The OpenGL® Graphics System:
A Specification
(Version 3.1 - May 28, 2009)

Mark Segal
Kurt Akeley

Editor (version 1.1): Chris Frazier
Editor (versions 1.2-3.1): 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, Inc.

Contents

1 **Introduction** 1
- 1.1 What is the OpenGL Graphics System? 1
- 1.2 Programmer's View of OpenGL 1
- 1.3 Implementor's View of OpenGL 2
- 1.4 Our View 2
- 1.5 The Deprecation Model 3
- 1.6 Companion Documents 3
  - 1.6.1 OpenGL Shading Language 3
  - 1.6.2 Window System Bindings 3

2 **OpenGL Operation** 5
- 2.1 OpenGL Fundamentals 5
  - 2.1.1 Floating-Point Computation 7
  - 2.1.2 16-Bit Floating-Point Numbers 8
  - 2.1.3 Unsigned 11-Bit Floating-Point Numbers 8
  - 2.1.4 Unsigned 10-Bit Floating-Point Numbers 9
  - 2.1.5 Fixed-Point Data Conversions 10
- 2.2 GL State 12
  - 2.2.1 Shared Object State 13
- 2.3 GL Command Syntax 13
- 2.4 Basic GL Operation 15
- 2.5 GL Errors 18
- 2.6 Primitives and Vertices 19
  - 2.6.1 Primitive Types 21
- 2.7 Vertex Specification 23
- 2.8 Vertex Arrays 24
  - 2.8.1 Transferring Array Elements 26
  - 2.8.2 Drawing Commands 27
- 2.9 Buffer Objects 30

- 2.9.1 Mapping and Unmapping Buffer Data . . . 34
- 2.9.2 Effects of Accessing Outside Buffer Bounds . . . 38
- 2.9.3 Copying Between Buffers . . . 38
- 2.9.4 Vertex Arrays in Buffer Objects . . . 39
- 2.9.5 Array Indices in Buffer Objects . . . 40
- 2.9.6 Buffer Object State . . . 40
- 2.10 Vertex Array Objects . . . 40
- 2.11 Vertex Shaders . . . 41
  - 2.11.1 Shader Objects . . . 42
  - 2.11.2 Program Objects . . . 44
  - 2.11.3 Vertex Attributes . . . 46
  - 2.11.4 Uniform Variables . . . 49
  - 2.11.5 Samplers . . . 64
  - 2.11.6 Varying Variables . . . 65
  - 2.11.7 Shader Execution . . . 67
  - 2.11.8 Required State . . . 72
- 2.12 Coordinate Transformations . . . 73
  - 2.12.1 Controlling the Viewport . . . 74
- 2.13 Asynchronous Queries . . . 75
- 2.14 Conditional Rendering . . . 77
- 2.15 Transform Feedback . . . 78
- 2.16 Primitive Queries . . . 81
- 2.17 Primitive Clipping . . . 81
  - 2.17.1 Clipping Shader Varying Outputs . . . 83

3 Rasterization **84**

- 3.1 Discarding Primitives Before Rasterization . . . 85
- 3.2 Invariance . . . 85
- 3.3 Antialiasing . . . 86
  - 3.3.1 Multisampling . . . 87
- 3.4 Points . . . 88
  - 3.4.1 Basic Point Rasterization . . . 89
  - 3.4.2 Point Rasterization State . . . 90
  - 3.4.3 Point Multisample Rasterization . . . 90
- 3.5 Line Segments . . . 91
  - 3.5.1 Basic Line Segment Rasterization . . . 91
  - 3.5.2 Other Line Segment Features . . . 93
  - 3.5.3 Line Rasterization State . . . 94
  - 3.5.4 Line Multisample Rasterization . . . 95
- 3.6 Polygons . . . 95

- 3.6.1 Basic Polygon Rasterization . . . 95
- 3.6.2 Antialiasing . . . 98
- 3.6.3 Options Controlling Polygon Rasterization . . . 98
- 3.6.4 Depth Offset . . . 99
- 3.6.5 Polygon Multisample Rasterization . . . 100
- 3.6.6 Polygon Rasterization State . . . 100
- 3.7 Pixel Rectangles . . . 101
  - 3.7.1 Pixel Storage Modes and Pixel Buffer Objects . . . 101
  - 3.7.2 Transfer of Pixel Rectangles . . . 102
- 3.8 Texturing . . . 114
  - 3.8.1 Texture Image Specification . . . 115
  - 3.8.2 Alternate Texture Image Specification Commands . . . 127
  - 3.8.3 Compressed Texture Images . . . 134
  - 3.8.4 Buffer Textures . . . 138
  - 3.8.5 Texture Parameters . . . 141
  - 3.8.6 Depth Component Textures . . . 141
  - 3.8.7 Cube Map Texture Selection . . . 143
  - 3.8.8 Texture Minification . . . 144
  - 3.8.9 Texture Magnification . . . 152
  - 3.8.10 Combined Depth/Stencil Textures . . . 153
  - 3.8.11 Texture Completeness . . . 153
  - 3.8.12 Texture State and Proxy State . . . 154
  - 3.8.13 Texture Objects . . . 156
  - 3.8.14 Texture Comparison Modes . . . 157
  - 3.8.15 sRGB Texture Color Conversion . . . 159
  - 3.8.16 Shared Exponent Texture Color Conversion . . . 159
- 3.9 Fragment Shaders . . . 160
  - 3.9.1 Shader Variables . . . 160
  - 3.9.2 Shader Execution . . . 161
- 3.10 Antialiasing Application . . . 165
- 3.11 Multisample Point Fade . . . 165

4 Per-Fragment Operations and the Framebuffer 166
- 4.1 Per-Fragment Operations . . . 167
  - 4.1.1 Pixel Ownership Test . . . 168
  - 4.1.2 Scissor Test . . . 168
  - 4.1.3 Multisample Fragment Operations . . . 169
  - 4.1.4 Stencil Test . . . 170
  - 4.1.5 Depth Buffer Test . . . 172
  - 4.1.6 Occlusion Queries . . . 173

4.1.7 Blending 173
4.1.8 sRGB Conversion 178
4.1.9 Dithering 178
4.1.10 Logical Operation 179
4.1.11 Additional Multisample Fragment Operations 180
4.2 Whole Framebuffer Operations 181
4.2.1 Selecting a Buffer for Writing 181
4.2.2 Fine Control of Buffer Updates 185
4.2.3 Clearing the Buffers 187
4.3 Reading and Copying Pixels 189
4.3.1 Reading Pixels 189
4.3.2 Copying Pixels 196
4.3.3 Pixel Draw/Read State 198
4.4 Framebuffer Objects 199
4.4.1 Binding and Managing Framebuffer Objects 199
4.4.2 Attaching Images to Framebuffer Objects 202
4.4.3 Feedback Loops Between Textures and the Framebuffer . 209
4.4.4 Framebuffer Completeness 211
4.4.5 Effects of Framebuffer State on Framebuffer Dependent Values 216
4.4.6 Mapping between Pixel and Element in Attached Image . 216

5 Special Functions **218**
5.1 Flush and Finish 218
5.2 Hints 218

6 State and State Requests **220**
6.1 Querying GL State 220
6.1.1 Simple Queries 220
6.1.2 Data Conversions 221
6.1.3 Enumerated Queries 222
6.1.4 Texture Queries 224
6.1.5 String Queries 226
6.1.6 Asynchronous Queries 227
6.1.7 Buffer Object Queries 229
6.1.8 Vertex Array Object Queries 231
6.1.9 Shader and Program Queries 231
6.1.10 Framebuffer Object Queries 235
6.1.11 Renderbuffer Object Queries 238
6.2 State Tables 238

A Invariance 280
A.1 Repeatability 280
A.2 Multi-pass Algorithms 281
A.3 Invariance Rules 281
A.4 What All This Means 282

B Corollaries 284

C Compressed Texture Image Formats 286
C.1 RGTC Compressed Texture Image Formats 286
C.1.1 Format `COMPRESSED_RED_RGTC1` 287
C.1.2 Format `COMPRESSED_SIGNED_RED_RGTC1` 288
C.1.3 Format `COMPRESSED_RG_RGTC2` 288
C.1.4 Format `COMPRESSED_SIGNED_RG_RGTC2` 289

D Shared Objects and Multiple Contexts 290
D.1 Object Deletion Behavior 290
D.2 Propagating State Changes 291
D.2.1 Definitions 292
D.2.2 Rules 292

E The Deprecation Model 294
E.1 Profiles and Deprecated Features of OpenGL 3.0 294

F Version 3.0 and Before 300
F.1 New Features 300
F.2 Deprecation Model 301
F.3 Changed Tokens 302
F.4 Change Log 302
F.5 Credits and Acknowledgements 304

G Version 3.1 307
G.1 New Features 307
G.2 Deprecation Model 308
G.3 Change Log 308
G.4 Credits and Acknowledgements 309

H Extension Registry, Header Files, and ARB Extensions 312
H.1 Extension Registry 312
H.2 Header Files 312
H.3 ARB Extensions 313

H.3.1 Naming Conventions 314
H.3.2 Promoting Extensions to Core Features 314
H.3.3 Multitexture 314
H.3.4 Transpose Matrix 315
H.3.5 Multisample 315
H.3.6 Texture Add Environment Mode 315
H.3.7 Cube Map Textures 315
H.3.8 Compressed Textures 315
H.3.9 Texture Border Clamp 315
H.3.10 Point Parameters 315
H.3.11 Vertex Blend 315
H.3.12 Matrix Palette 316
H.3.13 Texture Combine Environment Mode 316
H.3.14 Texture Crossbar Environment Mode 316
H.3.15 Texture Dot3 Environment Mode 316
H.3.16 Texture Mirrored Repeat 316
H.3.17 Depth Texture 316
H.3.18 Shadow 316
H.3.19 Shadow Ambient 317
H.3.20 Window Raster Position 317
H.3.21 Low-Level Vertex Programming 317
H.3.22 Low-Level Fragment Programming 317
H.3.23 Buffer Objects 317
H.3.24 Occlusion Queries 317
H.3.25 Shader Objects 318
H.3.26 High-Level Vertex Programming 318
H.3.27 High-Level Fragment Programming 318
H.3.28 OpenGL Shading Language 318
H.3.29 Non-Power-Of-Two Textures 318
H.3.30 Point Sprites 318
H.3.31 Fragment Program Shadow 318
H.3.32 Multiple Render Targets 319
H.3.33 Rectangular Textures 319
H.3.34 Floating-Point Color Buffers 319
H.3.35 Half-Precision Floating Point 319
H.3.36 Floating-Point Textures 320
H.3.37 Pixel Buffer Objects 320
H.3.38 Floating-Point Depth Buffers 320
H.3.39 Instanced Rendering 320
H.3.40 Framebuffer Objects 320

H.3.41 sRGB Framebuffers 321
H.3.42 Geometry Shaders 321
H.3.43 Half-Precision Vertex Data 321
H.3.44 Instanced Rendering 321
H.3.45 Flexible Buffer Mapping 321
H.3.46 Texture Buffer Objects 322
H.3.47 RGTC Texture Compression Formats 322
H.3.48 One- and Two-Component Texture Formats 322
H.3.49 Vertex Array Objects 322
H.3.50 Versioned Context Creation 322
H.3.51 Uniform Buffer Objects 322
H.3.52 Restoration of features removed from OpenGL 3.0 323
H.3.53 Fast Buffer-to-Buffer Copies 323
H.3.54 Shader Texture Level of Detail Control 323

List of Figures

2.1 Block diagram of the GL. 15
2.2 Vertex processing and primitive assembly. 19
2.3 Triangle strips, fans, and independent triangles. 22

3.1 Rasterization. 84
3.2 Visualization of Bresenham's algorithm. 92
3.3 The region used in rasterizing an antialiased line segment. 94
3.4 Transfer of pixel rectangles. 102
3.5 Selecting a subimage from an image 107
3.6 A texture image and the coordinates used to access it. 127

4.1 Per-fragment operations. 168
4.2 Operation of **ReadPixels**. 189

List of Tables

2.1 GL command suffixes 14
2.2 GL data types 16
2.3 Summary of GL errors 19
2.4 Vertex array sizes (values per vertex) and data types 25
2.5 Buffer object binding targets. 31
2.6 Buffer object parameters and their values. 31
2.7 Buffer object initial state. 33
2.8 Buffer object state set by **MapBufferRange**. 36
2.9 OpenGL Shading Language type tokens 56
2.10 Transform feedback modes 78

3.1 **PixelStore** parameters. 102
3.2 Pixel data types. 105
3.3 Pixel data formats. 106
3.4 Swap Bytes bit ordering. 106
3.5 Packed pixel formats. 108
3.6 `UNSIGNED_BYTE` formats. Bit numbers are indicated for each component. 109
3.7 `UNSIGNED_SHORT` formats 110
3.8 `UNSIGNED_INT` formats 111
3.9 `FLOAT_UNSIGNED_INT` formats 112
3.10 Packed pixel field assignments. 113
3.11 Conversion from RGBA, depth, and stencil pixel components to internal texture components. 117
3.12 Sized internal color formats. 122
3.13 Sized internal depth and stencil formats. 123
3.14 Generic and specific compressed internal formats. 124
3.15 Internal formats for buffer textures 140
3.16 Texture parameters and their values. 142

3.17 Selection of cube map images. 143
3.18 Texel location wrap mode application. 147
3.19 Depth texture comparison functions. 158
3.20 Correspondence of filtered texture components to texture source components. 161

4.1 RGB and Alpha blend equations. 176
4.2 Blending functions. 177
4.3 Arguments to **LogicOp** and their corresponding operations. 180
4.4 Buffer selection for the default framebuffer 183
4.5 Buffer selection for a framebuffer object 183
4.6 **DrawBuffers** buffer selection for the default framebuffer 183
4.7 **PixelStore** parameters. 191
4.8 **ReadPixels** index masks. 194
4.9 **ReadPixels** GL data types and reversed component conversion formulas. 195
4.10 Correspondence of renderbuffer sized to base internal formats. . . 204
4.11 Framebuffer attachment points. 206

5.1 Hint targets and descriptions 219

6.1 Texture, table, and filter return values. 226
6.2 State Variable Types 239
6.3 Vertex Array Object State (cont.) 240
6.4 Vertex Array Object State (cont.) 241
6.5 Vertex Array Data (not in Vertex Array objects) 242
6.6 Buffer Object State 243
6.7 Transformation state 244
6.8 Coloring 245
6.9 Rasterization 246
6.10 Rasterization (cont.) 247
6.11 Multisampling 248
6.12 Textures (state per texture unit and binding point) 249
6.13 Textures (state per texture object) 250
6.14 Textures (state per texture image) 251
6.15 Texture Environment and Generation 252
6.16 Pixel Operations 253
6.17 Pixel Operations (cont.) 254
6.18 Framebuffer Control 255
6.19 Framebuffer (state per target binding point) 256

6.20 Framebuffer (state per framebuffer object) 257
6.21 Framebuffer (state per attachment point) 258
6.22 Renderbuffer (state per target and binding point) 259
6.23 Renderbuffer (state per renderbuffer object) 260
6.24 Pixels 261
6.25 Shader Object State 262
6.26 Program Object State 263
6.27 Program Object State (cont.) 264
6.28 Program Object State (cont.) 265
6.29 Program Object State (cont.) 266
6.30 Vertex Shader State 267
6.31 Query Object State 268
6.32 Transform Feedback State 269
6.33 Hints 270
6.34 Implementation Dependent Values 271
6.35 Implementation Dependent Values (cont.) 272
6.36 Implementation Dependent Values (cont.) 273
6.37 Implementation Dependent Values (cont.) 274
6.38 Implementation Dependent Values (cont.) 275
6.39 Implementation Dependent Values (cont.) 276
6.40 Implementation Dependent Values (cont.) (1) The minimum value for each stage is `MAX_`*`stage`*`_UNIFORM_BLOCKS` × `MAX_`*`stage`*`_UNIFORM_BLOCK_SIZE` + `MAX_`*`stage`*`_UNIFORM_COMPONENTS` 277
6.41 Framebuffer Dependent Values 278
6.42 Miscellaneous 279

F.1 New token names 302

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 What is the OpenGL Graphics System?

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

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

1.2 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.

1.3 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.

1.4 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.

1.5 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.

1.6 Companion Documents

1.6.1 OpenGL Shading Language

This specification should be read together with a companion document titled *The OpenGL Shading Language*. The latter document (referred to as the OpenGL Shading Language Specification hereafter) defines the syntax and semantics of the programming language used to write vertex and fragment shaders (see sections 2.11 and 3.9). These sections may include references to concepts and terms (such as shading language variable types) defined in the companion document.

OpenGL 3.1 implementations are guaranteed to support at least version 1.30 of the shading language. The actual version supported may be queried as described in section 6.1.4.

1.6.2 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 H).

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

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

The *Khronos Native Platform Graphics Interface* or “EGL Specification” describes the EGL API for use of OpenGL ES on mobile and embedded devices.

EGL implementations may be available supporting OpenGL as well. The EGL Specification is available in the Khronos Extension Registry at URL

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

Chapter 2

OpenGL Operation

2.1 OpenGL Fundamentals

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

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

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

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

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

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

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

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

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

Allocation and configuration of the default framebuffer occurs outside of the GL in conjunction with the window system, using companion APIs described in section 1.6.2.

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:

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

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

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

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

$$M = N \mod 1024.$$

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

2.1.3 Unsigned 11-Bit Floating-Point Numbers

An unsigned 11-bit floating-point number has no sign bit, a 5-bit exponent (E), and a 6-bit mantissa (M). The value V of an unsigned 11-bit floating-point number is 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 3.8.1), b is the number of bits allocated to that component in the internal format of the texture or renderbuffer. When the integer is a framebuffer color or depth component (see section 4, b is the number of bits allocated to that component in the framebuffer. For framebuffer and renderbuffer A components, b must be at least 2 if the buffer does not contain an A component, or if there is only 1 bit of A in the buffer.

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

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

$$f = \frac{c}{2^b - 1}. \tag{2.1}$$

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:

$$f = \frac{2c + 1}{2^b - 1} \tag{2.2}$$

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

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

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 1.6.2. These APIs are responsible for creation and management of contexts, and not discussed further here. More detailed discussion of the behavior of shared objects is included in appendix D. Except as defined in this appendix, all state in a context is specific to that context only.

2.3 GL Command Syntax

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

GL commands are formed from a *name* followed, depending on the particular command, by up to 4 characters. The first character indicates the number of values of the indicated type that must be presented to the command. The second character or character pair indicates the specific type of the arguments: 8-bit integer, 16-bit integer, 32-bit integer, single-precision floating-point, or double-precision floating-point. The final character, if present, is `v`, indicating that the command takes a pointer to an array (a vector) of values rather than a series of individual arguments. Two specific examples are:

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

and

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

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

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

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

rtype **Name**$\{\epsilon$**1234**$\}\{\epsilon$ **b s i 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 characters (or character pairs) of which one is selected. ϵ indicates no character. The arguments enclosed in brackets (*[args ,]* and *[, args]*) may or may not be present. The N arguments *arg1* through *argN* have type *T*, which corresponds to one of the type letters or letter pairs as indicated in table 2.1 (if there are no letters, then the arguments' type is given explicitly). If the final character is not **v**, then N is given by the digit **1**, **2**, **3**, or **4** (if there is no digit, then the number of arguments is fixed). If the final character is **v**, then only *arg1* is present and it is an array of N values of the indicated type. Finally, we indicate an `unsigned` type by the shorthand of prepending a `u` to the beginning of the type name (so that, for instance, `unsigned byte` is abbreviated `ubyte`).

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. Commands are effectively sent through a processing pipeline.

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

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

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

GL Type	Minimum Bit Width	Description
`boolean`	1	Boolean
`byte`	8	Signed 2's complement binary integer
`ubyte`	8	Unsigned binary integer
`char`	8	Characters making up strings
`short`	16	Signed 2's complement binary integer
`ushort`	16	Unsigned binary integer
`int`	32	Signed 2's complement binary integer
`uint`	32	Unsigned binary integer
`sizei`	32	Non-negative binary integer size
`enum`	32	Enumerated binary integer value
`intptr`	*ptrbits*	Signed 2's complement binary integer
`sizeiptr`	*ptrbits*	Non-negative binary integer size
`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` and `sizeiptr` must be sufficiently large as to store any address.

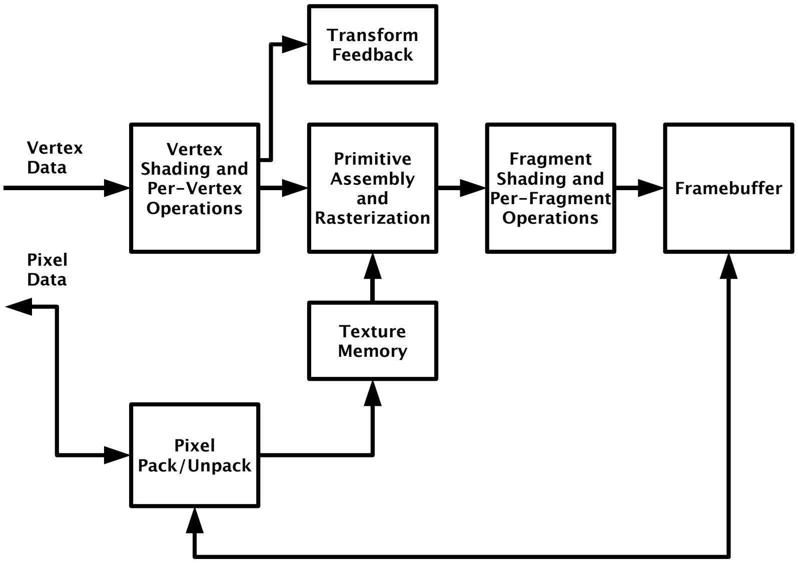

Transform Feedback
Vertex Data
Vertex Shading and Per-Vertex Operations
Primitive Assembly and Rasterization
Fragment Shading and Per-Fragment Operations
Framebuffer
Pixel Data
Texture Memory
Pixel Pack/Unpack

Figure 2.1. Block diagram of the GL.

2.5 GL Errors

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

The command

`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.

