXTURE_STENCIL_SIZE, 371
TEXTURE_WIDTH, 231, 233, 235, 236, 371, 415
TEXTURE_WRAP_R, 241, 242, 247, 248, 414, 474
TEXTURE_WRAP_S, 241, 242, 247, 248, 414, 474
TEXTURE_WRAP_T, 241, 242, 247, 248, 414, 474
textureGather, 509
textureLOD, 509
TEXTURE*n*, 265, 269
TIMEOUT_EXPIRED, 362
TIMEOUT_IGNORED, 363
TRANSFORM_BIT, 394
TRANSFORM_FEEDBACK_BUFFER, 48, 49, 131, 133
TRANSFORM_FEEDBACK_BUFFER_BINDING, 383, 440
TRANSFORM_FEEDBACK_BUFFER_MODE, 386, 435
TRANSFORM_FEEDBACK_BUFFER_SIZE, 384, 440

TRANSFORM_FEEDBACK_BUFFER_START, 384, 440
TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, 127, 132, 133, 380
TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH, 109, 386, 435
TRANSFORM_FEEDBACK_VARYINGS, 108, 109, 386, 435
TransformFeedbackVaryings, 107–109, 132, 358, 491
Translate, 63, 64, 459, 472
TRANSPOSE_COLOR_MATRIX, 368, 375
TRANSPOSE_MODELVIEW_MATRIX, 368
TRANSPOSE_PROJECTION_MATRIX, 368
TRANSPOSE_TEXTURE_MATRIX, 368
TRIANGLE_FAN, 24, 27, 119, 125, 131
TRIANGLE_STRIP, 23, 27, 119, 120, 125, 131, 386, 435
TRIANGLE_STRIP_ADJACENCY, 27, 119, 126
TRIANGLES, 24, 27, 29, 119, 125, 131, 132, 386, 435
triangles, 119
TRIANGLES_ADJACENCY, 25, 27, 119, 126, 386
triangles_adjacency, 119
TRUE, 29, 35, 49, 54, 55, 73, 75, 81, 83, 85, 100, 116, 134, 135, 138, 151, 154, 170, 172, 181, 182, 197, 237, 241, 242, 253, 259, 276, 285, 302, 310, 313, 332, 338, 358, 367, 374, 377, 378, 380–382, 384, 385, 388, 389, 391, 398, 406, 411, 415, 419, 420, 473

uint, 102
Uniform, 14, 99
Uniform*, 90, 100, 101, 106
Uniform*f{v}, 100
Uniform*i{v}, 100
Uniform*ui{v}, 100
Uniform1f, 15
Uniform1i, 14
Uniform1i{v}, 100, 106
Uniform1iv, 100
Uniform2{if ui}*, 100
Uniform2f, 15
Uniform2i, 15
Uniform3f, 15
Uniform3i, 15
Uniform4f, 13, 15
Uniform4f{v}, 100
Uniform4i, 15
Uniform4i{v}, 101
UNIFORM_ARRAY_STRIDE, 99, 103, 437
UNIFORM_BLOCK_ACTIVE_UNIFORM_INDICES, 93, 437
UNIFORM_BLOCK_ACTIVE_UNIFORMS, 93, 437
UNIFORM_BLOCK_BINDING, 92, 437
UNIFORM_BLOCK_DATA_SIZE, 92, 105, 437
UNIFORM_BLOCK_INDEX, 98, 436
UNIFORM_BLOCK_NAME_LENGTH, 93
UNIFORM_BLOCK_REFERENCED_BY_FRAGMENT_SHADER, 93, 437

UNIFORM_BLOCK_REFERENCED_BY_GEOMETRY_SHADER, 93, 437
UNIFORM_BLOCK_REFERENCED_BY_VERTEX_SHADER, 93, 437
UNIFORM_BUFFER, 48, 49, 105
UNIFORM_BUFFER_BINDING, 383, 436
UNIFORM_BUFFER_OFFSET_ALIGNMENT, 105, 450
UNIFORM_BUFFER_SIZE, 383, 436
UNIFORM_BUFFER_START, 383, 436
UNIFORM_IS_ROW_MAJOR, 99, 437
UNIFORM_MATRIX_STRIDE, 99, 102, 103, 437
UNIFORM_NAME_LENGTH, 98, 436
UNIFORM_OFFSET, 99, 436
UNIFORM_SIZE, 98, 436
UNIFORM_TYPE, 98, 436
Uniform{1,2,3,4}ui, 99
Uniform{1,2,3,4}uiv, 99
UniformBlockBinding, 105
UniformMatrix2x4fv, 100
UniformMatrix3fv, 100
UniformMatrix{234}fv, 99, 100
UniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}fv, 99, 100
UnmapBuffer, 53, 55, 56, 90, 358, 467
UNPACK_ALIGNMENT, 170, 187, 213, 426
UNPACK_IMAGE_HEIGHT, 170, 213, 426
UNPACK_LSB_FIRST, 170, 196, 426
UNPACK_ROW_LENGTH, 170, 187, 188, 213, 426
UNPACK_SKIP_IMAGES, 170, 213, 224, 426
UNPACK_SKIP_PIXELS, 170, 188, 196, 426
UNPACK_SKIP_ROWS, 170, 188, 196, 426
UNPACK_SWAP_BYTES, 170, 187, 426
UNSIGNALED, 361, 382, 441
unsigned int, 96
UNSIGNED_BYTE, 35, 41, 45, 185, 314, 315, 357, 462
UNSIGNED_BYTE_2_3_3_REV, 185, 189, 190, 315
UNSIGNED_BYTE_3_3_2, 185, 189, 190, 315
UNSIGNED_INT, 35, 41, 88, 96, 185, 314, 315, 357, 369, 371, 390
UNSIGNED_INT_10_10_10_2, 185, 189, 192, 315
UNSIGNED_INT_10F_11F_11F_REV, 185, 189, 192, 194, 214, 313–315
UNSIGNED_INT_24_8, 183, 185, 189, 192, 311, 314, 315
UNSIGNED_INT_2_10_10_10_REV, 185, 189, 192, 315
UNSIGNED_INT_5_9_9_9_REV, 185, 189, 192, 194, 214, 217, 313–315
UNSIGNED_INT_8_8_8_8, 185, 189, 192, 315
UNSIGNED_INT_8_8_8_8_REV, 185, 189, 192, 315
UNSIGNED_INT_SAMPLER_1D, 97
UNSIGNED_INT_SAMPLER_1D_ARRAY, 97
UNSIGNED_INT_SAMPLER_2D, 97
UNSIGNED_INT_SAMPLER_2D_ARRAY, 98
UNSIGNED_INT_SAMPLER_2D_MULTISAMPLE, 98
UNSIGNED_INT_SAMPLER_2D_-