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
`OUT_OF_MEMORY`	Not enough memory left to execute command	Unknown

Table 2.3: Summary of GL errors

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

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

2.6 Primitives and Vertices

In the GL, most geometric objects are drawn by specifying a series of generic attribute sets using **DrawArrays** or one of the other drawing commands defined in section 2.8.2. There are seven geometric objects that are drawn this way: points, line segment strips, line segment loops, separated line segments, triangle strips, triangle fans, and separated triangles,

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

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

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

Figure 2.2 shows the sequence of operations that builds a *primitive* (point, line segment, or polygon) from a sequence of vertices. After a primitive is formed, it

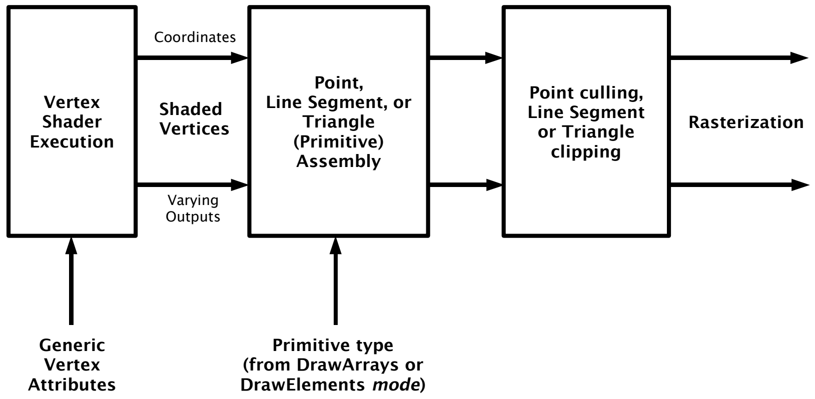

Vertex Shader Execution
Coordinates
Shaded Vertices
Varying Outputs
Point, Line Segment, or Triangle (Primitive) Assembly
Point culling, Line Segment or Triangle clipping
Rasterization
Generic Vertex Attributes
Primitive type (from DrawArrays or DrawElements *mode*)

Figure 2.2. Vertex processing and primitive assembly.

is clipped to a viewing volume. This may alter the primitive by altering vertex coordinates and varying vertex shader outputs. In the case of line and polygon primitives, clipping may insert new vertices into the primitive. The vertices defining a primitive to be rasterized have varying outputs associated with them.

2.6.1 Primitive Types

A sequence of vertices is passed to the GL using **DrawArrays** or one of the other drawing commands defined in section 2.8.2. There is no limit to the number of vertices that may be specified, other than the size of the vertex arrays. The *mode* parameter of these commands determines the type of primitives to be drawn using the vertices. The types, and the corresponding *mode* parameters, are:

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

Line Strips. A series of one or more connected line segments may be specified with *mode* `LINE_STRIP`. In this case, the first vertex specifies the first segment's start point while the second vertex specifies the first segment's endpoint and the second segment's start point. In general, the 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 passed define the first segment, with subsequent pairs of vertices each defining one more segment. If the number of specified vertices is odd, then the last one is ignored. The state required is the same as for line strips but it is used differently: a processed vertex holding the first vertex of the current segment, and a boolean flag indicating whether the current vertex is odd or even (a segment start or end).

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

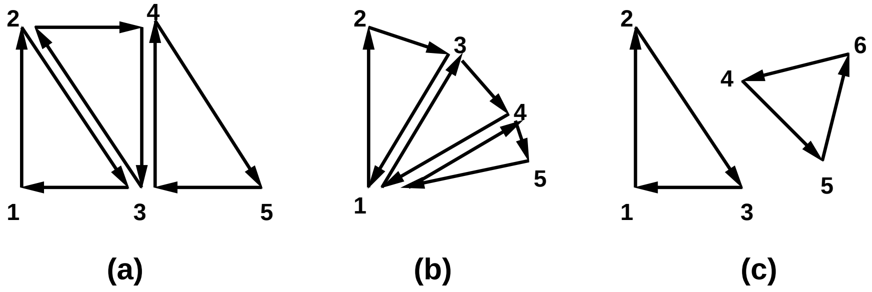


Figure 2.3. (a) A triangle strip. (b) A triangle fan. (c) Independent triangles. The numbers give the sequencing of the vertices in order within the vertex arrays. Note that in (a) and (b) triangle edge ordering is determined by the first triangle, while in (c) the order of each triangle's edges is independent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

produced. See figure 2.3.

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

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

Separate Triangles. Separate triangles are specified with *mode* `TRIANGLES`. In this case, The $3i + 1$st, $3i + 2$nd, and $3i + 3$rd vertices (in that order) determine a triangle for each $i = 0, 1, \dots, n - 1$, 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.

Depending on the current state of the GL, a *polygon primitive* generated from a drawing command with *mode* `TRIANGLE_FAN`, `TRIANGLE_STRIP`, or `TRIANGLES` may be rendered in one of several ways, such as outlining its border or filling its interior. The order of vertices in such a primitive is significant in polygon rasterization and fragment shading (see sections 3.6.1 and 3.9.2).

2.7 Vertex Specification

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

Current generic attribute values define generic attributes for a vertex when a vertex array defining that data is not enabled, as described in section 2.8. The current values of a generic shader attribute declared as a floating-point scalar, vector, 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`.

The state required to support vertex specification consists of the value of `MAX_VERTEX_ATTRIBS` four-component vectors to store generic vertex attributes.

The initial values for all generic vertex attributes are $(0.0, 0.0, 0.0, 1.0)$.

2.8 Vertex Arrays

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

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. *size* indicates the number of values per vertex (1, 2, 3, or 4) that are stored in the array. Table 2.4 indicates the allowable values for *size* and *type* (when present). For *type* the values `BYTE`, `SHORT`, `INT`, `FLOAT`, `HALF_FLOAT`, and `DOUBLE` indicate types `byte`, `short`, `int`, `float`, `half`, and `double`, respectively; and the values `UNSIGNED_BYTE`, `UNSIGNED_SHORT`, and `UNSIGNED_INT` indicate types `ubyte`, `ushort`, and `uint`, respectively. The error `INVALID_VALUE` is generated if *size* is specified with a value other than that indicated in the table.

Command	Sizes	Integer Handling	Types
VertexAttribPointer	1,2,3,4	flag	`byte`, `ubyte`, `short`, `ushort`, `int`, `uint`, `float`, `half`, `double`
VertexAttribIPointer	1,2,3,4	integer	`byte`, `ubyte`, `short`, `ushort`, `int`, `uint`

Table 2.4: Vertex array sizes (values per vertex) and data types. The "Integer Handling" column indicates how fixed-point data types are handled: "integer" means that they remain as integer values, and "flag" means that they are either converted to floating-point directly, or converted by normalizing to $[0, 1]$ (for unsigned types) or $[-1, 1]$ (for signed types), depending on the setting of the *normalized* flag in **VertexAttribPointer**.

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

The one, two, three, or four values in an array that correspond to a single vertex comprise an array *element*. The values within each array element are stored sequentially in memory. If *stride* is specified as zero, then array elements are stored sequentially as well. The error `INVALID_VALUE` is generated if *stride* is negative. Otherwise pointers to the 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 offset within a buffer of the first value of the first element of the array being specified.

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

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

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

2.8.1 Transferring Array Elements

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

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

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

and

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

with *target* `PRIMITIVE_RESTART`. The command

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

specifies a vertex array element that is treated specially when primitive restarting is enabled. This value is called the *primitive restart index*. When one of the **Draw*** commands transfers the ith successive set of generic attribute array elements to the GL, if i [2] is equal to the primitive restart index, then the GL does not process those elements as a vertex. Instead, it is as if the drawing command ended with the immediately preceding transfer, and another drawing command is immediately started with the same parameters, but only transferring elements $i + 1$ through the end of the originally specified elements.

---

[2] Note that i is used here to index the successively transferred attribute sets, so that the first set transferred has $i = 0$, the second set transferred has $i = 1$, and so on. i is not the same as the index of a set within the attribute arrays, which will also depend on either the *first* or *indices* arguments, depending on which drawing command is used.

2.8.2 Drawing Commands

The command

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

constructs a sequence of geometric primitives by transferring elements *first* through $first + count - 1$ of each enabled array to the GL. *mode* specifies what kind of primitives are constructed, as defined in section 2.6.1. If an array corresponding to a generic attribute required by a vertex shader is not enabled, then the corresponding element is taken from the current generic attribute state (see section 2.7).

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

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

The command

`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 by successively transferring the *count* elements whose indices are stored in the currently bound element array buffer (see section 2.9.5) at the offset defined by *indices* to the GL. The ith element transferred by **DrawElements** will be taken from element $indices[i]$ of each enabled array. *type* must be one of `UNSIGNED_BYTE`, `UNSIGNED_SHORT`, or `UNSIGNED_INT`, indicating that the index values are of GL type `ubyte`, `ushort`, or `uint` respectively. *mode* specifies what kind of primitives are constructed, as defined in section 2.6.1. If an array corresponding to a generic attribute required by a vertex shader is not enabled, then the corresponding element is taken from the current generic attribute state (see section 2.7).

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

The command

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

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

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

The command

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

is a restricted form of **DrawElements**. *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 indices to lie outside the range $[start, end]$, but implementations may not 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 2.11.7. 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;
}
```

If the number of supported generic vertex attributes (the value of `MAX_VERTEX_ATTRIBS`) is n, then the client state required to implement vertex arrays consists of n boolean values, n memory pointers, n integer stride values,

n symbolic constants representing array types, n integers representing values per element, n boolean values indicating normalization, n boolean values indicating whether the attribute values are pure integers, and an unsigned integer representing the restart index.

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

2.9 Buffer Objects

Vertex array data are stored in high-performance server memory. GL buffer objects provide a mechanism that clients can use to allocate, initialize, and render from such memory.

The command

`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.

A buffer object is created by binding a name returned by **GenBuffers** 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 2.5. If the buffer object named *buffer* has not been previously bound, the GL creates a new state vector, initialized with a zero-sized memory buffer and comprising the state values listed in table 2.6.

Buffer objects created by binding a name returned by **GenBuffers** 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.

Target name	Purpose	Described in section(s)
`ARRAY_BUFFER`	Vertex attributes	2.9.4
`COPY_READ_BUFFER`	Buffer copy source	2.9.3
`COPY_WRITE_BUFFER`	Buffer copy destination	2.9.3
`ELEMENT_ARRAY_BUFFER`	Vertex array indices	2.9.5
`PIXEL_PACK_BUFFER`	Pixel read target	4.3.1, 6.1
`PIXEL_UNPACK_BUFFER`	Texture data source	3.7
`TEXTURE_BUFFER`	Texture data buffer	3.8.4
`TRANSFORM_FEEDBACK_BUFFER`	Transform feedback buffer	2.15
`UNIFORM_BUFFER`	Uniform block storage	2.11.4

Table 2.5: Buffer object binding targets.

Name	Type	Initial Value	Legal Values
`BUFFER_SIZE`	integer	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`	integer	0	See section 2.9.1
`BUFFER_MAPPED`	boolean	`FALSE`	`TRUE`, `FALSE`
`BUFFER_MAP_POINTER`	void*	`NULL`	address
`BUFFER_MAP_OFFSET`	integer	0	any non-negative integer
`BUFFER_MAP_LENGTH`	integer	0	any non-negative integer

Table 2.6: Buffer object parameters and their values.

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

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

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

If a buffer object is deleted while it is bound, all bindings to that object in the current context (i.e. in the thread that called **DeleteBuffers**) are reset to zero. Bindings to that buffer in other contexts and other threads are not affected, but 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.

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 2.5. *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.

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 2.7: Buffer object initial state.

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

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

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

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

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

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

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

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

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

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

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

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

with *target* set to one of the targets listed in table 2.5. *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 2.9.1).

2.9.1 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 2.5. *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

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 2.8: 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`.

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 2.8.

Errors

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

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

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

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

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

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

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

`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* value passed to **MapBufferRange** equal to

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

`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 2.5. *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 2.5. 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 2.7.

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

Any GL command that attempts to read data from a buffer object will fail and generate an `INVALID_OPERATION` error if the object is mapped at the time the command is issued.

2.9.2 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.

2.9.3 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 2.5. 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.

2.9.4 Vertex Arrays in Buffer Objects

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

Rendering commands **DrawArrays**, and the other drawing commands defined in section 2.8.2 operate as previously defined, where data for enabled generic attribute arrays are sourced from buffer objects. When an array is sourced from a buffer object, the pointer value of that array is used to compute an offset, in basic machine units, into the data store of the buffer object. This offset is computed by subtracting a null pointer from the pointer value, where both pointers are treated as pointers to basic machine units.

If any enabled array's buffer binding is zero when **DrawArrays** or one of the other drawing commands defined in section 2.8.2 is called, the result is undefined.

2.9.5 Array Indices in Buffer Objects

Blocks of array indices are stored in buffer objects in the formats described by the *type* parameter of **DrawElements** (see section 2.8.2).

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

DrawElements, **DrawRangeElements**, and **DrawElementsInstanced** source their indices from the buffer object whose name is bound to `ELEMENT_ARRAY_BUFFER`, using their *indices* parameters as offsets into the buffer object in the same fashion as described in section 2.9.4. **MultiDrawElements** also sources its indices from that buffer object, using its *indices* parameter as a pointer to an array of pointers that represent offsets into the buffer object. If zero is bound to `ELEMENT_ARRAY_BUFFER`, the result of these drawing commands is undefined.

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

2.9.6 Buffer Object State

The state required to support buffer objects consists of binding names for the array buffer, element buffer, pixel unpack buffer, and pixel pack buffer. Additionally, each vertex array has an associated binding so there is a buffer object binding for each of the vertex attribute arrays. The initial values for all buffer object bindings is zero.

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

2.10 Vertex Array Objects

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

The command

`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, just as if they were unused.

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 6.3 and 6.4.

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**.

An `INVALID_OPERATION` error is generated if any of the ***Pointer** commands specifying the location and organization of vertex array data are called while zero is bound to the `ARRAY_BUFFER` buffer object binding point, and the *pointer* argument is not `NULL`[3].

2.11 Vertex Shaders

Vertex shaders describe the operations that occur on vertex values and their associated data.

---

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

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

To use a vertex shader, shader source code is first loaded into a *shader object* and then *compiled*. One or more vertex shader objects are then attached to a *program object*. A program object is then *linked*, which generates executable code from all the compiled shader objects attached to the program. When a linked program object is used as the current program object, the executable code for the vertex shaders it contains is used to process vertices.

In addition to vertex shaders, *fragment shaders* can be created, compiled, and linked into program objects. Fragment shaders affect the processing of fragments during rasterization, and are described in section 3.9. A single program object can contain both vertex and fragment shaders.

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

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

2.11.1 Shader Objects

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

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

To create a shader object, use the command

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

The shader object is empty when it is created. The *type* argument specifies the type of shader object to be created. For vertex shaders, *type* must be `VERTEX_SHADER`.

A non-zero name that can be used to reference the shader object is returned. If an error occurs, zero will be returned.

The command

`void` **ShaderSource**(`uint` *shader*, `sizei` *count*, `const char` ***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 6.1.9). 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 6.1.9).

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 6.1.9). **DeleteShader** will silently ignore the value zero.

2.11.2 Program Objects

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

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

Program objects are empty when they are created. A non-zero name that can be used to reference the program object is returned. If an error occurs, 0 will be returned.

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

`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 6.1.9). This status will be set to `TRUE` if a valid executable is created, and `FALSE` otherwise. Linking can fail for a variety of reasons as specified in the OpenGL Shading Language Specification. Linking will also fail if one or more of the shader objects, attached to *program* are not compiled successfully, or if more active uniform or active sampler variables are used in *program* than allowed (see section 2.11.5). 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 6.1.9).

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

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

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

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

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

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

Program objects can be deleted with the command void **DeleteProgram**(uint *program*);

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

2.11.3 Vertex Attributes

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

When an attribute variable declared as a `float`, `vec2`, `vec3` or `vec4` is bound to a generic attribute index i, its value(s) are taken from the x, (x, y), (x, y, z), or (x, y, z, w) components, respectively, of the generic attribute i. When an attribute variable is declared as a `mat2`, `mat3x2` or `mat4x2`, its matrix columns are taken from the (x, y) components of generic attributes i and $i+1$ (`mat2`), from attributes i through $i + 2$ (`mat3x2`), or from attributes i through $i + 3$ (`mat4x2`). When an attribute variable is declared as a `mat2x3`, `mat3` or `mat4x3`, its matrix columns are taken from the (x, y, z) components of generic attributes i and $i+1$ (`mat2x3`), from attributes i through $i+2$ (`mat3`), or from attributes i through $i+3$ (`mat4x3`). When an attribute variable is declared as a `mat2x4`, `mat3x4` or `mat4`, its matrix columns are taken from the (x, y, z, w) components of generic attributes i and $i+1$ (`mat2x4`), from attributes i through $i + 2$ (`mat3x4`), or from attributes i through $i + 3$ (`mat4`).

A generic attribute variable is considered *active* if it is determined by the compiler and linker that the attribute may be accessed when the shader is executed. Attribute variables that are declared in a vertex shader but never used will not count against the limit. In cases where the compiler and linker cannot make a conclusive determination, an attribute will be considered active. A program object will fail to link if the number of active vertex attributes exceeds `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 6.1.9). 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 must be the name of a generic attribute. The length of the longest attribute name in *program* is given by `ACTIVE_ATTRIBUTE_MAX_LENGTH`, which can be queried with **GetProgramiv** (see section 6.1.9).

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

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

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

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

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

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

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

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

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

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

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

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

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

2.11.4 Uniform Variables

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

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

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

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

Similarly, when a program is successfully linked, all active uniforms belonging to the program's named uniform blocks are assigned offsets (and strides for array and matrix type uniforms) within the uniform block according to layout rules described below. Uniform buffer objects provide the storage for named uniform blocks, so the values of active uniforms in named uniform blocks may be changed by modifying the contents of the buffer object using commands such as **BufferData**, **BufferSubData**, **MapBuffer**, and **UnmapBuffer**. Uniforms in a named uniform block are not assigned a location and may be 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* 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 6.1.9). 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 2.11.4).

The total amount of buffer object storage available for any given uniform block is subject to an implementation-dependent limit. The maximum amount of available space, in basic machine units, can be queried by calling **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` or `UNIFORM_BLOCK_REFERENCED_BY_FRAGMENT_SHADER`, then a boolean value indicating whether the uniform block identified by *uniformBlockIndex* is referenced by the vertex 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 2.9, 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*.

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 2.9.

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.

Type Name Token	Keyword	Type Name Token	Keyword
`FLOAT`	`float`	`SAMPLER_1D`	`sampler1D`
`FLOAT_VEC2`	`vec2`	`SAMPLER_2D`	`sampler2D`
`FLOAT_VEC3`	`vec3`	`SAMPLER_3D`	`sampler3D`
`FLOAT_VEC4`	`vec4`	`SAMPLER_CUBE`	`samplerCube`
`INT`	`int`	`SAMPLER_1D_SHADOW`	`sampler1DShadow`
`INT_VEC2`	`ivec2`	`SAMPLER_2D_SHADOW`	`sampler2DShadow`
`INT_VEC3`	`ivec3`	`SAMPLER_1D_ARRAY`	`sampler1DArray`
`INT_VEC4`	`ivec4`	`SAMPLER_2D_ARRAY`	`sampler2DArray`
`UNSIGNED_INT`	`unsigned int`	`SAMPLER_1D_ARRAY_SHADOW`	`sampler1DArraySh`
`UNSIGNED_INT_VEC2`	`uvec2`	`SAMPLER_2D_ARRAY_SHADOW`	`sampler2DArraySh`
`UNSIGNED_INT_VEC3`	`uvec3`	`SAMPLER_CUBE_SHADOW`	`samplerCubeShado`
`UNSIGNED_INT_VEC4`	`uvec4`	`SAMPLER_2D_RECT`	`sampler2DRect`
`BOOL`	`bool`	`SAMPLER_2D_RECT_SHADOW`	`sampler2DRectSha`
`BOOL_VEC2`	`bvec2`	`INT_SAMPLER_1D`	`isampler1D`
`BOOL_VEC3`	`bvec3`	`INT_SAMPLER_2D`	`isampler2D`
`BOOL_VEC4`	`bvec4`	`INT_SAMPLER_3D`	`isampler3D`
`FLOAT_MAT2`	`mat2`	`INT_SAMPLER_CUBE`	`isamplerCube`
`FLOAT_MAT3`	`mat3`	`INT_SAMPLER_1D_ARRAY`	`isampler1DArray`
`FLOAT_MAT4`	`mat4`	`INT_SAMPLER_2D_ARRAY`	`isampler2DArray`
`FLOAT_MAT2x3`	`mat2x3`	`UNSIGNED_INT_SAMPLER_1D`	`usampler1D`
`FLOAT_MAT2x4`	`mat2x4`	`UNSIGNED_INT_SAMPLER_2D`	`usampler2D`
`FLOAT_MAT3x2`	`mat3x2`	`UNSIGNED_INT_SAMPLER_3D`	`usampler3D`
`FLOAT_MAT3x4`	`mat3x4`	`UNSIGNED_INT_SAMPLER_CUBE`	`usamplerCube`
`FLOAT_MAT4x2`	`mat4x2`	`UNSIGNED_INT_SAMPLER_1D_ARRAY`	`usampler1DArray`
`FLOAT_MAT4x3`	`mat4x3`	`UNSIGNED_INT_SAMPLER_2D_ARRAY`	`usampler2DArray`

Table 2.9: OpenGL Shading Language type tokens returned by **GetActiveUniform** and **GetActiveUniformsiv**, and corresponding shading language keywords declaring each such type.

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 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 and fragment shaders can be obtained by calling **GetIntegerv** with *pname* values of `MAX_VERTEX_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`x`R, 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`x`R, 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`.

Buffer objects are bound to uniform block binding points by calling one of the commands

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

with *target* set to `UNIFORM_BUFFER`. There is an array of buffer object binding points with which uniform blocks can be associated via **UniformBlockBinding**, plus 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 the general binding point, and additionally bind the buffer object to the binding point in the array given by *index*. The error `INVALID_VALUE` is generated if *index* is greater than or equal to the value of `MAX_UNIFORM_BUFFER_BINDINGS`.

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 the storage for a uniform block. Both *offset* and *size* are in basic machine units. The error `INVALID_VALUE` is generated if the value of *size* is less than or equal to zero, if $offset + size$ is greater than the value of `BUFFER_SIZE`, or if *offset* is not a multiple of the implementation-dependent required alignment (`UNIFORM_BUFFER_OFFSET_ALIGNMENT`). **BindBufferBase** is equivalent to calling **BindBufferRange** with *offset* zero and *size* equal to the size of *buffer*.

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.

uniformBlockIndex must be an active uniform block index of the program *program*. Otherwise, `INVALID_VALUE` is generated.

uniformBlockBinding must be less than `MAX_UNIFORM_BUFFER_BINDINGS`. Otherwise, `INVALID_VALUE` is generated.

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

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

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

2.11.5 Samplers

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

The type of the sampler identifies the target on the texture image unit. The texture object bound to that texture image unit's target is then used for the texture lookup. 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.

2.11.6 Varying Variables

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

The number of interpolators available for processing varying variables is given by the value of the implementation-dependent constant `MAX_VARYING_COMPONENTS`. This value represents the number of individual scalar numeric values that can be interpolated; varying variables declared as vectors, matrices, and arrays will all consume multiple interpolators. When a program is linked, all components of any varying variable written by a vertex shader, read by a fragment shader, or used for transform feedback will count against this limit. The transformed vertex position (`gl_Position`) is not a varying variable and does not count against this limit. A program whose shaders access more than the value of `MAX_VARYING_COMPONENTS` components worth of varying variables may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

Each program object can specify a set of one or more varying variables to be recorded in transform feedback mode with the command

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

program specifies the program object. *count* specifies the number of varying variables used for transform feedback. *varyings* is an array of *count* zero-terminated strings specifying the names of the varying variables to use for transform feedback. Varying variables are written out in the order they appear in the array *varyings*. *bufferMode* is either `INTERLEAVED_ATTRIBS` or `SEPARATE_ATTRIBS`, and identifies the mode used to capture the varying variables when transform feedback is active. The error `INVALID_VALUE` is generated if *program* is not the name of a program object, or if *bufferMode* is `SEPARATE_ATTRIBS` and *count* is greater than the value of the implementation-dependent limit `MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`.

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

- the *count* specified by **TransformFeedbackVaryings** is non-zero, but the program object has no vertex shader;
- any variable name specified in the *varyings* array is not declared as an output in the vertex shader.
- any two entries in the *varyings* array specify the same varying variable;
- the total number of components to capture in any varying variable in *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 6.1.9). 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 length of the longest varying name in *program* is given by `TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH`, which can be queried with **GetProgramiv** (see section 6.1.9).

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

2.11.7 Shader Execution

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

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

- Perspective division on clip coordinates (section 2.12).
- Viewport mapping, including depth range scaling (section 2.12.1).
- Clipping, including client-defined half-spaces (section 2.17).
- Front face determination (section 3.6.1).
- generic attribute clipping (section 2.17.1).

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

Shader Only Texturing

This section describes texture functionality that is accessible through vertex or fragment shaders. Also refer to section 3.8 and to the OpenGL Shading Language Specification, section 8.7.

Texel Fetches

The OpenGL Shading Language texel fetch functions provide the ability to extract a single texel from a specified texture image. The integer coordinates passed to the texel fetch functions are used directly as the texel coordinates (i, j, k) into the texture image. This in turn means the texture image is point-sampled (no filtering is performed).

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

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

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

- the computed LOD is less than the texture's base level ($level_{base}$) or greater than the maximum level ($level_{max}$)
- the computed LOD is not the texture's base level and the texture's minification filter is `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 border texel outside the defined extents of the specified LOD, 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 w_s, h_s, d_s, and b_s refer to the width, height, depth, and border size of the image, as in equations 3.16
- the texture being accessed is not complete (or cube complete for cubemaps).

Texture Size Query

The OpenGL Shading Language texture size functions provide the ability to query the size of a texture image. The LOD value *lod* passed in as an argument to the texture size functions is added to the $level_{base}$ of the texture to determine a texture image level. The dimensions of that image level, excluding a possible border, are then returned. If the computed texture image level is outside the range

$[level_{base}, level_{max}]$, the results are undefined. When querying the size of an array texture, both the dimensions and the layer index are returned.

Texture Access

Vertex shaders have the ability to do a lookup into a texture map. The maximum number of texture image units available to a vertex shader is the value of the implementation-dependent constant `MAX_VERTEX_TEXTURE_IMAGE_UNITS`. The maximum number of texture image units available to a fragment shader is the value of `MAX_TEXTURE_IMAGE_UNITS`. Both the vertex shader and fragment shader combined cannot use more than the value of `MAX_COMBINED_TEXTURE_IMAGE_UNITS` texture image units. If both the vertex shader and the fragment processing stage access the same texture image unit, then that counts as using two texture image units against the `MAX_COMBINED_TEXTURE_IMAGE_UNITS` limit.

When a texture lookup is performed in a vertex shader, the filtered texture value τ is computed in the manner described in sections 3.8.8 and 3.8.9, and converted to a texture source color C_s according to table 3.20 (section 3.9.2). A four-component vector (R_s, G_s, B_s, A_s) is returned to the vertex 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 shader, it is not possible to perform automatic level-of-detail calculations using partial derivatives of the texture coordinates with respect to window coordinates as described in section 3.8.8. 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 3.17). 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 3.21) are ignored.

Texture lookups involving textures with depth component data can either return the depth data directly or return the results of a comparison with a reference depth value specified in the coordinates passed to the texture lookup function, as described in section 3.8.14. 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`.

If a vertex shader uses a sampler where the associated texture object is not complete, as defined in section 3.8.11, the texture image unit will return (R, G, B, A) = $(0, 0, 0, 1)$.

Shader Inputs

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

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

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

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

Shader Outputs

A vertex shader can write to user-defined varying variables. These values are expected to be interpolated across the primitive it outputs, unless they are specified to be flat shaded. Refer to the OpenGL Shading Language specification sections 4.3.6, 7.1 and 7.6 for more detail.

The built-in special variable `gl_Position` is intended to hold the homogeneous vertex position. Writing `gl_Position` is optional.

The built-in special variable `gl_ClipDistance` holds the clip distance(s) used in the clipping stage, as described in section 2.17. If clipping is enabled, `gl_ClipDistance` should be written.

Validation

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

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

- any two active samplers in the current program object are of different types, but refer to the same texture image unit,
- the number of active samplers in the program exceeds the maximum number of texture image units allowed.

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

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

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

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

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

Undefined Behavior

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

2.11.8 Required State

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

The state required per shader object consists of:

- 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 vertex program point size mode (section 3.4.1) is enabled, initially disabled.

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

2.12 Coordinate Transformations

Clip coordinates for a vertex result from vertex 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

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

2.12.1 Controlling the Viewport

The viewport transformation is determined by the viewport's width and height in pixels, p_x and p_y, respectively, and its center (o_x, o_y) (also in pixels). The vertex's window coordinates, $\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix}$, are given by

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

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

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

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.

2.13 Asynchronous Queries

Asynchronous queries provide a mechanism to return information about the processing of a sequence of GL commands. There are two query types supported by the GL. Transform feedback queries (see section 2.15) return information on the number of vertices and primitives processed by the GL and written to one or more buffer objects. Occlusion queries (see section 4.1.6) count the number of fragments or samples that pass the depth test.

The results of asynchronous queries are not returned by the GL immediately after the completion of the last command in the set; subsequent commands can be processed while the query results are not complete. When available, the query results are stored in an associated query object. The commands described in section 6.1.6 provide mechanisms to determine when query results are available and return the actual results of the query. The name space for query objects is the unsigned integers, with zero reserved by the GL.

Each type of query supported by the GL has an active query object name. If the active query object name for a query type is non-zero, the GL is currently tracking the information corresponding to that query type and the query results will be written into the corresponding query object. If the active query object for a query type name is zero, no such information is being tracked.

A query object is created and made active by calling

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

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

BeginQuery fails and an `INVALID_OPERATION` error is generated if *id* is not a name returned from a previous call to **GenQueries**, or if such a name has since been deleted with **DeleteQueries**.

BeginQuery sets the active query object name for the query type given by *target* to *id*. If **BeginQuery** is called with an *id* of zero, if the active query object name for *target* is non-zero, if *id* is the name of an existing query object whose type does not match *target*, if *id* is the active query object name for any query type, or if *id* is the active query object for condtional rendering (see section 2.14), 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.

2.14 Conditional Rendering

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

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

id specifies the name of an occlusion query object whose results are used to determine if the rendering commands are discarded. If the result (`SAMPLES_PASSED`) of the query is zero, all rendering commands between **BeginConditionalRender** and the corresponding **EndConditionalRender** are discarded. In this case, all vertex array commands (see section 2.8), as well as **Clear** and **ClearBuffer*** (see section 4.2.3), have no effect. The effect of commands setting current vertex state, such as **VertexAttrib**, are undefined. If the result of the occlusion query is non-zero, such commands are not discarded.

mode specifies how **BeginConditionalRender** interprets the results of the occlusion query given by *id*. If *mode* is `QUERY_WAIT`, the GL waits for the results of the query to be available and then uses the results to determine if 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.

Transform Feedback *primitiveMode*	Allowed render primitive *modes*
`POINTS`	`POINTS`
`LINES`	`LINES`, `LINE_LOOP`, `LINE_STRIP`
`TRIANGLES`	`TRIANGLES`, `TRIANGLE_STRIP`, `TRIANGLE_FAN`

Table 2.10: Legal combinations of the transform feedback primitive mode, as passed to **BeginTransformFeedback**, and the current primitive mode.

2.15 Transform Feedback

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

Transform feedback is started and finished by calling

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

and

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

respectively. Transform feedback is said to be active after a call to **BeginTransformFeedback** and inactive after a call to **EndTransformFeedback**. *primitiveMode* is one of `TRIANGLES`, `LINES`, or `POINTS`, and specifies the output type of primitives that will be recorded into the buffer objects bound for transform feedback (see below). *primitiveMode* restricts the primitive types that may be rendered while transform feedback is active, as shown in table 2.10.

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

Transform feedback mode captures the values of varying variables written by the vertex shader.

When transform feedback is active, all geometric primitives generated must be compatible with the value of *primitiveMode* passed to **BeginTransformFeedback**. The error `INVALID_OPERATION` is generated by **DrawArrays** and the other drawing commands defined in section 2.8.2 if *mode* is not one of the allowed modes in table 2.10.

Buffer objects are made to be targets of transform feedback by calling one of the commands

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

with *target* set to `TRANSFORM_FEEDBACK_BUFFER`. There is an array of buffer object binding points that are used while transform feedback is active, plus a single general binding point that can be used by other buffer object manipulation functions (e.g., **BindBuffer**, **MapBuffer**). Both commands bind the buffer object named by *buffer* to the general binding point, and additionally bind the buffer object to the binding point in the array given by *index*. The error `INVALID_VALUE` is generated if *index* is greater than or equal to the value of `MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`.

For **BindBufferRange**, *offset* specifies a starting offset into the buffer object *buffer*, and *size* specifies the amount of data that can be written to the buffer object while transform feedback mode is active. Both *offset* and *size* are in basic machine units. The error `INVALID_VALUE` is generated if the value of *size* is less than or equal to zero, if $offset + size$ is greater than the value of `BUFFER_SIZE`, or if either *offset* or *size* are not a multiple of 4. **BindBufferBase** is equivalent to calling **BindBufferRange** with *offset* zero and *size* equal to the size of *buffer*, rounded down to the nearest multiple of 4.

When an individual point, line, or triangle primitive reaches the transform feedback stage while transform feedback is active, the values of the specified varying variables of the vertex are appended to the buffer objects bound to the transform feedback binding points. The attributes of the first vertex received after **BeginTransformFeedback** are written at the starting offsets of the bound buffer objects set by **BindBufferRange**, and subsequent vertex attributes are appended to the buffer object. When capturing line and triangle primitives, all attributes of the first vertex are written first, followed by attributes of the subsequent vertices. When writing varying variables that are arrays, individual array elements are written in order. For multi-component varying variables or varying array elements, the individual components are written in order. The value for any attribute specified to be streamed to a buffer object but not actually written by a vertex shader is undefined.

Individual lines or triangles of a strip or fan primitive will be extracted and recorded separately. Incomplete primitives are not recorded.

Transform feedback can operate in either `INTERLEAVED_ATTRIBS` or `SEPARATE_ATTRIBS` mode. In `INTERLEAVED_ATTRIBS` mode, the values of one or more varyings are written, interleaved, into the buffer object bound to the first transform feedback binding point ($index = 0$). If more than one varying variable is written, they will be recorded in the order specified by **TransformFeedbackVaryings** (see section 2.11.6). In `SEPARATE_ATTRIBS` mode, the first varying variable specified by **TransformFeedbackVaryings** is written to the first transform feedback binding point; subsequent varying variables are written to the subsequent transform feedback binding points. The total number of variables that may be captured in separate mode is given by `MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`.

If recording the vertices of a primitive to the buffer objects being used for transform feedback purposes would result in either exceeding the limits of any buffer object's size, or in exceeding the end position $\mathit{offset} + \mathit{size} - 1$, 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 2.16) is not incremented.

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

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

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

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

2.16 Primitive Queries

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

When **BeginQuery** is called with a *target* of `PRIMITIVES_GENERATED`, the primitives-generated count maintained by the GL is set to zero. When the generated primitive query is active, the primitives-generated count is incremented every time a primitive reaches the "Discarding Primitives Before Rasterization" stage (see section 3.1) immediately before rasterization.

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

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

2.17 Primitive Clipping

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

$$\begin{aligned} -w_c \le x_c \le w_c \\ -w_c \le y_c \le w_c \\ -w_c \le z_c \le w_c. \end{aligned}$$

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

A vertex shader may write a single clip distance for each supported half-space to elements of the `gl_ClipDistance[]` array. Half-space n is then given by the set of points satisfying the inequality

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

where $c_n(P)$ is the value of clip distance n at point P. For point primitives, $c_n(P)$ is simply the clip distance for the vertex in question. For line and triangle primitives, per-vertex clip distances are interpolated using a weighted mean, with weights derived according to the algorithms described in sections 3.5 and 3.6.

Client-defined half-spaces are enabled with the generic **Enable** command and disabled with the **Disable** command. The value of the argument to either command is `CLIP_DISTANCE`i where i is an integer between 0 and $n-1$; specifying a value of i enables or disables the plane equation with index i. The constants obey `CLIP_DISTANCE`i $=$ `CLIP_DISTANCE0` $+ i$.

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

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

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

This clipping produces a value, $0 \leq t \leq 1$, for each clipped vertex. If the coordinates of a clipped vertex are $\mathbf{P}$ and the original vertices' coordinates are $\mathbf{P}_1$ and $\mathbf{P}_2$, then t is given by

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

The value of t is used to clip vertex shader varying variables as described in section 2.17.1.

If the primitive is a polygon, then it is passed if every one of its edges lies entirely inside the clip volume and either clipped or discarded otherwise. Polygon clipping may cause polygon edges to be clipped, but because polygon connectivity must be maintained, these clipped edges are connected by new edges that lie along the clip volume's boundary. Thus, clipping may require the introduction of new vertices into a polygon.

If it happens that a polygon intersects an edge of the clip volume's boundary, then the clipped polygon must include a point on this boundary edge.

Primitives rendered with user-defined half-spaces must satisfy a complementarity criterion. Suppose a series of primitives is drawn where each vertex i has a single specified clip distance d_i (or a number of similarly specified clip distances, if multiple half-spaces are enabled). Next, suppose that the same series of primitives are drawn again with each such clip distance replaced by $-d_i$ (and the GL is otherwise in the same state). In this case, primitives must not be missing any pixels, nor may any pixels be drawn twice in regions where those primitives are cut by the clip planes.

The state required for clipping is at least 8 bits indicating which of the client-defined half-spaces are enabled. In the initial state, all half-spaces are disabled.

2.17.1 Clipping Shader Varying Outputs

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

Let the varying values assigned to the two vertices $\mathbf{P}_1$ and $\mathbf{P}_2$ of an unclipped edge be $\mathbf{c}_1$ and $\mathbf{c}_2$. The value of t (section 2.17) for a clipped point $\mathbf{P}$ is used to obtain the varying value associated with $\mathbf{P}$ as [4]

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

(Multiplying a varying value by a scalar means multiplying each of *x*, *y*, *z*, and *w* by the scalar.)

Polygon clipping may create a clipped vertex along an edge of the clip volume's boundary. This situation is handled by noting that polygon clipping proceeds by clipping against one half-space at a time. Varying value clipping is done in the same way, so that clipped points always occur at the intersection of polygon edges (possibly already clipped) with the clip volume's boundary.

For vertex shader varying variables specified to be interpolated without perspective correction (using the `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.

---

[4] Since this computation is performed in clip space before division by w_c, clipped varying values are perspective-correct.

Chapter 3

Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional image. Each point of this image contains such information as color and depth. Thus, rasterizing a primitive consists of two parts. The first is to determine which squares of an integer grid in window coordinates are occupied by the primitive. The second is assigning a depth value and one or more color values to each such square. The results of this process are passed on to the next stage of the GL (per-fragment operations), which uses the information to update the appropriate locations in the framebuffer. Figure 3.1 diagrams the rasterization process. The color values assigned to a fragment are initially determined by the rasterization operations (sections 3.4 through 3.7) and modified by a fragment shader as defined in section 3.9. The final depth value is initially determined by the rasterization operations and may be modified or replaced by a fragment shader. The results from rasterizing a point, line, or polygon can be routed through a fragment shader.

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

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

Several factors affect rasterization. Primitives may be discarded before rasterization. Points may be given differing diameters and line segments differing

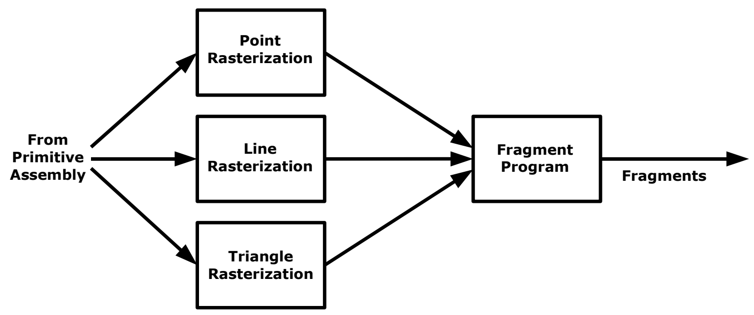


Figure 3.1. Rasterization.

widths. A point, line segment, or polygon may be antialiased.

3.1 Discarding Primitives Before Rasterization

Primitives can be optionally discarded before rasterization by calling **Enable** and **Disable** with `RASTERIZER_DISCARD`. When enabled, primitives are discarded immediately before the rasterization stage, but after the optional transform feedback stage (see section 2.15). When disabled, primitives are passed through to the rasterization stage to be processed normally. When enabled, `RASTERIZER_DISCARD` also causes the **Clear** and **ClearBuffer*** commands to be ignored.

3.2 Invariance

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

3.3 Antialiasing

The R, G, and B values of the rasterized fragment are left unaffected, but the A value is multiplied by a floating-point value in the range $[0, 1]$ that describes a fragment's screen pixel coverage. The per-fragment stage of the GL can be set up to use the A value to blend the incoming fragment with the corresponding pixel already present in the framebuffer.

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

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

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

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

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

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

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

In some implementations, varying degrees of antialiasing quality may be obtained by providing GL hints (section 5.2), allowing a user to make an image quality versus speed tradeoff.

3.3.1 Multisampling

Multisampling is a mechanism to antialias all GL primitives: points, lines, and polygons. The technique is to sample all primitives multiple times at each pixel. The color sample values are resolved to a single, displayable color each time a pixel is updated, so the antialiasing appears to be automatic at the application level. Because each sample includes color, depth, and stencil information, the color (including texture operation), depth, and stencil functions perform equivalently to the single-sample mode.

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

Multisample antialiasing is most valuable for rendering polygons, because it requires no sorting for hidden surface elimination, and it correctly handles adjacent polygons, object silhouettes, and even intersecting polygons. If only lines are being rendered, the "smooth" antialiasing mechanism provided by the base GL may result in a higher quality image. This mechanism is designed to allow multisample and smooth antialiasing techniques to be alternated during the rendering of a single scene.

If the value of `SAMPLE_BUFFERS` is one, the rasterization of all primitives is changed, and is referred to as multisample rasterization. Otherwise, primitive rasterization is referred to as single-sample rasterization. The value of `SAMPLE_BUFFERS` is queried by calling **GetIntegerv** with *pname* set to `SAMPLE_BUFFERS`.

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