MULTISAMPLE_ARRAY, 98
UNSIGNED_INT_SAMPLER_2D_RECT, 98
UNSIGNED_INT_SAMPLER_3D, 97
UNSIGNED_INT_SAMPLER_BUFFER, 98
UNSIGNED_INT_SAMPLER_CUBE, 97
UNSIGNED_INT_VEC2, 88, 96
UNSIGNED_INT_VEC3, 88, 96
UNSIGNED_INT_VEC4, 88, 96
UNSIGNED_NORMALIZED, 371, 390
UNSIGNED_SHORT, 35, 41, 185, 314, 315, 357, 369
UNSIGNED_SHORT_1_5_5_5_REV, 185, 189, 191, 315
UNSIGNED_SHORT_4_4_4_4, 185, 189, 191, 315
UNSIGNED_SHORT_4_4_4_4_REV, 185, 189, 191, 315
UNSIGNED_SHORT_5_5_5_1, 185, 189, 191, 315
UNSIGNED_SHORT_5_6_5, 185, 189, 191, 315
UNSIGNED_SHORT_5_6_5_REV, 185, 189, 191, 315
UPPER_LEFT, 151, 154, 155, 409
usampler1D, 97
usampler1DArray, 97
usampler2D, 97
usampler2DArray, 98
usampler2DMS, 98
usampler2DMSArray, 98
usampler2DRect, 98
usampler3D, 97
usamplerBuffer, 98
usamplerCube, 97
UseProgram, 85, 86, 109, 121, 133
uvec2, 96
uvec3, 96
uvec4, 96

V2F, 44, 45
V3F, 44, 45
VALIDATE_STATUS, 115, 385, 434
ValidateProgram, 115, 116, 359, 385
vec2, 86, 96
vec3, 86, 96
vec4, 86, 96, 100
VENDOR, 379, 446
VERSION, 379, 446
Vertex, 29, 30, 86, 140, 347
Vertex*, 472
Vertex2, 30, 33, 60
Vertex3, 30, 33
Vertex4, 30, 33
Vertex[size][type]v, 39
VERTEX_ARRAY, 37, 46, 399
VERTEX_ARRAY_BINDING, 368, 388, 403
VERTEX_ARRAY_BUFFER_BINDING, 402
VERTEX_ARRAY_POINTER, 378, 399
VERTEX_ARRAY_SIZE, 399
VERTEX_ARRAY_STRIDE, 399
VERTEX_ARRAY_TYPE, 399
VERTEX_ATTRIB_ARRAY_BUFFER_BINDING, 57, 388, 402
VERTEX_ATTRIB_ARRAY_ENABLED, 388, 401
VERTEX_ATTRIB_ARRAY_INTEGER, 388, 401
VERTEX_ATTRIB_ARRAY_NORMALIZED, 388, 401
VERTEX_ATTRIB_ARRAY_POINTER, 388, 401
VERTEX_ATTRIB_ARRAY_SIZE, 388, 401

VERTEX_ATTRIB_ARRAY_STRIDE, 388, 401
VERTEX_ATTRIB_ARRAY_TYPE, 388, 401
VERTEX_PROGRAM_POINT_SIZE, 491
VERTEX_PROGRAM_TWO_SIDE, 75, 76, 438, 473
VERTEX_SHADER, 83, 385
VertexAttrib, 29, 32, 129
VertexAttrib*, 33, 86, 472
VertexAttrib1*, 33
VertexAttrib2*, 33
VertexAttrib3*, 33
VertexAttrib4, 32
VertexAttrib4*, 33
VertexAttrib4N, 32
VertexAttrib4Nub, 32
VertexAttrib[size][type]v, 38, 39
VertexAttrib[size]N[type]v, 38, 39
VertexAttribI, 33
VertexAttribI4, 33
VertexAttribI[size][type]v, 38, 39
VertexAttribIPointer, 35–37, 358, 388
VertexAttribPointer, 30, 35, 36, 57, 59, 358, 388, 473
VertexPointer, 30, 34, 36, 46, 358, 472
VIEWPORT, 405
Viewport, 127
VIEWPORT_BIT, 394

WAIT_FAILED, 362
WaitSync, 361–364, 451, 466, 467
WGL_ARB_create_context, 506
WGL_ARB_create_context_profile, 471, 510
WGL_ARB_framebuffer_sRGB, 505
WGL_ARB_pixel_format_float, 503
WindowPos, 141, 351
WindowPos*, 473
WindowPos2, 141
WindowPos3, 141
WRITE_ONLY, 49, 54

XOR, 296

ZERO, 287, 293, 294, 419
ZOOM_X, 431
ZOOM_Y, 431