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

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

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

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

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

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

3.4 Points

If vertex program point size mode is enabled, then the derived point size is taken from the (potentially clipped) shader built-in `gl_PointSize` 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 vertex program point size mode is disabled, then the derived point size is specified with the command

`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`. Vertex program point size mode is enabled and disabled by calling **Enable** or **Disable** with the symbolic value `VERTEX_PROGRAM_POINT_SIZE`.

If multisampling is enabled, an implementation may optionally fade the point alpha (see section 3.11) instead of allowing the point width to go below a given threshold. In this case, the width of the rasterized point is

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

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 point fade $threshold$ is specified with

void **PointParameter{if}**(enum *pname*, T *param*);
void **PointParameter{if}v**(enum *pname*, const T *params*);

If *pname* is `POINT_FADE_THRESHOLD_SIZE`, then *param* specifies, or *params* points to the point fade $threshold$. Values of `POINT_FADE_THRESHOLD_SIZE` less than zero result in the error `INVALID_VALUE`.

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

3.4.1 Basic Point Rasterization

Point rasterization produces a fragment for each framebuffer pixel whose center lies inside a square centered at the point's (x_w, y_w), with side length equal to the current point size.

All fragments produced in rasterizing a point sprite are assigned the same associated data, which are those of the vertex corresponding to the point. However, the fragment shader builtin `gl_PointCoord` contains point sprite texture coordinates. The s point sprite texture coordinate varies from 0 to 1 across the point horizontally left-to-right. If `POINT_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:

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

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

where $size$ is the point's size, x_f and y_f are the (integral) window coordinates of the fragment, and x_w and y_w are the exact, unrounded window coordinates of the vertex for the point.

Not all point widths need be supported, but the width 1.0 must be provided. The range of supported widths and the width of evenly-spaced gradations within that range are implementation-dependent. The range and gradations may be obtained using the query mechanism described in chapter 6. If, for instance, the width range is from 0.1 to 2.0 and the gradation width is 0.1, then the widths $0.1, 0.2, \ldots, 1.9, 2.0$ are supported. Additional point widths may also be supported. There is no requirement that these widths must be equally spaced. If an unsupported width is requested, the nearest supported width is used instead.

3.4.2 Point Rasterization State

The state required to control point rasterization consists of the floating-point point width, a bit indicating whether or not vertex program point size mode is enabled, a bit for the point sprite texture coordinate origin, and a floating-point value specifying the point fade threshold size.

3.4.3 Point Multisample Rasterization

If `MULTISAMPLE` is enabled, and the value of `SAMPLE_BUFFERS` is one, then points are rasterized using the following algorithm Point rasterization produces a fragment for each framebuffer pixel with one or more sample points that intersect a region centered at the point's (x_w, y_w). This region is a square with side equal to the current point width. Coverage bits that correspond to sample points that intersect the region are 1, other coverage bits are 0. All data associated with each sample for the fragment are the data associated with the point being rasterized, .

The set of point sizes supported is equivalent to those for point sprites without multisample .

3.5 Line Segments

A line segment results from a line strip, a line loop, or a series of separate line segments. Line segment rasterization is controlled by several variables. Line width, which may be set by calling

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

with an appropriate positive floating-point width, controls the width of rasterized line segments. The default width is 1.0. Values less than or equal to 0.0 generate the error `INVALID_VALUE`. Antialiasing is controlled with **Enable** and **Disable** using the symbolic constant `LINE_SMOOTH`.

3.5.1 Basic Line Segment Rasterization

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

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

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

Essentially, a line segment starting at $\mathbf{p}_a$ 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 3.2.

To avoid difficulties when an endpoint lies on a boundary of R_f we (in principle) perturb the supplied endpoints by a tiny amount. Let $\mathbf{p}_a$ 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 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 $\mathbf{p}_b$)

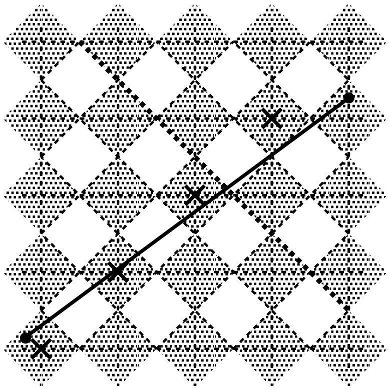


Figure 3.2. Visualization of Bresenham's algorithm. A portion of a line segment is shown. A diamond shaped region of height 1 is placed around each fragment center; those regions that the line segment exits cause rasterization to produce corresponding fragments.

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 a varying shader output or the clip w coordinate, is found as

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

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.

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 generic attribute value of the last (highest numbered) vertex transferred to the GL corresponding to that primitive.

3.5.2 Other Line Segment Features

We have just described the rasterization of non-antialiased line segments of width one. We now describe the rasterization of line segments for general values of the line segment rasterization parameters.

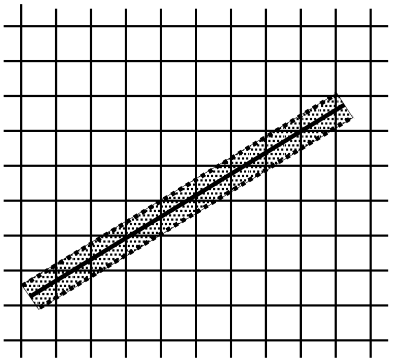


Figure 3.3. The region used in rasterizing and finding corresponding coverage values for an antialiased line segment (an x-major line segment is shown).

Antialiasing

Rasterized antialiased line segments produce fragments whose fragment squares intersect a rectangle centered on the line segment. Two of the edges are parallel to the specified line segment; each is at a distance of one-half the current width from that segment: one above the segment and one below it. The other two edges pass through the line endpoints and are perpendicular to the direction of the specified line segment. Coverage values are computed for each fragment by computing the area of the intersection of the rectangle with the fragment square (see figure 3.3; see also section 3.3). Equation 3.6 is used to compute associated data values just as with non-antialiased lines; equation 3.5 is used to find the value of t for each fragment whose square is intersected by the line segment's rectangle. Not all widths need be supported for line segment antialiasing, but width 1.0 antialiased segments must be provided. As with the point width, a GL implementation may be queried for the range and number of gradations of available antialiased line widths.

3.5.3 Line Rasterization State

The state required for line rasterization consists of the floating-point line width and a bit indicating whether line antialiasing is on or off. The initial value of the line width is 1.0. The initial state of line segment antialiasing is disabled.

3.5.4 Line Multisample Rasterization

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

Coverage bits that correspond to sample points that intersect a retained rectangle are 1, other coverage bits are 0. Each color, depth, and set of texture coordinates is produced by substituting the corresponding sample location into equation 3.5, then using the result to evaluate equation 3.7. An implementation may choose to assign the same color value and the same set of texture coordinates to more than one sample by evaluating equation 3.5 at any location within the pixel including the fragment center or any one of the sample locations, then substituting into equation 3.6. The color value and the set of texture coordinates need not be evaluated at the same location.

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

3.6 Polygons

A polygon results from a triangle arising from a triangle strip, triangle fan, or series of separate triangles. Like points and line segments, polygon rasterization is controlled by several variables. Polygon antialiasing is controlled with **Enable** and **Disable** with the symbolic constant `POLYGON_SMOOTH`.

3.6.1 Basic Polygon Rasterization

The first step of polygon rasterization is to determine if the polygon is *back-facing* or *front-facing*. This determination is made based on the sign of the (clipped or unclipped) polygon's area computed in window coordinates. One way to compute this area is

$$a = \frac{1}{2}\sum_{i=0}^{n-1} x_w^i y_w^{i\oplus 1} - x_w^{i\oplus 1} y_w^i \tag{3.8}$$

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 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.

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 generic attribute value of the last (highest numbered) vertex transferred to the GL corresponding to that primitive.

For a polygon with more than three edges, such as may be produced by clipping a triangle, we require only that a convex combination of the values of the datum at the polygon's vertices can be used to obtain the value assigned to each fragment produced by the rasterization algorithm. That is, it must be the case that at every fragment

$$f = \sum_{i=1}^{n} a_i f_i$$

where n is the number of vertices in the polygon, f_i is the value of the f at vertex i; for each i $0 \leq a_i \leq 1$ and $\sum_{i=1}^{n} a_i = 1$. The values of the a_i may differ from fragment to fragment, but at vertex i, $a_j = 0, j \neq i$ and $a_i = 1$.

One algorithm that achieves the required behavior is to triangulate a polygon (without adding any vertices) and then treat each triangle individually as already discussed. A scan-line rasterizer that linearly interpolates data along each edge and then linearly interpolates data across each horizontal span from edge to edge also satisfies the restrictions (in this case, the numerator and denominator of equation 3.9 should be iterated independently and a division performed for each fragment).

3.6.2 Antialiasing

Polygon antialiasing rasterizes a polygon by producing a fragment wherever the interior of the polygon intersects that fragment's square. A coverage value is computed at each such fragment, and this value is saved to be applied as described in section 3.10. An associated datum is assigned to a fragment by integrating the datum's value over the region of the intersection of the fragment square with the polygon's interior and dividing this integrated value by the area of the intersection. For a fragment square lying entirely within the polygon, the value of a datum at the fragment's center may be used instead of integrating the value across the fragment.

3.6.3 Options Controlling Polygon Rasterization

The interpretation of polygons for rasterization is controlled using

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

face must be `FRONT_AND_BACK`, indicating that the rasterizing method described by *mode* replaces the rasterizing method for both front- and back-facing polygons. *mode* is one of the symbolic constants `POINT`, `LINE`, or `FILL`. Calling **PolygonMode** with `POINT` causes the vertices of a polygon to be treated, for rasterization purposes, as if they had been drawn with *mode* `POINTS`. `LINE` causes edges to be rasterized as line segments. `FILL` is the default mode of polygon rasterization, corresponding to the description in sections 3.6.1, and 3.6.2. Note that these modes affect only the final rasterization of polygons: in particular, a polygon's vertices are lit, and the polygon is clipped and possibly culled before these modes are applied.

Polygon antialiasing applies only to the `FILL` state of **PolygonMode**. For `POINT` or `LINE`, point antialiasing or line segment antialiasing, respectively, apply.

3.6.4 Depth Offset

The depth values of all fragments generated by the rasterization of a polygon may be offset by a single value that is computed for that polygon. The function that determines this value is specified by calling

`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

$$m = \max\left\{\left|\frac{\partial z_w}{\partial x_w}\right|, \left|\frac{\partial z_w}{\partial y_w}\right|\right\}. \tag{3.12}$$

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.

3.6.5 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. If a polygon is culled, based on its orientation and the **CullFace** mode, then no fragments are produced during rasterization.

Coverage bits that correspond to sample points that satisfy the point sampling criteria are 1, other coverage bits are 0. Each associated datum is produced as described in section 3.6.1, but using the corresponding sample location instead of the fragment center. An implementation may choose to assign the same associated data values to more than one sample by barycentric evaluation using any location within the pixel including the fragment center or one of the sample locations.

When using a vertex shader, the `noperspective` and `flat` keywords affect how varying shader outputs are interpolated, as described in the OpenGL Shading Language Specification.

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.

3.6.6 Polygon Rasterization State

The state required for polygon rasterization consists of the current state of polygon antialiasing (enabled or disabled), the current values of the **PolygonMode** setting, whether point, line, and fill mode polygon offsets are enabled or disabled, and the factor and bias values of the polygon offset equation. The initial setting of polygon antialiasing is disabled. The initial state for **PolygonMode** is `FILL` . The initial polygon offset factor and bias values are both 0; initially polygon offset is disabled for all modes.

3.7 Pixel Rectangles

Rectangles of color, depth, and certain other values may be specified to the GL using **TexImage*D** (see section 3.8.1). Some of the parameters and operations governing the operation of these commands are shared by **ReadPixels** (used to obtain pixel values from the framebuffer); the discussion of **ReadPixels**, however, is deferred until chapter 4 after the framebuffer has been discussed in detail. Nevertheless, we note in this section when parameters and state pertaining to these commands also pertain to **ReadPixels**.

A number of parameters control the encoding of pixels in buffer object or client memory (for reading and writing) and how pixels are processed before being placed in or after being read from the framebuffer (for reading, writing, and copying). These parameters are set with **PixelStore**.

3.7.1 Pixel Storage Modes and Pixel Buffer Objects

Pixel storage modes affect the operation of **TexImage*D**, **TexSubImage*D**, and **ReadPixels** when one of these commands is issued. 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.

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 **TexImage1D**, **TexImage2D**, **TexImage3D**, **TexSubImage1D**, **TexSubImage2D**, and **TexSubImage3D**.

In addition to storing pixel data in client memory, pixel data may also be stored in buffer objects (described in section 2.9). The current pixel unpack and pack buffer objects are designated by the `PIXEL_UNPACK_BUFFER` and `PIXEL_PACK_BUFFER` targets respectively.

Initially, zero is bound for the `PIXEL_UNPACK_BUFFER`, indicating that image specification commands such as **TexImage*D** source their pixels from client memory pointer parameters. However, if a non-zero buffer object is bound as the current pixel unpack buffer, then the pointer parameter is treated as an offset into the designated buffer object.

3.7.2 Transfer of Pixel Rectangles

The process of transferring pixels encoded in buffer object or client memory is diagrammed in figure 3.4. We describe the stages of this process in the order in which they occur.

Commands accepting or returning pixel rectangles take the following arguments (as well as additional arguments specific to their function):

format is a symbolic constant indicating what the values in memory represent.

width and *height* are the width and height, respectively, of the pixel rectangle to be transferred.

data refers to the data to be drawn. These data are represented with one of several GL data types, specified by *type*. The correspondence between the *type* token values and the GL data types they indicate is given in table 3.2.

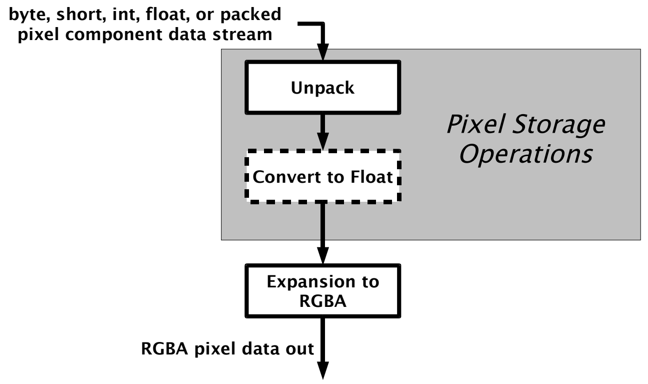


Figure 3.4. Transfer of pixel rectangles to the GL. Output is RGBA pixels. Depth and stencil pixel paths are not shown.

Not all combinations of *format* and *type* are valid. If *format* is `DEPTH_STENCIL` and *type* is not `UNSIGNED_INT_24_8` or `FLOAT_32_UNSIGNED_INT_24_8_REV`, then the error `INVALID_ENUM` occurs. If *format* is one of the integer component formats as defined in table 3.3 and *type* is `FLOAT`, the error `INVALID_ENUM` occurs. Some additional constraints on the combinations of *format* and *type* values that are accepted are discussed below. Additional restrictions may be imposed by specific commands.

Unpacking

Data are taken from the currently bound pixel unpack buffer or client memory as a sequence of signed or unsigned bytes (GL data types `byte` and `ubyte`), signed or unsigned short integers (GL data types `short` and `ushort`), signed or unsigned integers (GL data types `int` and `uint`), or floating point values (GL data types `half` and `float`). These elements are grouped into sets of one, two, three, or four values, depending on the *format*, to form a group. Table 3.3 summarizes the format of groups obtained from memory; it also indicates those formats that yield indices and those that yield floating-point or integer components.

If a pixel unpack buffer is bound (as indicated by a non-zero value of `PIXEL_UNPACK_BUFFER_BINDING`), *data* is an offset into the pixel unpack buffer and the pixels are unpacked from the buffer relative to this offset; otherwise, *data* is a pointer to client memory and the pixels are unpacked from client memory relative to the pointer. If a pixel unpack buffer object is bound and unpacking the pixel data according to the process described below would access memory beyond the size of the pixel unpack buffer's memory size, an `INVALID_OPERATION` error results. If a pixel unpack buffer object is bound and *data* is not evenly divisible by the number of basic machine units needed to store in memory the corresponding GL data type from table 3.2 for the *type* parameter, an `INVALID_OPERATION` error results.

By default the values of each GL data type are interpreted as they would be specified in the language of the client's GL binding. If `UNPACK_SWAP_BYTES` is enabled, however, then the values are interpreted with the bit orderings modified as per table 3.4. The modified bit orderings are defined only if the GL data type `ubyte` has eight bits, and then for each specific GL data type only if that type is represented with 8, 16, or 32 bits.

The groups in memory are treated as being arranged in a rectangle. This rectangle consists of a series of *rows*, with the first element of the first group of the first row pointed to by *data*. If the value of `UNPACK_ROW_LENGTH` is not positive, then the number of groups in a row is *width*; otherwise the number of groups is `UNPACK_ROW_LENGTH`. If p indicates the location in memory of the first element

type Parameter Token Name	Corresponding GL Data Type	Special Interpretation
`UNSIGNED_BYTE`	`ubyte`	No
`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 3.2: Pixel data *type* parameter values and the corresponding GL data types. Refer to table 2.2 for definitions of GL data types. Special interpretations are described near the end of section 3.5.

Format Name	Element Meaning and Order	Target Buffer
`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
`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
`RED_INTEGER`	iR	Color
`GREEN_INTEGER`	iG	Color
`BLUE_INTEGER`	iB	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 3.3: Pixel data formats. The second column gives a description of and the number and order of elements in a group. Unless specified as an index, formats yield components. Components are floating-point unless prefixed with the letter 'i', which indicates they are integer.

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 3.4: Bit ordering modification of elements when `UNPACK_SWAP_BYTES` is enabled. These reorderings are defined only when GL data type `ubyte` has 8 bits, and then only for GL data types with 8, 16, or 32 bits. Bit 0 is the least significant.

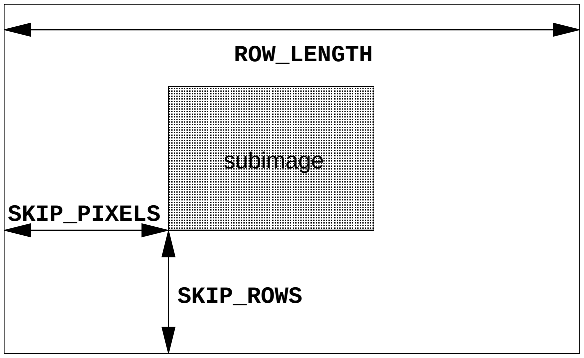


Figure 3.5. Selecting a subimage from an image. The indicated parameter names are prefixed by `UNPACK_` for **TexImage*** and by `PACK_` for **ReadPixels**.

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: `UNPACK_ROW_LENGTH`, `UNPACK_SKIP_ROWS`, and `UNPACK_SKIP_PIXELS`. Before obtaining the first group from memory, the *data* pointer is advanced by $(\text{UNPACK\_SKIP\_PIXELS})n + (\text{UNPACK\_SKIP\_ROWS})k$ elements. Then *width* groups are obtained from contiguous elements in memory (without advancing the pointer), after which the pointer is advanced by k elements. *height* sets of *width* groups of values are obtained this way. See figure 3.5.

Special Interpretations

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`

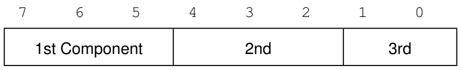
Table 3.5: Packed pixel formats.

A *type* matching one of the types in table 3.5 is a special case in which all the components of each group are packed into a single unsigned byte, unsigned short, or unsigned int, depending on the type. If *type* is `FLOAT_32_UNSIGNED_INT_24_8_REV`, the components of each group are contained within two 32-bit words; the first word contains the float component, and the second word contains a packed 24-bit unused field, followed by an 8-bit component. The number of components per packed pixel is fixed by the type, and must match the number of components per group indicated by the *format* parameter, as listed in table 3.5. The error `INVALID_OPERATION` is generated by any command processing pixel rectangles if a mismatch occurs.

Bitfield locations of the first, second, third, and fourth components of each packed pixel type are illustrated in tables 3.6- 3.9. Each bitfield is interpreted as an unsigned integer value. If the base GL type is supported with more than the minimum precision (e.g. a 9-bit byte) the packed components are right-justified in the pixel.

Components are normally packed with the first component in the most significant bits of the bitfield, and successive component occupying progressively less significant locations. Types whose token names end with `_REV` reverse the component packing order from least to most significant locations. In all cases, the most significant bit of each component is packed in the most significant bit location of its location in the bitfield.

`UNSIGNED_BYTE_3_3_2`:



`UNSIGNED_BYTE_2_3_3_REV`:

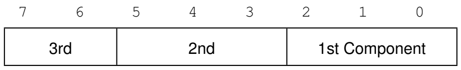


Table 3.6: `UNSIGNED_BYTE` formats. Bit numbers are indicated for each component.

`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>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>4th</td><td colspan="5">3rd</td><td colspan="5">2nd</td><td colspan="5">1st Component</td></tr>
</table>

Table 3.7: `UNSIGNED_SHORT` formats

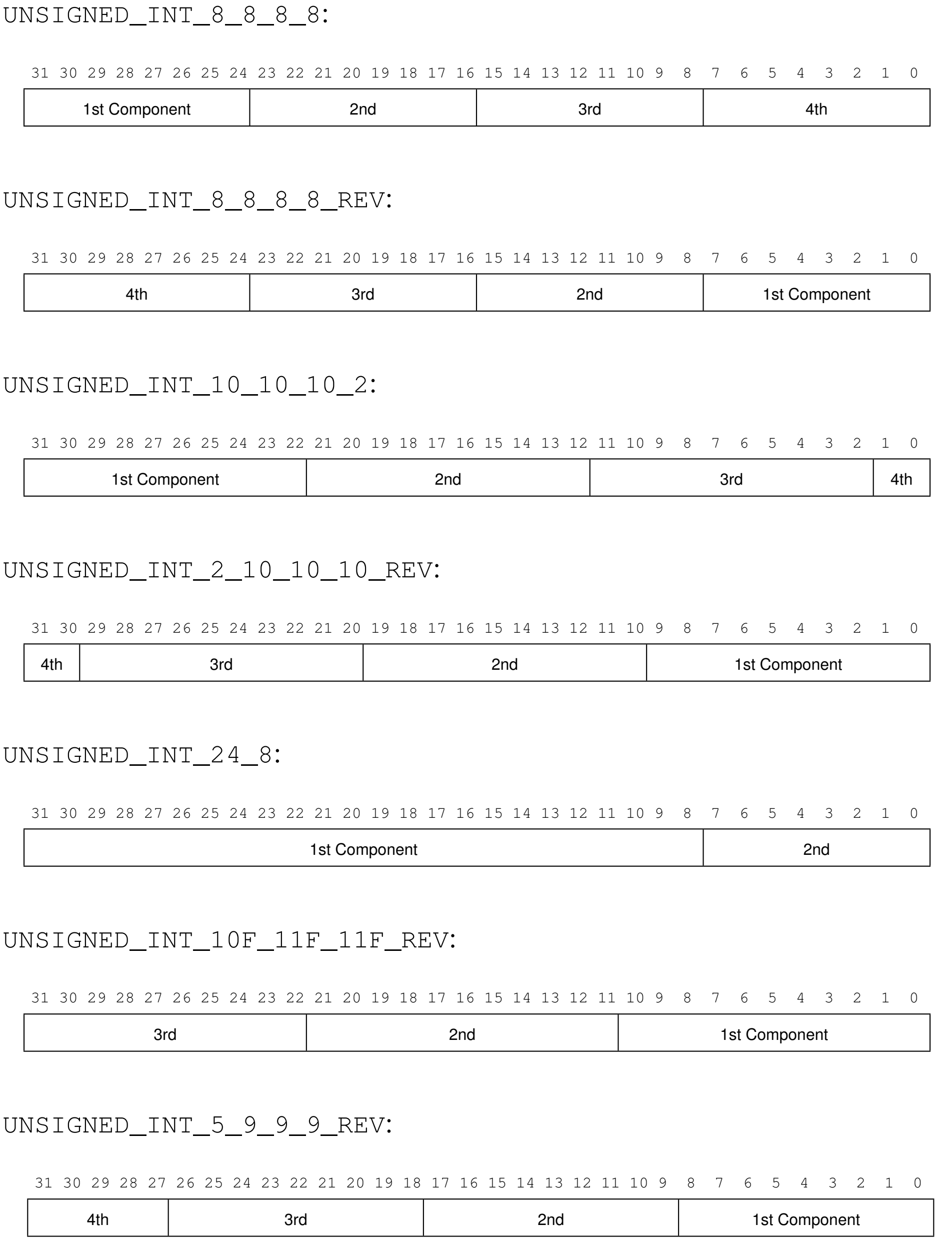


Table 3.8: `UNSIGNED_INT` formats

`FLOAT_32_UNSIGNED_INT_24_8_REV`:

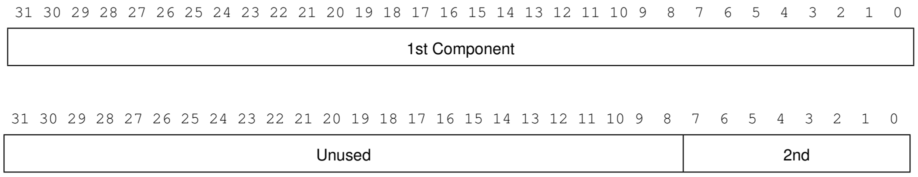


Table 3.9: `FLOAT_UNSIGNED_INT` formats

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 3.10: Packed pixel field assignments.

The assignment of component to fields in the packed pixel is as described in table 3.10.

Byte swapping, if enabled, is performed before the components are extracted from each pixel. The above discussions of row length and image extraction are valid for packed pixels, if “group” is substituted for “component” and the number of components per group is understood to be one.

A *type* of `UNSIGNED_INT_10F_11F_11F_REV` and *format* of `RGB` is a special case in which the data are a series of GL `uint` values. Each uint value specifies 3 packed components as shown in table 3.8. The 1st, 2nd, and 3rd components are called f_{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 3.8. 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:

$$red = p_{red} 2^{p_{exp}-B-N}$$
$$green = p_{green} 2^{p_{exp}-B-N}$$
$$blue = p_{blue} 2^{p_{exp}-B-N}$$

where $B = 15$ (the exponent bias) and $N = 9$ (the number of mantissa bits).

Conversion to floating-point

This step applies only to groups of floating-point components. It is not performed on indices or integer components. For groups containing both components and indices, such as `DEPTH_STENCIL`, the indices are not converted.

Each element in a group is converted to a floating-point value. For unsigned integer elements, equation 2.1 is used. For signed integer elements, equation 2.2 is used unless the final destination of the transferred element is a texture or framebuffer component in one of the `SNORM` formats described in table 3.12, in which case equation 2.3 is used instead.

Final Expansion to RGBA

This step is performed only for non-depth component groups. Each group is converted to a group of 4 elements as follows: if a group does not contain an A element, then A is added and set to 1 for integer components or 1.0 for floating-point components. If any of R, G, or B is missing from the group, each missing element is added and assigned a value of 0 for integer components or 0.0 for floating-point components.

3.8 Texturing

Texturing maps a portion of one or more specified images onto a fragment or vertex. This mapping is accomplished in shaders by *sampling* the color of an image at the location indicated by specified (s, t, r) *texture coordinates*. Texture lookups are typically used to modify a fragment's RGBA color but may be used for any purpose in a shader.

The internal data type of a texture may be signed or unsigned normalized fixed-point, signed or unsigned integer, or floating-point, depending on the internal format of the texture. The correspondence between the internal format and the internal data type is given in tables 3.12-3.13. Fixed-point and floating-point textures return a floating-point value and integer textures return signed or unsigned integer values. The fragment shader is responsible for interpreting the result of a texture lookup as the correct data type, otherwise the result is undefined.

Eight types of texture are supported; each is a collection of images built from one-, two-, or three-dimensional array of image elements referred to as texels. One-, two-, and three-dimensional textures consist respectively of one-, two-, or three-dimensional texel arrays. One- and two-dimensional array textures are arrays of one- or two-dimensional images, consisting of one or more layers. Cube maps are special two-dimensional array textures with six layers that represent the faces of a cube. When accessing a cube map, the texture coordinates are projected onto one of the six faces of the cube. Rectangular textures are special two-dimensional textures consisting of only a single image and accessed using unnormalized coordinates. Buffer textures are special one-dimensional textures whose texel arrays are stored in separate buffer objects.

Implementations must support texturing using multiple images. The following subsections (up to and including section 3.8.8) specify the GL operation with a single texture. The process by which multiple texture images may be sampled and combined by the application-supplied vertex and fragment shaders is described in sections 2.11 and 3.9.

The coordinates used for texturing in a fragment shader are defined by the OpenGL Shading Language Specification.

The command

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

specifies the active texture unit selector, `ACTIVE_TEXTURE`. Each texture image unit consists of all the texture state defined in section 3.8.

The active texture unit selector selects the texture image unit accessed by commands involving texture image processing. Such commands include **TexParameter**, **TexImage**, **BindTexture**, and queries of all such state. If the texture image unit number corresponding to the current value of `ACTIVE_TEXTURE` is greater than or equal to the implementation-dependent constant `MAX_COMBINED_TEXTURE_IMAGE_UNITS`, the error `INVALID_OPERATION` is generated by any such command.

ActiveTexture generates the error `INVALID_ENUM` if an invalid *texture* is specified. *texture* is a symbolic constant of the form TEXTUREi, indicating that texture unit i is to be modified. The constants obey TEXTUREi = TEXTURE0 + i (i is in the range 0 to $k - 1$, where k is the value of `MAX_COMBINED_TEXTURE_IMAGE_UNITS`).

The state required for the active texture image unit selector is a single integer. The initial value is `TEXTURE0`.

3.8.1 Texture Image Specification

The command

`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 3.8.12. *format*, *type*, and *data* specify the format of the image data, the type of those data, and a reference to the image data in the currently bound pixel unpack buffer or client memory, as described in section 3.7.2. The *format* `STENCIL_INDEX` is not allowed.

The groups in memory are treated as being arranged in a sequence of adjacent rectangles. Each rectangle is a two-dimensional image, whose size and organization are specified by the *width* and *height* parameters to **TexImage3D**. The values of `UNPACK_ROW_LENGTH` and `UNPACK_ALIGNMENT` control the row-to-row spacing in these images as described in section 3.7.2. If the value of the integer parameter `UNPACK_IMAGE_HEIGHT` is not positive, then the number of rows in each two-dimensional image is *height*; otherwise the number of rows is `UNPACK_IMAGE_HEIGHT`. Each two-dimensional image comprises an integral number of rows, and is exactly adjacent to its neighbor images.

The mechanism for selecting a sub-volume of a three-dimensional image relies on the integer parameter `UNPACK_SKIP_IMAGES`. If `UNPACK_SKIP_IMAGES` is positive, the pointer is advanced by `UNPACK_SKIP_IMAGES` times the number of elements in one two-dimensional image before obtaining the first group from memory. Then *depth* two-dimensional images are processed, each having a subimage extracted as described in section 3.7.2.

The selected groups are transferred to the GL as described in section 3.7.2 and then clamped to the representable range of the internal format. If the *internalformat* of the texture is signed or unsigned integer, components are clamped to $[-2^{n-1}, 2^{n-1} - 1]$ or $[0, 2^n - 1]$, respectively, where n is the number of bits per component. For color component groups, if the *internalformat* of the texture is signed or unsigned normalized fixed-point, components are clamped to $[-1, 1]$ or $[0, 1]$, respectively. For depth component groups, the depth value is clamped to $[0, 1]$. Otherwise, values are not modified. Stencil index values are masked by $2^n - 1$, where n is the number of stencil bits in the internal format resolution (see below). If the base internal format is `DEPTH_STENCIL` and *format* is not `DEPTH_STENCIL`, then the values of the stencil index texture components are undefined.

Components are then selected from the resulting R, G, B, A, depth, or stencil values to obtain a texture with the *base internal format* specified by (or derived from) *internalformat*. Table 3.11 summarizes the mapping of R, G, B, A, depth, or stencil values to texture components, as a function of the base internal format of the texture image. *internalformat* may be specified as one of the internal format symbolic constants listed in table 3.11, as one of the *sized internal format* symbolic

Base Internal Format	RGBA, Depth, and Stencil Values	Internal Components
`DEPTH_COMPONENT`	Depth	D
`DEPTH_STENCIL`	Depth,Stencil	D,S
`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 3.11: Conversion from RGBA, depth, and stencil pixel components to internal texture components. Texture components R, G, B, and A are converted back to RGBA colors during filtering as shown in table 3.20.

constants listed in tables 3.12- 3.13, as one of the generic compressed internal format symbolic constants listed in table 3.14, or as one of the specific compressed internal format symbolic constants (if listed in table 3.14). Specifying a value for *internalformat* that is not one of the above values generates the error `INVALID_VALUE`.

Textures with a base internal format of `DEPTH_COMPONENT` or `DEPTH_STENCIL` are supported by texture image specification commands only if *target* is `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`, `TEXTURE_RECTANGLE`, `TEXTURE_CUBE_MAP`, `PROXY_TEXTURE_1D`, `PROXY_TEXTURE_2D`, `PROXY_TEXTURE_1D_ARRAY`, `PROXY_TEXTURE_2D_ARRAY`, `PROXY_TEXTURE_RECTANGLE`, or `PROXY_TEXTURE_CUBE_MAP`. Using these formats in conjunction with any other *target* will result in an `INVALID_OPERATION` error.

Textures with a base internal format of `DEPTH_COMPONENT` or `DEPTH_STENCIL` require either depth component data or depth/stencil component data. Textures with other base internal formats require RGBA component data. The error `INVALID_OPERATION` is generated if one of the base internal format and *format* is `DEPTH_COMPONENT` or `DEPTH_STENCIL`, and the other is neither of these values.

Textures with integer internal formats (see table 3.12) require integer data. The error `INVALID_OPERATION` is generated if the internal format is integer and *format* is not one of the integer formats listed in table 3.3; if the internal format is not integer and *format* is an integer format; or if *format* is an integer format and *type* is `FLOAT`, `HALF_FLOAT`, `UNSIGNED_INT_10F_11F_11F_REV`, or `UNSIGNED_INT_5_9_9_9_REV`.

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

Generic compressed internal formats are never used directly as the internal formats of texture images. If *internalformat* is one of the six generic compressed internal formats, its value is replaced by the symbolic constant for a specific compressed internal format of the GL's choosing with the same base internal format. If no specific compressed format is available, *internalformat* is instead replaced by the corresponding base internal format. If *internalformat* is given as or mapped to a specific compressed internal format, but the GL can not support images compressed in the chosen internal format for any reason (e.g., the compression format might not support 3D textures), *internalformat* is replaced by the corresponding base internal format and the texture image will not be compressed by the GL.

The *internal component resolution* is the number of bits allocated to each value in a texture image. If *internalformat* is specified as a base internal format, the GL stores the resulting texture with internal component resolutions of its own choosing. If a sized internal format is specified, the mapping of the R, G, B, A, depth, and stencil values to texture components is equivalent to the mapping of the corresponding base internal format's components, as specified in table 3.11; the type (unsigned int, float, etc.) is assigned the same type specified by *internalformat*; and the memory allocation per texture component is assigned by the GL to match the allocations listed in tables 3.12- 3.13 as closely as possible. (The definition of closely is left up to the implementation. However, a non-zero number of bits must be allocated for each component whose *desired* allocation in tables 3.12- 3.13 is non-zero, and zero bits must be allocated for all other components).

Required Texture Formats

Implementations are required to support at least one allocation of internal component resolution for each type (unsigned int, float, etc.) for each base internal format.

In addition, implementations are required to support the following sized and compressed internal formats. Requesting one of these sized internal formats for any texture type will allocate at least the internal component sizes, and exactly the component types shown for that format in tables 3.12- 3.13:

- Texture and renderbuffer color formats (see section 4.4.2)).
  - `RGBA32F`, `RGBA32I`, `RGBA32UI`, `RGBA16`, `RGBA16F`, `RGBA16I`, `RGBA16UI`, `RGBA8`, `RGBA8I`, `RGBA8UI`, `SRGB8_ALPHA8`, and `RGB10_A2`.
  - `R11F_G11F_B10F`.
  - `RG32F`, `RG32I`, `RG32UI`, `RG16`, `RG16F`, `RG16I`, `RG16UI`, `RG8`, `RG8I`, and `RG8UI`.
  - `R32F`, `R32I`, `R32UI`, `R16F`, `R16I`, `R16UI`, `R16`, `R8`, `R8I`, and `R8UI`.
- Texture-only color formats:
  - `RGBA16_SNORM` and `RGBA8_SNORM`.
  - `RGB32F`, `RGB32I`, and `RGB32UI`.
  - `RGB16_SNORM`, `RGB16F`, `RGB16I`, `RGB16UI`, and `RGB16`.
  - `RGB8_SNORM`, `RGB8`, `RGB8I`, `RGB8UI`, and `SRGB8`.
  - `RGB9_E5`.
  - `RG16_SNORM`, `RG8_SNORM`, `COMPRESSED_RG_RGTC2` and `COMPRESSED_SIGNED_RG_RGTC2`.
  - `R16_SNORM`, `R8_SNORM`, `COMPRESSED_RED_RGTC1` and `COMPRESSED_SIGNED_RED_RGTC1`.
- Depth formats: `DEPTH_COMPONENT32F`, `DEPTH_COMPONENT24`, and `DEPTH_COMPONENT16`.
- Combined depth+stencil formats: `DEPTH32F_STENCIL8` and `DEPTH24_STENCIL8`.

Encoding of Special Internal Formats

If *internalformat* is `R11F_G11F_B10F`, the red, green, and blue bits are converted to unsigned 11-bit, unsigned 11-bit, and unsigned 10-bit floating-point values as described in sections 2.1.3 and 2.1.4.

If *internalformat* is `RGB9_E5`, the red, green, and blue bits are converted to a shared exponent format according to the following procedure:

Components red, $green$, and $blue$ are first clamped (in the process, mapping NaN to zero) as follows:

$$\begin{aligned} red_c &= max(0, min(sharedexp_{max}, red)) \\ green_c &= max(0, min(sharedexp_{max}, green)) \\ blue_c &= max(0, min(sharedexp_{max}, blue)) \end{aligned}$$

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:

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

$$exp_s = \begin{cases} exp_p, & 0 \le max_s < 2^N \\ exp_p + 1, & max_s = 2^N \end{cases}$$

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".

Sized Internal Format	Base Internal Format	R bits	G bits	B bits	A bits	Shared bits
R8	RED	8				
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
R16	RED	16				
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
RG8	RG	8	8			
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
RG16	RG	16	16			
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
R3_G3_B2	RGB	3	3	2		
RGB4	RGB	4	4	4		
RGB5	RGB	5	5	5		
RGB8	RGB	8	8	8		
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
RGB10	RGB	10	10	10		
RGB12	RGB	12	12	12		
RGB16	RGB	16	16	16		
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
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	
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
RGB10_A2	RGBA	10	10	10	2	
RGBA12	RGBA	12	12	12	12	
RGBA16	RGBA	16	16	16	16	
[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]	[redacted]
SRGB8	RGB	8	8	8		
SRGB8_ALPHA8	RGBA	8	8	8	8	
R16F	RED	f16				
RG16F	RG	f16	f16			
RGB16F	RGB	f16	f16	f16		
RGBA16F	RGBA	f16	f16	f16	f16	
R32F	RED	f32				
RG32F	RG	f32	f32			
RGB32F	RGB	f32	f32	f32		

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
`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 3.12: Correspondence of sized internal color formats to base internal formats, internal data type, and *desired* component resolutions for each sized internal format. The component resolution prefix indicates the internal data type: *f* is floating point, *i* is signed integer, *ui* is unsigned integer, *s* is signed normalized fixed-point, and no prefix is unsigned normalized fixed-point.

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 3.13: Correspondence of sized internal depth and stencil formats to base internal formats, internal data type, and *desired* component resolutions for each sized internal format. The component resolution prefix indicates the internal data type: *f* is floating point, *i* is signed integer, *ui* is unsigned integer, and no prefix is fixed-point.

If a compressed internal format is specified, the mapping of the R, G, B, and A values to texture components is equivalent to the mapping of the corresponding base internal format's components, as specified in table 3.11. The specified image is compressed using a (possibly lossy) compression algorithm chosen by the GL.

A GL implementation may vary its allocation of internal component resolution or compressed internal format based on any **TexImage3D**, **TexImage2D** (see below), or **TexImage1D** (see below) parameter (except *target*), but the allocation and chosen compressed image format must not be a function of any other state and cannot be changed once they are established. In addition, the choice of a compressed image format may not be affected by the *data* parameter. Allocations must be invariant; the same allocation and compressed image format must be chosen each time a texture image is specified with the same parameter values. These allocation rules also apply to proxy textures, which are described in section 3.8.12.

The image itself (referred to by *data*) is a sequence of groups of values. The first group is the lower left back corner of the texture image. Subsequent groups fill out rows of width *width* from left to right; *height* rows are stacked from bottom to top forming a single two-dimensional image slice; and *depth* slices are stacked from back to front. When the final R, G, B, and A components have been computed for a group, they are assigned to components of a *texel* as described by table 3.11. Counting from zero, each resulting Nth texel is assigned internal integer coordinates (i, j, k), where

$$i = (N \bmod width) - w_b$$

Compressed Internal Format	Base Internal Format	Type
`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_RED_RGTC1`	`RED`	Specific
`COMPRESSED_SIGNED_RED_RGTC1`	`RED`	Specific
`COMPRESSED_RG_RGTC2`	`RG`	Specific
`COMPRESSED_SIGNED_RG_RGTC2`	`RG`	Specific

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

$$j = (\lfloor \frac{N}{width} \rfloor \bmod height) - h_b$$
$$k = (\lfloor \frac{N}{width \times height} \rfloor \bmod depth) - d_b$$

and w_b, h_b, and d_b are the specified border width, height, and depth. w_b and h_b are the specified *border* value; d_b is the specified *border* value if *target* is `TEXTURE_3D`, or zero if *target* is `TEXTURE_2D_ARRAY`. Thus the last two-dimensional image slice of the three-dimensional image is indexed with the highest value of k.

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

$$\begin{aligned} w_s &= w_t + 2w_b \\ h_s &= h_t + 2h_b \\ d_s &= d_t + 2d_b \end{aligned} \quad (3.16)$$

where w_s, h_s, and d_s are the specified image *width*, *height*, and *depth*, and w_t, h_t, and d_t are the dimensions of the texture image internal to the border. If w_t, h_t, or d_t are less than zero, then the error `INVALID_VALUE` is generated.

The maximum border width b_t is 0. If *border* is less than zero, or greater than b_t, then the error `INVALID_VALUE` is generated.

The maximum allowable width, height, or depth of a texel array for a three-dimensional texture is an implementation-dependent function of the level-of-detail and internal format of the resulting image array. It must be at least $2^{k-lod} + 2b_t$ 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, or one- or two-dimensional array texture, and the maximum allowable height of a two-dimensional or two-dimensional array texture, must be at least $2^{k-lod} + 2b_t$ for image arrays of level 0 through k, where k is the log base 2 of `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 3.8.11.

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 3.8.12. The other parameters match the corresponding parameters of **TexImage3D**.

For the purposes of decoding the texture image, **TexImage2D** is equivalent to calling **TexImage3D** with corresponding arguments and *depth* of 1, except that `UNPACK_SKIP_IMAGES` is ignored.

A two-dimensional or rectangle texture consists of a single two-dimensional texture image. A cube map texture is a set of six two-dimensional texture images. The six cube map texture targets form a single cube map texture though each target names a distinct face of the cube map. The `TEXTURE_CUBE_MAP_*` targets listed above update their appropriate cube map face 2D texture image. Note that the six cube map two-dimensional image tokens such as `TEXTURE_CUBE_MAP_POSITIVE_X` are used when specifying, updating, or querying one of a cube map's six two-dimensional images, but when binding to a cube map texture object (that is when the cube map is accessed as a whole as opposed to a particular two-dimensional image), the `TEXTURE_CUBE_MAP` target is specified.

When the *target* parameter to **TexImage2D** is one of the six cube map two-dimensional image targets, the error `INVALID_VALUE` is generated if the *width* and *height* parameters are not equal.

When *target* is `TEXTURE_RECTANGLE`, an `INVALID_VALUE` error is generated if *level* is non-zero.

An `INVALID_VALUE` error is generated if *border* is non-zero.

Finally, the command

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 3.8.12.

For the purposes of decoding the texture image, **TexImage1D** is equivalent to calling **TexImage2D** with corresponding arguments and *height* of 1.

The image indicated to the GL by the image pointer is decoded and copied into the GL's internal memory.

We shall refer to the decoded image as the *texel array*. A three-dimensional texel array has width, height, and depth w_s, h_s, and d_s as defined in equation 3.16. A two-dimensional or rectangular texel array has depth $d_s = 1$, with height h_s and width w_s as above. A one-dimensional texel array has depth $d_s = 1$, height $h_s = 1$, and width w_s as above.

An element (i, j, k) of the texel array is called a *texel* (for a two-dimensional texture or one-dimensional array texture, k is irrelevant; for a one-dimensional texture, j and k are both irrelevant). The *texture value* used in texturing a fragment is determined by sampling the texture in a shader, but may not correspond to any actual texel. See figure 3.6.

If the *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.

3.8.2 Alternate Texture Image Specification Commands

Two-dimensional and one-dimensional texture images may also be specified using image data taken directly from the framebuffer, and rectangular subregions of existing texture images may be respecified.

The command

```
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

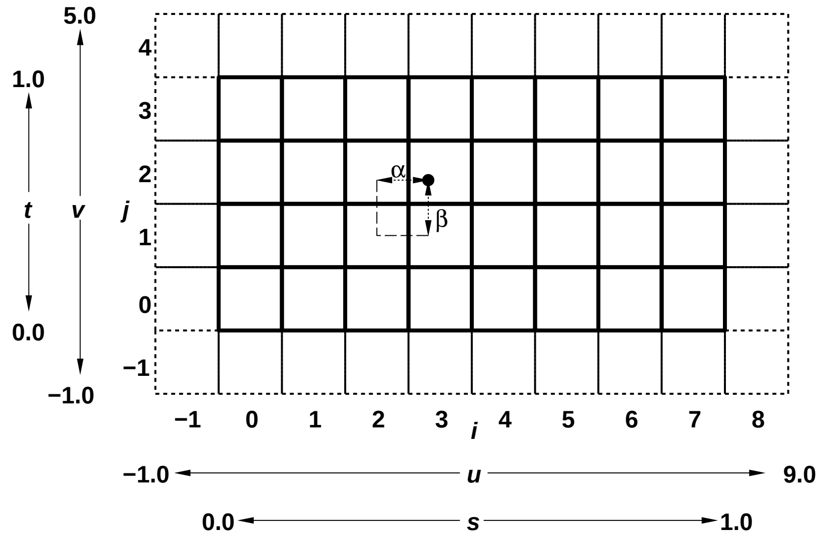


Figure 3.6. A texture image and the coordinates used to access it. This is a two-dimensional texture with $n = 3$ and $m = 2$. A one-dimensional texture would consist of a single horizontal strip. α and β, values used in blending adjacent texels to obtain a texture value, are also shown.

memory. Currently, *target* must be one of `TEXTURE_2D`, `TEXTURE_1D_ARRAY`, `TEXTURE_RECTANGLE`, `TEXTURE_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, or `TEXTURE_CUBE_MAP_NEGATIVE_Z`. *x*, *y*, *width*, and *height* correspond precisely to the corresponding arguments to **ReadPixels** (refer to section 4.3.1); they specify the image's *width* and *height*, and the lower left (x, y) coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to **ReadPixels** with argument *type* set to `COLOR`, `DEPTH`, or `DEPTH_STENCIL`, depending on *internalformat*, stopping after conversion of depth values. RGBA data is taken from the current color buffer, while depth component and stencil index data are taken from the depth and stencil buffers, respectively. The error `INVALID_OPERATION` is generated if depth component data is required and no depth buffer is present; if stencil index data is required and no stencil buffer is present; if integer RGBA data is required and the format of the current color buffer is not integer; or if floating- or fixed-point RGBA data is required and the format of the current color buffer is integer.

Subsequent processing is identical to that described for **TexImage2D**, beginning with clamping of the R, G, B, A, or depth values, and masking of the stencil index values from the resulting pixel groups. Parameters *level*, *internalformat*, and *border* are specified using the same values, with the same meanings, as the equivalent arguments of **TexImage2D**. An invalid value specified for *internalformat* generates the error `INVALID_ENUM`. The constraints on *width*, *height*, and *border* are exactly those for the equivalent arguments of **TexImage2D**.

When the *target* parameter to **CopyTexImage2D** is one of the six cube map two-dimensional image targets, the error `INVALID_VALUE` is generated if the *width* and *height* parameters are not equal.

An `INVALID_FRAMEBUFFER_OPERATION` error will be generated if the object bound to `READ_FRAMEBUFFER_BINDING` (see section 4.4) is not framebuffer complete (as defined in section 4.4.4). An `INVALID_OPERATION` error will be generated if the object bound to `READ_FRAMEBUFFER_BINDING` is framebuffer complete and the value of `SAMPLE_BUFFERS` is greater than zero.

The command

```
void CopyTexImage1D( enum target, int level,
    enum internalformat, int x, int y, sizei width,
    int border );
```

defines a one-dimensional texel array in exactly the manner of **TexImage1D**, except that the image data are taken from the framebuffer, rather than from client memory. Currently, *target* must be `TEXTURE_1D`. For the purposes of decoding the texture image, **CopyTexImage1D** is equivalent to calling **CopyTexImage2D** with corresponding arguments and *height* of 1, except that the *height* of the image is always 1, regardless of the value of *border*. *level*, *internalformat*, and *border* are specified using the same values, with the same meanings, as the equivalent arguments of **TexImage1D**. The constraints on *width* and *border* are exactly those of the equivalent arguments of **TexImage1D**.

Six additional commands,

```
void TexSubImage3D( enum target, int level, int xoffset,
    int yoffset, int zoffset, sizei width, sizei height,
    sizei depth, enum format, enum type, 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**[1]. **CopyTexSubImage1D** arguments *x*, *y*, and *width* match the corresponding arguments to **CopyTexImage1D**. Each of the **TexSubImage** commands interprets and processes pixel groups in exactly the manner of its **TexImage** counterpart, except that the assignment of R, G, B, A, depth, and stencil index pixel group values to the texture components is controlled by the *internalformat* of the texel array, not by an argument to the command. The same constraints and errors apply to the **TexSubImage** commands' argument *format* and the *internalformat* of the texel array being 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 3.6. Taking w_s, h_s, d_s, w_b, h_b, and d_b to be the specified width, height, depth, and border width, border height, and border depth of the texel array, and taking x, y, z, w, h, and d to be the *xoffset*, *yoffset*, *zoffset*, *width*, *height*, and *depth* argument values, any of the following relationships generates the error `INVALID_VALUE`:

$$x < -w_b$$

$$x + w > w_s - w_b$$

$$y < -h_b$$

$$y + h > h_s - h_b$$

$$z < -d_b$$

---

[1] Because the framebuffer is inherently two-dimensional, there is no **CopyTexImage3D** command.

$$z + d > d_s - d_b$$

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 3.6. Taking w_s, h_s, and b_s to be the specified width, height, and border width of the texel array, and taking x, y, w, and h to be the *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 3.14, 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 4.3.1) of the read framebuffer. This situation is discussed in more detail in the description of feedback loops in section 4.4.2.

3.8.3 Compressed Texture Images

Texture images may also be specified or modified using image data already stored in a known compressed image format, such as the RGTC formats defined in appendix C, or additional formats defined by GL extensions.

The commands

`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 modes are ignored when decoding a compressed texture image. If the *imageSize* parameter is not consistent with the format, dimensions, and contents of the compressed image, an `INVALID_VALUE` error results. If the compressed image is not encoded according to the defined image format, the results of the call are undefined.

Specific compressed internal formats may impose format-specific restrictions on the use of the compressed image specification calls or parameters. For example, the compressed image format might be supported only for 2D textures, or might not allow non-zero *border* values. Any such restrictions will be documented in the extension specification defining the compressed internal format; violating these restrictions will result in an `INVALID_OPERATION` error.

Any restrictions imposed by specific compressed internal formats will be invariant, meaning that if the GL accepts and stores a texture image in compressed form, providing the same image to **CompressedTexImage1D**, **CompressedTexImage2D**, or **CompressedTexImage3D** will not result in an `INVALID_OPERATION` error if the following restrictions are satisfied:

- *data* points to a compressed texture image returned by **GetCompressedTexImage** (section 6.1.4).
- *target*, *level*, and *internalformat* match the *target*, *level* and *format* parameters provided to the **GetCompressedTexImage** call returning *data*.
- *width*, *height*, *depth*, *border*, *internalformat*, and *imageSize* match the values of `TEXTURE_WIDTH`, `TEXTURE_HEIGHT`, `TEXTURE_DEPTH`, `TEXTURE_BORDER`, `TEXTURE_INTERNAL_FORMAT`, and `TEXTURE_COMPRESSED_IMAGE_SIZE` for image level *level* in effect at the time of the **GetCompressedTexImage** call returning *data*.

This guarantee applies not just to images returned by **GetCompressedTexImage**, but also to any other properly encoded compressed texture image of the same size and format.

If *internalformat* is one of the specific `RGTC` formats described in table 3.14, the compressed image data is stored using one of the RGTC compressed texture image encodings (see appendix C.1) The RGTC texture compression algorithm supports only two-dimensional images without borders. If *internalformat* is an RGTC format, **CompressedTexImage1D** will generate an `INVALID_ENUM` error; **CompressedTexImage2D** will generate an `INVALID_OPERATION` error if *border* is non-zero or *target* is `TEXTURE_RECTANGLE`; and **CompressedTexImage3D** will generate an `INVALID_OPERATION` error if *border* is non-zero or *target* is not `TEXTURE_2D_ARRAY`.

If the *data* argument of **CompressedTexImage1D**, **CompressedTexImage2D**, or **CompressedTexImage3D** is a null pointer (a zero-valued pointer in the C implementation), and the pixel unpack buffer object is zero, a texel array with unspecified image contents is created, just as when a null pointer is passed to **TexImage1D**, **TexImage2D**, or **TexImage3D**.

The commands

```
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 3.14, the texture is stored using one of the RGTC compressed texture image encodings (see appendix C.1). If *internalformat* is an RGTC format, **CompressedTex-**

SubImage1D 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.

3.8.4 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. Also unlike most other texture types, 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 3.15.

When a buffer object is attached to a buffer texture, the buffer object's data store is taken as the texture's texel array. The number of texels in the buffer texture's texel array is given by

$$\left\lfloor \frac{buffer\_size}{components \times sizeof(base\_type)} \right\rfloor.$$

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 3.15. The number of texels in the texel array is then clamped to the implementation-dependent limit `MAX_TEXTURE_BUFFER_SIZE`. When a buffer texture is accessed in a shader, the results of a texel fetch are undefined if the specified texel coordinate is negative, or greater than or equal to the clamped number of texels in the texel array.

When a buffer texture is accessed in a shader, an integer is provided to indicate the texel coordinate being accessed. If no buffer object is bound to the buffer texture, the results of the texel access are undefined. Otherwise, the attached buffer object's data store is interpreted as an array of elements of the GL data type corresponding to *internalformat*. Each texel consists of one to four elements that are mapped to texture components (R, G, B, and A). Element m of the texel numbered n is taken from element $n \times components + m$ of the attached buffer object's data store. Elements and texels are both numbered starting with zero. For texture formats with signed or unsigned normalized fixed-point components, the extracted values are converted to floating-point using equations 2.1 or 2.3, respectively. The components of the texture are then converted to an (R,G,B,A) vector according to table 3.15, and returned to the shader as a four-component result vector with components of the appropriate data type for the texture's internal format. The base data type, component count, normalized component information, and mapping of data store elements to texture components is specified in table 3.15.

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.

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 3.15: Internal formats for buffer textures. For each format, the data type of each element is indicated in the “Base Type” column and the element count is in the “Components” column. The “Norm” column indicates whether components should be treated as normalized floating-point values. The “Component 0, 1, 2, and 3” columns indicate the mapping of each element of a texel to texture components.

3.8.5 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*);

taget is the target, either `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`. `TEXTURE_RECTANGLE`, or `TEXTURE_CUBE_MAP`. *params* is a symbolic constant indicating the parameter to be set; the possible constants and corresponding parameters are summarized in table 3.16. In the first form of the command, *param* is a value to which to set a single-valued parameter; in the remaining forms, *params* is an array of parameters whose type depends on the parameter being set.

If the values for `TEXTURE_BORDER_COLOR` are specified with **TexParameterIiv** or **TexParameterIuiv**, the values are unmodified and stored with an internal data type of integer. If specified with **TexParameteriv**, they are converted to floating-point using equation 2.2. Otherwise the values are unmodified and stored as floating-point.

In the remainder of section 3.8, denote by 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.

Texture parameters for a cube map texture apply to the cube map as a whole; the six distinct two-dimensional texture images use the texture parameters of the cube map itself.

When *target* is `TEXTURE_RECTANGLE`, certain texture parameter values may not be specified. In this case, the error `INVALID_ENUM` is generated if the `TEXTURE_WRAP_S`, `TEXTURE_WRAP_T`, or `TEXTURE_WRAP_R` parameter is set 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.

3.8.6 Depth Component Textures

Depth textures and the depth components of depth/stencil textures can be treated as `RED` textures during texture filtering and application (see section 3.8.14). The

Name	Type	Legal Values
`TEXTURE_WRAP_S`	enum	`CLAMP_TO_EDGE`, `REPEAT`, `CLAMP_TO_BORDER`, `MIRRORED_REPEAT`
`TEXTURE_WRAP_T`	enum	`CLAMP_TO_EDGE`, `REPEAT`, `CLAMP_TO_BORDER`, `MIRRORED_REPEAT`
`TEXTURE_WRAP_R`	enum	`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_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
`TEXTURE_COMPARE_MODE`	enum	`NONE`, `COMPARE_REF_TO_TEXTURE`

`TEXTURE_COMPARE_FUNC`	enum	`LEQUAL`, `GEQUAL`
		`LESS`, `GREATER`,
		`EQUAL`, `NOTEQUAL`,
		`ALWAYS`, `NEVER`

Table 3.16: Texture parameters and their values.

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

initial state for depth and depth/stencil textures treats them as `RED` textures.

3.8.7 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 3.17 explains how the major axis direction maps to the two-dimensional image of a particular cube map target.

Using the s_c, t_c, and m_a determined by the major axis direction as specified in table 3.17, an updated $\begin{pmatrix} s & t \end{pmatrix}$ is calculated as follows:

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

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

This new $\begin{pmatrix} s & t \end{pmatrix}$ is used to find a texture value in the determined face's two-dimensional texture image using the rules given in sections 3.8.8 through 3.8.9.

3.8.8 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

$$\lambda_{base}(x,y) = \log_2[\rho(x,y)] \tag{3.17}$$

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

$$\lambda = \begin{cases} lod_{max}, & \lambda' > lod_{max} \\ \lambda', & lod_{min} \leq \lambda' \leq lod_{max} \\ lod_{min}, & \lambda' < lod_{min} \\ undefined, & lod_{min} > lod_{max} \end{cases} \tag{3.19}$$

$bias_{texobj}$ is the value of TEXTURE_LOD_BIAS for the bound texture object (as described in section 3.8.5). $bias_{shader}$ 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 3.8.9) the texture is said to be magnified; if it is greater, the texture is minified. Sampling of minified textures is described in the remainder of this section, while sampling of magnified textures is described in section 3.8.9.

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

$$\begin{aligned} u(x,y) &= \begin{cases} w_t + \delta_u, & \text{rectangular texture} \\ w_t \times s(x,y) + \delta_u, & \text{otherwise} \end{cases} \\ v(x,y) &= \begin{cases} h_t + \delta_u, & \text{rectangular texture} \\ h_t \times t(x,y) + \delta_v, & \text{otherwise} \end{cases} \\ w(x,y) &= d_t \times r(x,y) + \delta_w \end{aligned} \tag{3.20}$$

where w_t, h_t, and d_t are as defined by equation 3.16 with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is $level_{base}$. For a one-dimensional or one-dimensional array texture, define $v(x, y) = 0$ and $w(x, y) = 0$; for a two-dimensional, two-dimensional array, rectangular, or cube map texture, define $w(x, y) = 0$.

$(\delta_u, \delta_v, \delta_w)$ are the texel offsets specified in the OpenGL Shading Language texture lookup functions that support offsets. If the texture function used does not support offsets, all three shader offsets are taken to be zero. If any of the offset values are outside the range of the implementation-defined values `MIN_PROGRAM_TEXEL_OFFSET` and `MAX_PROGRAM_TEXEL_OFFSET`, results of the texture lookup are undefined.

For a polygon or point, ρ is given at a fragment with window coordinates (x, y) by

$$\rho = \max\left\{ \sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial w}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial y}\right)^2} \right\} \tag{3.21}$$

where $\partial u/\partial x$ indicates the derivative of u with respect to window x, and similarly for the other derivatives.

For a line, the formula is

$$\rho = \sqrt{\left(\frac{\partial u}{\partial x}\Delta x + \frac{\partial u}{\partial y}\Delta y\right)^2 + \left(\frac{\partial v}{\partial x}\Delta x + \frac{\partial v}{\partial y}\Delta y\right)^2 + \left(\frac{\partial w}{\partial x}\Delta x + \frac{\partial w}{\partial y}\Delta y\right)^2} \Big/ l, \tag{3.22}$$

where $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$ with (x_1, y_1) and (x_2, y_2) being the segment's window coordinate endpoints and $l = \sqrt{\Delta x^2 + \Delta y^2}$.

While it is generally agreed that equations 3.21 and 3.22 give the best results when texturing, they are often impractical to implement. Therefore, an implementation may approximate the ideal 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 $\max\{m_u, m_v, m_w\} \leq f(x, y) \leq m_u + m_v + m_w$.

Coordinate Wrapping and Texel Selection

After generating $u(x, y)$, $v(x, y)$, and $w(x, y)$, they may be clamped and wrapped before sampling the texture, depending on the corresponding texture wrap modes. Let

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

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

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

where $clamp(a, b, c)$ 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 3.18. For a three-dimensional texture, the texel at location (i, j, k) becomes the texture value. For two-dimensional, two-dimensional array, rectangular, or cube map textures, k is irrelevant, and the texel at location (i, j) becomes the texture value. For one-dimensional texture or one-dimensional array textures, j and k are irrelevant, and the texel at location i becomes the texture value.

For one- and two-dimensional array textures, the texel is obtained from image layer l, where

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

Wrap mode	Result of $wrap(coord)$
`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 3.18: 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.

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

$$
\begin{aligned}
i &< -b_s & i &\geq w_t + b_s \\
j &< -b_s & j &\geq h_t + b_s \\
k &< -b_s & k &\geq d_t + b_s
\end{aligned}
$$

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 3.11. The internal data type of the border values must be consistent with the type returned by the texture as described in section 3.8, or the result is undefined. Border values are clamped before they are used, according to the format in which texture components are stored. For signed and unsigned normalized fixed-point formats, border values are clamped to $[-1, 1]$ and $[0, 1]$, respectively. For floating-point and integer formats, border values are clamped to the representable range of the format. If the texture contains depth components, the first component of `TEXTURE_BORDER_COLOR` is interpreted as a depth value.

When the value of `TEXTURE_MIN_FILTER` is `LINEAR`, a $2 \times 2 \times 2$ cube of texels in the image array of level $level_{base}$ 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

$$\begin{aligned}
\tau = {} & (1-\alpha)(1-\beta)(1-\gamma)\tau_{i_0 j_0 k_0} + \alpha(1-\beta)(1-\gamma)\tau_{i_1 j_0 k_0} \\
& + (1-\alpha)\beta(1-\gamma)\tau_{i_0 j_1 k_0} + \alpha\beta(1-\gamma)\tau_{i_1 j_1 k_0} \\
& + (1-\alpha)(1-\beta)\gamma\tau_{i_0 j_0 k_1} + \alpha(1-\beta)\gamma\tau_{i_1 j_0 k_1} \\
& + (1-\alpha)\beta\gamma\tau_{i_0 j_1 k_1} + \alpha\beta\gamma\tau_{i_1 j_1 k_1}
\end{aligned} \qquad (3.23)$$

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 4.4.2.

- 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 $level_{base}$ 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 3.8.11.

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

$$d = \begin{cases} level_{base}, & \lambda \le \frac{1}{2} \\ \lceil level_{base} + \lambda + \frac{1}{2} \rceil - 1, & \lambda > \frac{1}{2}, level_{base} + \lambda \le q + \frac{1}{2} \\ q, & \lambda > \frac{1}{2}, level_{base} + \lambda > q + \frac{1}{2} \end{cases} \tag{3.24}$$

The rules for `NEAREST` or `LINEAR` filtering are then applied to the selected array. Specifically, the coordinate (u, v, w) is computed as in equation 3.20, with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is d.

For mipmap filters `NEAREST_MIPMAP_LINEAR` and `LINEAR_MIPMAP_LINEAR`, the level d_1 and d_2 mipmap arrays are selected, where

$$d_1 = \begin{cases} q, & level_{base} + \lambda \ge q \\ \lfloor level_{base} + \lambda \rfloor, & \text{otherwise} \end{cases} \tag{3.25}$$

$$d_2 = \begin{cases} q, & level_{base} + \lambda \ge q \\ d_1 + 1, & \text{otherwise} \end{cases} \tag{3.26}$$

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 3.20, with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is d_1. For level d_2 the coordinate (u', v', w') is computed as in equation 3.20, with w_s, h_s, and d_s equal to the width, height, and depth of the image array whose level is d_2.

The final texture value is then found as

$$\tau = [1 - \mathrm{frac}(\lambda)]\tau_1 + \mathrm{frac}(\lambda)\tau_2.$$

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 3.8.11.

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

The internal formats of the derived mipmap arrays all match those of the $level_{base}$ array, and the dimensions of the derived arrays follow the requirements described in section 3.8.11.

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

3.8.9 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 `TEXTURE_MIN_FILTER` as described in section 3.8.8, including the texture coordinate wrap modes specified in table 3.18. The level-of-detail $level_{base}$ texel array is always used for magnification.

Implementations may either unconditionally assume $c = 0$ for the minification vs. magnification switch-over point, or may choose to make c depend on the combination of minification and magnification modes as follows: if the magnification filter is given by `LINEAR` and the minification filter is given by `NEAREST_MIPMAP_NEAREST` or `NEAREST_MIPMAP_LINEAR`, then $c = 0.5$. This is done to ensure that a minified texture does not appear “sharper” than a magnified texture. Otherwise $c = 0$.

3.8.10 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.

3.8.11 Texture Completeness

A texture is said to be complete if all the image arrays and texture parameters required to utilize the texture for texture application are consistently defined. The definition of completeness varies depending on the texture dimensionality.

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

- The set of mipmap arrays $level_{base}$ through q (where q is defined in the **Mipmapping** discussion of section 3.8.8) were each specified with the same internal format.
- The dimensions of the arrays follow the sequence described in the **Mipmapping** discussion of section 3.8.8.
- $level_{base} \leq level_{max}$
- Each dimension of the $level_{base}$ array is positive.
- If the internal format of the arrays is integer (see (see table 3.12), `TEXTURE_MAG_FILTER` must be `NEAREST` and `TEXTURE_MIN_FILTER` must be `NEAREST` or `NEAREST_MIPMAP_NEAREST`.

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

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

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

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

Effects of Completeness on Texture Application

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

Effects of Completeness on Texture Image Specification

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

3.8.12 Texture State and Proxy State

The state necessary for texture can be divided into two categories. First, there are the multiple sets of texel arrays (a single array for the rectangular texture target; one set of mipmap arrays each for the one-, two-, and three-dimensional and one- and two-dimensional array texture targets; and six sets of mipmap arrays for the cube map texture targets) and their number. Each array has associated with it a width, height (two- and three-dimensional, rectangular, one-dimensional array, and cube map only), and depth (three-dimensional and two-dimensional array only), an integer describing the internal format of the image, integer values describing the resolutions of each of the red, green, blue, alpha, depth, and stencil components of the image, integer values describing the type (unsigned normalized, integer, floating-point, etc.) of each of the components, a boolean describing whether the image is compressed or not, and an integer size of a compressed image. Each initial texel array is null (zero width, height, and depth, internal format `RGBA`, component sizes set to zero and component types set to `NONE`, the compressed flag set to `FALSE`, and a zero compressed size). 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 `R8`. Next, there are the four sets of texture properties, corresponding to the one-, two-, three-dimensional, and cube map texture targets. Each set consists of the selected minification and magnification filters, the wrap modes for s, t (two- and three-dimensional and cube map only), and r (three-dimensional only), the `TEXTURE_BORDER_COLOR`, two floating-point numbers describing the minimum and maximum level of detail, two integers describing the base and maximum mipmap array, a boolean flag indicating whether the texture is resident, and three integers describing the depth texture mode, compare mode, and compare function. In the initial state, the value assigned to `TEXTURE_MIN_FILTER` is `NEAREST_MIPMAP_LINEAR`, (except for rectangular textures, where the initial value is `LINEAR`), and the value for `TEXTURE_MAG_FILTER` is `LINEAR`. s, t, and r wrap modes are all set to `REPEAT` (except for rectangular textures, where the initial value is `CLAMP_TO_EDGE`). The values of `TEXTURE_MIN_LOD` and `TEXTURE_MAX_LOD` are -1000 and 1000 respectively. The values of `TEXTURE_BASE_LEVEL` and `TEXTURE_MAX_LEVEL` are 0 and 1000 respectively. The value of `TEXTURE_BORDER_COLOR` is (0,0,0,0). The values of `TEXTURE_COMPARE_MODE`, and `TEXTURE_COMPARE_FUNC` are `NONE`, and `LEQUAL` respectively.

In addition to image arrays for the non-proxy texture targets described above, partially instantiated image arrays are maintained for one-, two-, and three-dimensional, rectangular, and one- and two-dimensional array textures. Additionally, a single proxy image array is maintained for the cube map texture. Each proxy image array includes width, height, depth, and internal format state values, as well as state for the red, green, blue, alpha, depth, and stencil component resolutions and types. Proxy arrays do not include image data nor texture parameters. When **TexImage3D** is executed with *target* specified as `PROXY_TEXTURE_3D`, the three-dimensional proxy state values of the specified level-of-detail are recomputed and updated. If the image array would not be supported by **TexImage3D** called with *target* set to `TEXTURE_3D`, no error is generated, but the proxy width, height, depth, and component resolutions are set to zero, and the component types are set to `NONE`. If the image array would be supported by such a call to **TexImage3D**, the proxy state values are set exactly as though the actual image array were being specified. No pixel data are transferred or processed in either case.

Proxy arrays for one- and two-dimensional textures and one- and two-dimensional array textures are operated on in the same way when **TexImage1D** is executed with *target* specified as `PROXY_TEXTURE_1D`, **TexImage2D** is executed with *target* specified as `PROXY_TEXTURE_2D`, `PROXY_TEXTURE_1D_ARRAY`, or `PROXY_TEXTURE_RECTANGLE`, or **TexImage3D** is executed with *target* specified as `PROXY_TEXTURE_2D_ARRAY`.

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.

3.8.13 Texture Objects

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

A texture object is created by *binding* an unused name to 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 3.8.12, set to the same initial values. The new texture object bound to *target* is, and remains a texture of the dimensionality and type specified by *target* until it is deleted.

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

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

While a texture object is bound, GL operations on the target to which it is bound affect the bound object, and queries of the target to which it is bound return state from the bound object. If texture mapping of the dimensionality of the target 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`, and `TEXTURE_CUBE_MAP` have one-, two-, and three-dimensional, one- and two-dimensional array, rectangular, buffer, and cube map texture state vectors respectively associated with them. In order that access to these initial textures not be lost, they are treated as texture objects all of whose names are 0. The initial one-, two-, three-dimensional, one- and two-dimensional 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`, or `TEXTURE_CUBE_MAP` respectively while 0 is bound to the corresponding targets.

Texture objects are deleted by calling

`void` **DeleteTextures**(`sizei` *n*, `uint` **textures*);

textures contains *n* names of texture objects to be deleted. After a texture object is deleted, it has no contents or dimensionality, and its name is again unused. If a texture that is currently bound to any of the *target* bindings of **BindTexture** is deleted, it is as though **BindTexture** had been executed with the same *target* and *texture* zero. Additionally, special care must be taken when deleting a texture if any of the images of the texture are attached to a framebuffer object. See section 4.4.2 for details.

Unused names in *textures* are silently ignored, as is the value zero.

The command

`void` **GenTextures**(`sizei` *n*, `uint` **textures*);

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

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

3.8.14 Texture Comparison Modes

Texture values can also be computed according to a specified comparison function. Texture parameter `TEXTURE_COMPARE_MODE` specifies the comparison operands, and parameter `TEXTURE_COMPARE_FUNC` specifies the comparison function.

Depth Texture Comparison Mode

If the currently bound texture's base internal format is `DEPTH_COMPONENT` or `DEPTH_STENCIL`, then `TEXTURE_COMPARE_MODE` and `TEXTURE_COMPARE_FUNC` control the output of the texture unit as described below. Otherwise, the texture unit operates in the normal manner and texture comparison is bypassed.

Let D_t be the depth texture value and D_{ref} be the reference value, provided by the shader's texture lookup function.

If the texture's internal format indicates a fixed-point depth texture, then D_t and D_{ref} are clamped to the range $[0, 1]$; otherwise no clamping is performed. Then the effective texture value is computed as follows:

If the value of `TEXTURE_COMPARE_MODE` is `NONE`, then

$$r = D_t$$

If the value of `TEXTURE_COMPARE_MODE` is `COMPARE_REF_TO_TEXTURE`, then r depends on the texture comparison function as shown in table 3.19.

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 3.19: Depth texture comparison functions.

The resulting r is assigned to R_t.

If the value of `TEXTURE_MAG_FILTER` is not `NEAREST`, or the value of `TEXTURE_MIN_FILTER` is not `NEAREST` or `NEAREST_MIPMAP_NEAREST`, then r may be computed by comparing more than one depth texture value to the texture reference value. The details of this are implementation-dependent, but r should be a value in the range $[0, 1]$ which is proportional to the number of comparison passes or failures.

3.8.15 sRGB Texture Color Conversion

If the currently bound texture's internal format is one of `SRGB`, `SRGB8`, `SRGB_ALPHA`, `SRGB8_ALPHA8`, `COMPRESSED_SRGB`, or `COMPRESSED_SRGB_ALPHA`, the red, green, and blue components are converted from an sRGB color space to a linear color space as part of filtering described in sections 3.8.8 and 3.8.9. 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.

$$c_l = \begin{cases} \frac{c_s}{12.92}, & c_s \leq 0.04045 \\ \left(\frac{c_s+0.055}{1.055}\right)^{2.4}, & c_s > 0.04045 \end{cases} \qquad (3.27)$$

Assume c_s is the sRGB component in the range $[0, 1]$.

3.8.16 Shared Exponent Texture Color Conversion

If the currently bound texture's internal format is `RGB9_E5`, the red, green, blue, and shared bits are converted to color components (prior to filtering) using shared exponent decoding. The component red_s, $green_s$, $blue_s$, and exp_{shared} values (see section 3.8.1) are treated as unsigned integers and are converted to red, $green$, and $blue$ as follows:

$$red = red_s 2^{exp_{shared}-B}$$
$$green = green_s 2^{exp_{shared}-B}$$
$$blue = blue_s 2^{exp_{shared}-B}$$

3.9 Fragment Shaders

The sequence of operations that are applied to fragments that result from rasterizing a point, line segment, or polygon are described using a *fragment shader*.

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

Fragment shaders are created as described in section 2.11.1 using a *type* parameter of `FRAGMENT_SHADER`. They are attached to and used in program objects as described in section 2.11.2.

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

3.9.1 Shader Variables

Fragment shaders can access uniforms belonging to the current shader object. The amount of storage available for fragment shader uniform variables in the default uniform block is specified by the value of the implementation-dependent constant `MAX_FRAGMENT_UNIFORM_COMPONENTS`. The total amount of combined storage available for fragment shader uniform variables in all uniform blocks (including the default uniform block) is specified by the value of the implementation-dependent constant `MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS`. These values represent the numbers of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a fragment shader. A uniform matrix will consume no more than $4 \times min(r, c)$ such values, where r and c are the number of rows and columns in the matrix. A link error will be generated if an attempt is made to utilize more than the space available for fragment shader uniform variables.

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

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

Texture Base Internal Format	Texture source color C_s	A_s
`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 3.20: Correspondence of filtered texture components to texture source components.

the same name are defined in the fragment shader.

A fragment shader can also write to varying out variables. Values written to these variables are used in the subsequent per-fragment operations. Varying out variables can be used to write floating-point, integer or unsigned integer values destined for buffers attached to a framebuffer object, or destined for color buffers attached to the default framebuffer. The **Shader Outputs** subsection of section 3.9.2 describes how to direct these values to buffers.

3.9.2 Shader Execution

The executable version of the fragment shader is used to process incoming fragment values that are the result of rasterization.

Texture Access

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

When a texture lookup is performed in a fragment shader, the GL computes the filtered texture value τ in the manner described in sections 3.8.8 and 3.8.9, and converts it to a texture source color C_s according to table 3.20. The GL returns a four-component vector (R_s, G_s, B_s, A_s) to the fragment shader. For the purposes of level-of-detail calculations, the derivatives $\frac{du}{dx}$, $\frac{du}{dy}$, $\frac{dv}{dx}$, $\frac{dv}{dy}$, $\frac{dw}{dx}$ and $\frac{dw}{dy}$ may be approximated by a differencing algorithm as detailed in section 8.8 of the OpenGL Shading Language specification.

Texture lookups involving textures with depth component data can either return the depth data directly or return the results of a comparison with the D_{ref} value (see section 3.8.14) 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`.

If a fragment shader uses a sampler whose associated texture object is not complete, as defined in section 3.8.11, the texture image unit will return (R, G, B, A) = $(0, 0, 0, 1)$.

The number of separate texture units that can be accessed from within a fragment shader during the rendering of a single primitive is specified by the implementation-dependent constant `MAX_TEXTURE_IMAGE_UNITS`.

Shader Inputs

The OpenGL Shading Language specification describes the values that are available as inputs to the fragment shader.

The built-in variable `gl_FragCoord` holds the window coordinates x, y, z, and $\frac{1}{w}$ for the fragment. The z component of `gl_FragCoord` undergoes an implied conversion to floating-point. This conversion must leave the values 0 and 1 invariant. Note that this z component already has a polygon offset added in, if enabled (see section 3.6.4). The $\frac{1}{w}$ value is computed from the w_c coordinate (see section 2.12).

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 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.

The built-in variable `gl_PrimitiveID` is filled with the number of primitives processed by the rasterizer since the last drawing command was called. The first primitive generated by a drawing command is numbered zero, and the primitive ID counter is incremented after every individual point, line, or polygon primitive is processed. For polygons drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may be drawn.

Restarting a primitive using the primitive restart index (see section 2.8) has no effect on the primitive ID counter.

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

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 2.12.1). For floating-point depth buffers, conversion is not performed but clamping is. Note that the depth range computation is not applied here, only the conversion to fixed-point.

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

3.10 Antialiasing Application

If antialiasing is enabled for the primitive from which a rasterized fragment was produced, then the computed coverage value is applied to the fragment. The value is multiplied by the fragment's alpha (A) value to yield a final alpha value. The coverage value is applied separately to each fragment color, and only applied if the corresponding color buffer in the framebuffer has a fixed- or floating-point format.

3.11 Multisample Point Fade

Finally, if multisampling is enabled and the rasterized fragment results from a point primitive, then the computed fade factor from equation 3.2 is applied to the fragment. The fade factor is multiplied by the fragment's alpha value to yield a final alpha value. The fade factor is applied separately to each fragment color, and only applied if the corresponding color buffer in the framebuffer has a fixed- or floating-point format.

Chapter 4

Per-Fragment Operations and the Framebuffer

The framebuffer, whether it is the default framebuffer or a framebuffer object (see section 2.1), consists of a set of pixels arranged as a two-dimensional array. For purposes of this discussion, each pixel in the framebuffer is simply a set of some number of bits. The number of bits per pixel may vary depending on the GL implementation, the type of framebuffer selected, and parameters specified when the framebuffer was created. Creation and management of the default framebuffer is outside the scope of this specification, while creation and management of framebuffer objects is described in detail in section 4.4.

Corresponding bits from each pixel in the framebuffer are grouped together into a *bitplane*; each bitplane contains a single bit from each pixel. These bitplanes are grouped into several *logical buffers*. These are the *color*, *depth*, and *stencil* buffers. The color buffer actually consists of a number of buffers, and these color buffers serve related but slightly different purposes depending on whether the GL is bound to the default framebuffer or a framebuffer object.

For the default framebuffer, the color buffers are the *front left* buffer, the *front right* buffer, the *back left* buffer, and the *back right* buffer. Typically the contents of the front buffers are displayed on a color monitor while the contents of the back buffers are invisible. (Monoscopic contexts display only the front left buffer; stereoscopic contexts display both the front left and the front right buffers.) All color buffers must have the same number of bitplanes, although an implementation or context may choose not to provide right buffers, or back buffers at all. Further, an implementation or context may choose not to provide depth or stencil buffers. If no default framebuffer is associated with the GL context, the framebuffer is incomplete except when a framebuffer object is bound (see sections 4.4.1 and 4.4.4).

Framebuffer objects are not visible, and do not have any of the color buffers present in the default framebuffer. Instead, the buffers of an framebuffer object are specified by attaching individual textures or renderbuffers (see section 4.4) to a set of attachment points. A framebuffer object has an array of color buffer attachment points, numbered zero through n, a depth buffer attachment point, and a stencil buffer attachment point. In order to be used for rendering, a framebuffer object must be *complete*, as described in section 4.4.4. Not all attachments of a framebuffer object need to be populated.

Each pixel in a color buffer consists of up to four color components. The four color components are named R, G, B, and A, in that order; color buffers are not required to have all four color components. R, G, B, and A components may be represented as signed or unsigned normalized fixed-point, floating-point, or signed or unsigned integer values; all components must have the same representation. Each pixel in a depth buffer consists of a single unsigned integer value in the format described in section 2.12.1 or a floating-point value. Each pixel in a stencil buffer consists of a single unsigned integer value.

The number of bitplanes in the color, depth, and stencil buffers is dependent on the currently bound framebuffer. For the default framebuffer, the number of bitplanes is fixed. For framebuffer objects, the number of bitplanes in a given logical buffer may change if the image attached to the corresponding attachment point changes.

The GL has two active framebuffers; the *draw framebuffer* is the destination for rendering operations, and the *read framebuffer* is the source for readback operations. The same framebuffer may be used for both drawing and reading. Section 4.4.1 describes the mechanism for controlling framebuffer usage.

The default framebuffer is initially used as the draw and read framebuffer [1], and the initial state of all provided bitplanes is undefined. The format and encoding of buffers in the draw and read framebuffers can be queried as described in section 6.1.3.

4.1 Per-Fragment Operations

A fragment produced by rasterization with window coordinates of (x_w, y_w) modifies the pixel in the framebuffer at that location based on a number of parameters and conditions. We describe these modifications and tests, diagrammed in figure 4.1, in the order in which they are performed. Figure 4.1 diagrams these

---

[1] The window system binding API may allow associating a GL context with two separate "default framebuffers" provided by the window system as the draw and read framebuffers, but if so, both default framebuffers are referred to by the name zero at their respective binding points.

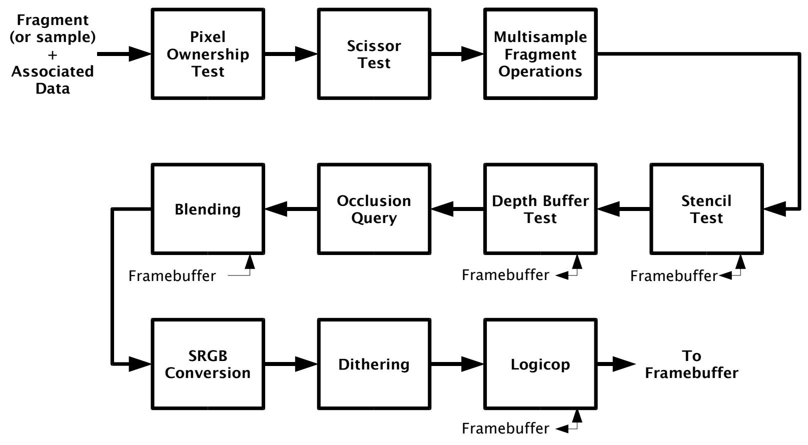


Figure 4.1. Per-fragment operations.

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 $\mathit{left} \leq x_w < \mathit{left} + \mathit{width}$ and $\mathit{bottom} \leq y_w < \mathit{bottom} + \mathit{height}$, 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, $\mathit{left} = \mathit{bottom} = 0$. *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`, and `SAMPLE_COVERAGE_INVERT`. No changes to the fragment alpha or coverage values are made at this step if `MULTISAMPLE` is disabled, or if the value of `SAMPLE_BUFFERS` is not one.

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

No specific algorithm is required for converting the sample alpha values to a temporary coverage value. It is intended that the number of 1's in the temporary coverage be proportional to the set of alpha values for the fragment, with all 1's corresponding to the maximum of all alpha values, and all 0's corresponding to all alpha values being 0. The alpha values used to generate a coverage value are clamped to the range $[0, 1]$. It is also intended that the algorithm be pseudo-random in nature, to avoid image artifacts due to regular coverage sample locations. The algorithm can and probably should be different at different pixel locations. If it does differ, it should be defined relative to window, not screen, coordinates, so that rendering results are invariant with respect to window position.

Next, if `SAMPLE_ALPHA_TO_ONE` is enabled, each alpha value is replaced by the maximum representable alpha value. Otherwise, the alpha values are not changed.

Finally, if `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`.

4.1.4 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 and lines) and front-facing polygon primitives while the back set of stencil state is used when processing fragments rasterized from back-facing polygon primitives. For the purposes of stencil testing, a primitive is still considered a polygon even if the polygon is to be rasterized as points or lines due to the current polygon mode. Whether a polygon is front- or back-facing is determined in the same manner used for two-sided lighting and face culling (see section 3.6.1).

StencilFuncSeparate and **StencilOpSeparate** take a *face* argument which can be `FRONT`, `BACK`, or `FRONT_AND_BACK` and indicates which set of state is affected. **StencilFunc** and **StencilOp** set front and back stencil state to identical values.

StencilFunc and **StencilFuncSeparate** take three arguments that control whether the stencil test passes or fails. *ref* is an integer reference value that is used in the unsigned stencil comparison. Stencil comparison operations and queries of *ref* clamp its value to the range $[0, 2^s - 1]$, where s is the number of bits in the stencil buffer attached to the draw framebuffer. The s least significant bits of *mask* are bitwise ANDed with both the reference and the stored stencil value, and the resulting masked values are those that participate in the comparison controlled by *func*. *func* is a symbolic constant that determines the stencil comparison function; the eight symbolic constants are `NEVER`, `ALWAYS`, `LESS`, `LEQUAL`, `EQUAL`, `GEQUAL`, `GREATER`, or `NOTEQUAL`. Accordingly, the stencil test passes never, always, and if the masked reference value is less than, less than or equal to, equal to, greater than or equal to, greater than, or not equal to the masked stored value in the stencil buffer.

StencilOp and **StencilOpSeparate** take three arguments that indicate what happens to the stored stencil value if this or certain subsequent tests fail or pass. *sfail* indicates what action is taken if the stencil test fails. The symbolic constants are `KEEP`, `ZERO`, `REPLACE`, `INCR`, `DECR`, `INVERT`, `INCR_WRAP`, and `DECR_WRAP`. These correspond to keeping the current value, setting to zero, replacing with the reference value, incrementing with saturation, decrementing with saturation, bitwise inverting it, incrementing without saturation, and decrementing without saturation.

For purposes of increment and decrement, the stencil bits are considered as an unsigned integer. Incrementing or decrementing with saturation clamps the stencil value at 0 and the maximum representable value. Incrementing or decrementing without saturation will wrap such that incrementing the maximum representable value results in 0, and decrementing 0 results in the maximum representable value.

The same symbolic values are given to indicate the stencil action if the depth buffer test (see section 4.1.5) fails (*dpfail*), or if it passes (*dppass*).

If the stencil test fails, the incoming fragment is discarded. The state required consists of the most recent values passed to **StencilFunc** or **StencilFuncSeparate** and to **StencilOp** or **StencilOpSeparate**, and a bit indicating whether stencil testing is enabled or disabled. In the initial state, stenciling is disabled, the front and back stencil reference value are both zero, the front and back stencil comparison functions are both `ALWAYS`, and the front and back stencil mask are both set to the value $2^s - 1$, where s is greater than or equal to the number of bits in the deepest stencil buffer supported by the GL implementation. Initially, all three front and back stencil operations are `KEEP`.

If there is no stencil buffer, no stencil modification can occur, and it is as if the stencil tests always pass, regardless of any calls to **StencilFunc**.

4.1.5 Depth Buffer Test

The depth buffer test discards the incoming fragment if a depth comparison fails. The comparison is enabled or disabled with the generic **Enable** and **Disable** commands using the symbolic constant `DEPTH_TEST`. When disabled, the depth comparison and subsequent possible updates to the depth buffer value are bypassed and the fragment is passed to the next operation. The stencil value, however, is modified as indicated below as if the depth buffer test passed. If enabled, the comparison takes place and the depth buffer and stencil value may subsequently be modified.

The comparison is specified with

`void` **DepthFunc**(`enum` *func*);

This command takes a single symbolic constant: one of `NEVER`, `ALWAYS`, `LESS`, `LEQUAL`, `EQUAL`, `GREATER`, `GEQUAL`, `NOTEQUAL`. Accordingly, the depth buffer test passes never, always, if the incoming fragment's z_w value is less than, less than or equal to, equal to, greater than, greater than or equal to, or not equal to the depth value stored at the location given by the incoming fragment's (x_w, y_w) coordinates.

If the depth buffer test fails, the incoming fragment is discarded. The stencil value at the fragment's (x_w, y_w) coordinates is updated according to the function currently in effect for depth buffer test failure. Otherwise, the fragment continues to the next operation and the value of the depth buffer at the fragment's (x_w, y_w) location is set to the fragment's z_w value. In this case the stencil value is updated according to the function currently in effect for depth buffer test success.

The necessary state is an eight-valued integer and a single bit indicating whether depth buffering is enabled or disabled. In the initial state the function is `LESS` and the test is disabled.

If there is no depth buffer, it is as if the depth buffer test always passes.

4.1.6 Occlusion Queries

Occlusion queries use query objects to track the number of fragments or samples that pass the depth test. An occlusion query can be started and finished by calling **BeginQuery** and **EndQuery**, respectively, with a *target* of `SAMPLES_PASSED`.

When an occlusion query is started, the samples-passed count maintained by the GL is set to zero. When an occlusion query is active, the samples-passed count is incremented for each fragment that passes the depth test. If the value of `SAMPLE_BUFFERS` is 0, then the samples-passed count is incremented by 1 for each fragment. If the value of `SAMPLE_BUFFERS` is 1, then the samples-passed count is incremented by the number of samples whose coverage bit is set. However, implementations, at their discretion, may instead increase the samples-passed count by the value of `SAMPLES` if any sample in the fragment is covered.

When an occlusion query finishes and all fragments generated by commands issued prior to **EndQuery** have been generated, the samples-passed count is written to the corresponding query object as the query result value, and the query result for that object is marked as available.

If the samples-passed count overflows (exceeds the value $2^n - 1$, where n is the number of bits in the samples-passed count), its value becomes undefined. It is recommended, but not required, that implementations handle this overflow case by saturating at $2^n - 1$ and incrementing no further.

The necessary state is a single bit indicating whether an occlusion query is active, the identifier of the currently active occlusion query, and a counter keeping track of the number of samples that have passed.

4.1.7 Blending

Blending combines the incoming *source* fragment's R, G, B, and A values with the *destination* R, G, B, and A values stored in the framebuffer at the fragment's (x_w, y_w) location.

Source and destination values are combined according to the *blend equation*, quadruplets of source and destination weighting factors determined by the *blend functions*, and a constant *blend color* to obtain a new set of R, G, B, and A values, as described below.

If the color buffer is fixed-point, the components of the source and destination values and blend factors are clamped to $[0, 1]$ prior to evaluating the blend equation. If the color buffer is floating-point, no clamping occurs. The resulting four values are sent to the next operation.

Blending applies only if the color buffer has a fixed-point or floating-point format. If the color buffer has an integer format, proceed to the next operation.

Blending is enabled or disabled for an individual draw buffer with the commands

void **Enablei**(enum *target*, uint *index*);
void **Disablei**(enum *target*, uint *index*);

target is the symbolic constant `BLEND` and *index* is an integer i specifying the draw buffer associated with the symbolic constant `DRAW_BUFFER`i. If the color buffer associated with `DRAW_BUFFER`i is one of `FRONT`, `BACK`, `LEFT`, `RIGHT`, or `FRONT_AND_BACK` (specifying multiple color buffers), then the state enabled or disabled is applicable for all of the buffers. Blending can be enabled or disabled for all draw buffers using **Enable** or **Disable** with the symbolic constant `BLEND`. If blending is disabled for a particular draw buffer, or if logical operation on color values is enabled (section 4.1.10), proceed to the next operation.

An `INVALID_VALUE` error is generated if *index* is greater than the value of `MAX_DRAW_BUFFERS` minus one.

If multiple fragment colors are being written to multiple buffers (see section 4.2.1), blending is computed and applied separately for each fragment color and the corresponding buffer.

Blend Equation

Blending is controlled by the *blend equations*, defined by the commands

void **BlendEquation**(enum *mode*);
void **BlendEquationSeparate**(enum *modeRGB*,
enum *modeAlpha*);

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

Signed or unsigned normalized fixed-point destination (framebuffer) components are represented as described in section 2.1.5. Constant color components, floating-point destination components, and source (fragment) components are taken to be floating point values. If source components are represented internally by the GL as fixed-point values, they are also interpreted according to section 2.1.5.

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

If `FRAMEBUFFER_SRGB` is enabled and the value of `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` for the framebuffer attachment corresponding to the destination buffer is `SRGB` (see section 6.1.3), the R, G, and B destination color values (after conversion from fixed-point to floating-point) are considered to be encoded for the sRGB color space and hence must be linearized prior to their use in blending. Each R, G, and B component is converted in the same fashion described for sRGB texture components in section 3.8.15.

If `FRAMEBUFFER_SRGB` is disabled or the value of `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` is not `SRGB`, no linearization is performed.

The resulting linearized R, G, and B and unmodified A values are recombined as the destination color used in blending computations.

Table 4.1 provides the corresponding per-component blend equations for each mode, whether acting on RGB components for *modeRGB* or the alpha component for *modeAlpha*.

In the table, the s subscript on a color component abbreviation (R, G, B, or A) refers to the source color component for an incoming fragment, the d subscript on a color component abbreviation refers to the destination color component at the corresponding framebuffer location, and the c subscript on a color component abbreviation refers to the constant blend color component. A color component abbreviation without a subscript refers to the new color component resulting from blending. Additionally, S_r, S_g, S_b, and S_a are the red, green, blue, and alpha components of the source weighting factors determined by the source blend function, and D_r, D_g, D_b, and D_a are the red, green, blue, and alpha components of the destination weighting factors determined by the destination blend function. Blend functions are described below.

Blend Functions

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

```
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*

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.

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

`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`

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] $f = \min(A_s, 1 - A_d)$.

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.

4.1.8 sRGB Conversion

If `FRAMEBUFFER_SRGB` is enabled and the value of `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` for the framebuffer attachment corresponding to the destination buffer is `SRGB` (see section 6.1.3), the R, G, and B values after blending are converted into the non-linear sRGB color space by computing

$$c_s = \begin{cases} 0.0, & c_l \leq 0 \\ 12.92 c_l, & 0 < c_l < 0.0031308 \\ 1.055 c_l^{0.41666} - 0.055, & 0.0031308 \leq c_l < 1 \\ 1.0, & c_l \geq 1 \end{cases} \tag{4.1}$$

where c_l is the R, G, or B element and c_s is the result (effectively converted into an sRGB color space).

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

$$c_s = c_l.$$

The resulting c_s values for R, G, and B, and the unmodified A form a new RGBA color value. If the color buffer is fixed-point, each component is clamped to the range $[0, 1]$ and then converted to a fixed-point value using equation 2.4. The resulting four values are sent to the subsequent dithering operation.

4.1.9 Dithering

Dithering selects between two representable color values or indices. A representable value is a value that has an exact representation in the color buffer. Dithering selects, for each color component, either the largest positive representable color value (for that particular color component) that is less than or equal to the incoming color component value, c, or the smallest negative representable color value that is greater than or equal to c. The selection may depend on the x_w and y_w coordinates of the pixel, as well as on the exact value of c. If one of the two values does not exist, then the selection defaults to the other value.

Many dithering selection algorithms are possible, but an individual selection must depend only on the incoming component value and the fragment's x and y window coordinates. If dithering is disabled, then each incoming color component c is replaced with the largest positive representable color value (for that particular component) that is less than or equal to c, or by the smallest negative representable value, if no representable value is less than or equal to c.

Dithering is enabled with **Enable** and disabled with **Disable** using the symbolic constant `DITHER`. The state required is thus a single bit. Initially, dithering is enabled.

4.1.10 Logical Operation

Finally, a logical operation is applied between the incoming fragment's color values and the color values stored at the corresponding location in the framebuffer. The result replaces the values in the framebuffer at the fragment's (x_w, y_w) coordinates. If the selected draw buffers refer to the same framebuffer-attachable image more than once, then the values stored in that image are undefined.

The logical operation on color values is enabled or disabled with **Enable** or **Disable** using the symbolic constant `COLOR_LOGIC_OP`. If the logical operation is enabled for color values, it is as if blending were disabled, regardless of the value of `BLEND`. If multiple fragment colors are being written to multiple buffers (see section 4.2.1), the logical operation is computed and applied separately for each fragment color and the corresponding buffer.

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 red, green, blue, and alpha value of each color buffer that is selected for writing. The required state is an integer indicating the logical operation, and a bit indicating whether the logical operation is enabled or disabled. The initial state is for the logic operation to be given by `COPY`, and to be disabled.

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.

4.1.11 Additional Multisample Fragment Operations

If the **DrawBuffer** mode is `NONE`, no change is made to any multisample or color buffer. Otherwise, fragment processing is as described below.

If `MULTISAMPLE` is enabled, and the value of `SAMPLE_BUFFERS` is one, the stencil test, depth test, blending, dithering, and logical operations are performed for each pixel sample, rather than just once for each fragment. Failure of the stencil or depth test results in termination of the processing of that sample, rather than discarding of the fragment. All operations are performed on the color, depth, and stencil values stored in the multisample buffer (to be described in a following section). The contents of the color buffers are not modified at this point.

Stencil, depth, blending, dithering, and logical operations are performed for a pixel sample only if that sample's fragment coverage bit is a value of 1. If the corresponding coverage bit is 0, no operations are performed for that sample.

If `MULTISAMPLE` is disabled, and the value of `SAMPLE_BUFFERS` is one, the fragment may be treated exactly as described above, with optimization possible because the fragment coverage must be set to full coverage. Further optimization is allowed, however. An implementation may choose to identify a centermost sample, and to perform stencil and depth tests on only that sample. Regardless of the outcome of the stencil test, all multisample buffer stencil sample values are set to the appropriate new stencil value. If the depth test passes, all multisample buffer depth sample values are set to the depth of the fragment's centermost sample's depth value, and all multisample buffer color sample values are set to the color value of the incoming fragment. Otherwise, no change is made to any multisample buffer color or depth value.

After all operations have been completed on the multisample buffer, the sample values for each color in the multisample buffer are combined to produce a single color value, and that value is written into the corresponding color buffers selected by **DrawBuffer** or **DrawBuffers**. An implementation may defer the writing of the color buffers until a later time, but the state of the framebuffer must behave as if the color buffers were updated as each fragment was processed. The method of combination is not specified. If the framebuffer contains sRGB values, then it is recommended that the an average of sample values is computed in a linearized space, as for blending (see section 4.1.7). Otherwise, a simple average computed independently for each color component is recommended.

4.2 Whole Framebuffer Operations

The preceding sections described the operations that occur as individual fragments are sent to the framebuffer. This section describes operations that control or affect 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). Arguments that omit reference to `LEFT` or `RIGHT` refer to both left and right buffers. Arguments that omit reference to `FRONT` or `BACK` refer to both front and back buffers.

If the GL is bound to a framebuffer object, *buf* must be one of the values listed in table 4.5, which summarizes the constants and the buffers they indicate. In this case, *buf* is a symbolic constant specifying a single color buffer for writing. Specifying `COLOR_ATTACHMENT`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`.

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 `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.

Symbolic Constant	Front Left	Front Right	Back Left	Back Right
`NONE`				
`FRONT_LEFT`	●			
`FRONT_RIGHT`		●		
`BACK_LEFT`			●	
`BACK_RIGHT`				●
`FRONT`	●	●		
`BACK`			●	●
`LEFT`	●		●	
`RIGHT`		●		●
`FRONT_AND_BACK`	●	●	●	●

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

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
`NONE`				
`FRONT_LEFT`	●			
`FRONT_RIGHT`		●		
`BACK_LEFT`			●	
`BACK_RIGHT`				●

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

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 a fragment shader writes to `gl_FragColor`, **DrawBuffers** specifies a set of draw buffers into which the single fragment color defined by `gl_FragColor` is written. If a fragment shader writes to `gl_FragData`, or a user-defined varying out variable, **DrawBuffers** specifies a set of draw buffers into which each of the multiple output colors defined by these variables are separately written. If a fragment shader writes to none of `gl_FragColor`, `gl_FragData`, nor any user-defined varying out variables, the values of the fragment colors following shader execution are undefined, and may differ for each fragment color.

For both the default framebuffer and framebuffer objects, the constants `FRONT`, `BACK`, `LEFT`, `RIGHT`, and `FRONT_AND_BACK` are not valid in the *bufs* array passed to **DrawBuffers**, and will result in the error `INVALID_OPERATION`. This restriction is because these constants may themselves refer to multiple buffers, as shown in table 4.4.

If the GL is bound to the default framebuffer and **DrawBuffers** is supplied with a constant (other than `NONE`) that does not indicate any of the color buffers allocated to the GL context by the window system, the error `INVALID_OPERATION` will be generated.

If the GL is bound to a framebuffer object and **DrawBuffers** is supplied with a constant from table 4.6, or `COLOR_ATTACHMENT`m where m is greater than or equal to the value of `MAX_COLOR_ATTACHMENTS`, then the error `INVALID_OPERATION` results.

Indicating a buffer or buffers using **DrawBuffer** or **DrawBuffers** causes subsequent pixel color value writes to affect the indicated buffers.

Specifying `NONE` as the draw buffer for a fragment color will inhibit that fragment color from being written to any buffer.

Monoscopic contexts include only left buffers, while stereoscopic contexts include both left and right buffers. Likewise, single buffered contexts include only front buffers, while double buffered contexts include both front and back buffers.

The type of context is selected at GL initialization.

The state required to handle color buffer selection for each framebuffer is an integer for each supported fragment color. For the default framebuffer, in the initial state the draw buffer for fragment color zero is `BACK` if there is a back buffer; `FRONT` if there is no back buffer; and `NONE` if no default framebuffer is associated with the context. For framebuffer objects, in the initial state the draw buffer for fragment color zero is `COLOR_ATTACHMENT0`. For both the default framebuffer and framebuffer objects, the initial state of draw buffers for fragment colors other then zero is `NONE`.

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

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 color values are enabled for writing for all draw buffers.

The value of the color writemask for draw buffer i can be queried by calling **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 4.1.4). The clear operation always uses the front stencil write mask when clearing the stencil buffer.

The state required for the various masking operations is two integers for the front and back stencil values, and a bit for depth values. A set of four bits is also required indicating which color components of an RGBA value should be written. In the initial state, the integer masks are all ones, as are the bits controlling depth value and RGBA component writing.

Fine Control of Multisample Buffer Updates

When the value of `SAMPLE_BUFFERS` is one, **ColorMask**, **DepthMask**, and **StencilMask** or **StencilMaskSeparate** control the modification of values in the multisample buffer. The color mask has no effect on modifications to the color buffers. If the color mask is entirely disabled, the color sample values must still be combined (as described above) and the result used to replace the color values of the buffers enabled by **DrawBuffer**.

4.2.3 Clearing the Buffers

The GL provides a means for setting portions of every pixel in a particular buffer to the same value. The argument to

void **Clear**(bitfield *buf*);

is the bitwise OR of a number of values indicating which buffers are to be cleared. The values are `COLOR_BUFFER_BIT`, `DEPTH_BUFFER_BIT`, `STENCIL_BUFFER_BIT`, and indicating the buffers currently enabled for color writing, the depth buffer, and the stencil buffer (see below), respectively. The value to which each buffer is cleared depends on the setting of the clear value for that buffer. If the mask is not a bitwise OR of the specified values, then the error `INVALID_VALUE` is generated.

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

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

The command

void **ClearDepth**(clampd *d*);

sets the depth value used when clearing the depth buffer. d is clamped to the range $[0, 1]$. When clearing a fixed-point depth buffer, d is converted to fixed-point according to the rules for a window z value given in section 2.12.1. No conversion is applied when clearing a floating-point depth buffer.

The command

void **ClearStencil**(int *s*);

takes a single integer argument that is the value to which to clear the stencil buffer. s is masked to the number of bitplanes in the stencil buffer.

When **Clear** is called, the only per-fragment operations that are applied (if enabled) are the pixel ownership test, the scissor test, and dithering. The masking operations described in section 4.2.2 are also applied. If a buffer is not present, then a **Clear** directed at that buffer has no effect. Unsigned normalized fixed-point and signed normalized fixed-point RGBA color buffers are cleared to color values derived by clamping each component of the clear color to the range $[0, 1]$ or $[-1, 1]$ respectively, then converting to fixed-point using equations 2.4 or 2.6, respectively. The result of clearing integer color buffers is undefined.

The state required for clearing is a clear value for each of the color buffer, the depth buffer, and the stencil buffer. Initially, the RGBA color clear value is $(0, 0, 0, 0)$, the depth buffer clear value is 1.0, and the stencil buffer clear index is 0.

Individual buffers of the currently bound draw framebuffer may be cleared with the command

`void` **ClearBuffer**$\{$**if ui**$\}$**v**`(` `enum` *buffer*`,` `int` *drawbuffer*`,`
`const T` **value* `);`

where *buffer* and *drawbuffer* identify a buffer to clear, and *value* specifies the value or values to clear it to.

If *buffer* is `COLOR`, a particular draw buffer `DRAW_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.

Clearing the Multisample Buffer

The color samples of the multisample buffer are cleared when one or more color buffers are cleared, as specified by the **Clear** mask bit `COLOR_BUFFER_BIT` and the **DrawBuffer** mode. If the **DrawBuffer** mode is `NONE`, the color samples of the multisample buffer cannot be cleared using **Clear**.

If the **Clear** mask bits `DEPTH_BUFFER_BIT` or `STENCIL_BUFFER_BIT` are set, then the corresponding depth or stencil samples, respectively, are cleared.

The **ClearBuffer** commands also clear color, depth, or stencil samples of multisample buffers corresponding to the specified buffer.

4.3 Reading and Copying Pixels

Pixels may be read from the framebuffer using **ReadPixels**. **BlitFramebuffer** can be used to copy a block of pixels from one portion of the framebuffer to another.

4.3.1 Reading Pixels

The method for reading pixels from the framebuffer and placing them in pixel pack buffer or client memory is diagrammed in figure 4.2. We describe the stages of the pixel reading process in the order in which they occur.

Initially, zero is bound for the `PIXEL_PACK_BUFFER`, indicating that image read and query commands such as **ReadPixels** return pixel results into client mem-

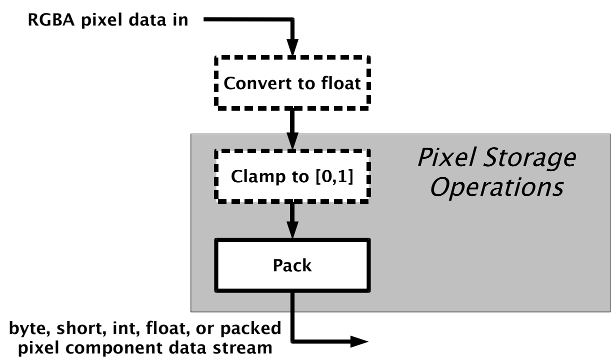


Figure 4.2. Operation of **ReadPixels**. Operations in dashed boxes are not performed for all data formats. Depth and stencil pixel paths are not shown.

ory pointer parameters. However, if a non-zero buffer object is bound as the current pixel pack buffer, then the pointer parameter is treated as an offset into the designated buffer object.

Pixels are read using

`void` **ReadPixels**(`int` *x*, `int` *y*, `sizei` *width*, `sizei` *height*, `enum` *format*, `enum` *type*, `void` **data*);

The arguments after *x* and *y* to **ReadPixels** are described in section 3.7.2. The pixel storage modes that apply to **ReadPixels** and other commands that query images (see section 6.1) are summarized in table 4.7.

ReadPixels generates an `INVALID_OPERATION` error if `READ_FRAMEBUFFER_BINDING` (see section 4.4) is non-zero, the read framebuffer is framebuffer complete, and the value of `SAMPLE_BUFFERS` for the read framebuffer is greater than zero.

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.

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**, and **GetTexImage**.

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

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

When `READ_FRAMEBUFFER_BINDING` is non-zero, the red, green, blue, and alpha values are obtained by first reading the internal component values of the corresponding value in the image attached to the selected logical buffer. Internal components are converted to an RGBA color by taking each R, G, B, and A component present according to the base internal format of the buffer (as shown in table 3.11). If G, B, or A values are not present in the internal format, they are taken to be zero, zero, and one respectively.

Conversion of RGBA values

This step applies only if *format* is not `STENCIL_INDEX`, `DEPTH_COMPONENT`, or `DEPTH_STENCIL`. The R, G, B, and A values form a group of elements.

For a signed or unsigned normalized fixed-point color buffer, each element is converted to floating-point using equations 2.3 or 2.1, respectively. For an integer or floating-point color buffer, the elements are unmodified.

Conversion of Depth values

This step applies only if *format* is `DEPTH_COMPONENT` or `DEPTH_STENCIL` and the depth buffer uses a fixed-point representation. An element is taken to be a fixed-point value in $[0, 1]$ with m bits, where m is the number of bits in the depth buffer (see section 2.12.1). No conversion is necessary if the depth buffer uses a floating-point representation.

Final Conversion

For an index, if the *type* is not `FLOAT` or `HALF_FLOAT`, final conversion consists of masking the index with the value given in table 4.8; if the *type* is `FLOAT` or `HALF_FLOAT`, then the integer index is converted to a GL `float` or `half` data value.

Read color clamping is controlled by calling

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

with *target* set to `CLAMP_READ_COLOR`. If *clamp* is `TRUE`, read color clamping is enabled; if *clamp* is `FALSE`, read color clamping is disabled. If *clamp* is `FIXED_ONLY`, read color clamping is enabled if the selected read color buffer has fixed-point components.

For a floating-point RGBA color, if *type* is not one of `FLOAT`, `HALF`, `UNSIGNED_INT_5_9_9_9_REV`, or `UNSIGNED_INT_10F_11F_11F_REV`; or if

type Parameter	Index Mask
`UNSIGNED_BYTE`	$2^8 - 1$
`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.

read color clamping is enabled, each component is first clamped to $[0, 1]$. Then the appropriate conversion formula from table 4.9 is applied to the component.

In the special case of calling **ReadPixels** with *type* of `UNSIGNED_INT_10F_11F_11F_REV` and *format* of `RGB`, conversion is performed as follows: the returned data are packed into a series of `uint` values. The red, green, and blue components are converted to unsigned 11-bit floating-point, unsigned 11-bit floating-point, and unsigned 10-bit floating point as described in sections 2.1.3 and 2.1.4. The resulting red 11 bits, green 11 bits, and blue 10 bits are then packed as the 1st, 2nd, and 3rd components of the `UNSIGNED_INT_10F_11F_11F_REV` format as shown in table 3.8.

In the special case of calling **ReadPixels** with *type* of `UNSIGNED_INT_5_9_9_9_REV` and *format* `RGB`, the conversion is performed as follows: the returned data are packed into a series of `uint` values. The red, green, and blue components are converted to red_s, $green_s$, $blue_s$, and exp_{shared} integers as described in section 3.8.1 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 3.8.

For an integer RGBA color, each component is clamped to the representable range of *type*.

Placement in Pixel Pack Buffer or Client Memory

If a pixel pack buffer is bound (as indicated by a non-zero value of `PIXEL_PACK_BUFFER_BINDING`), *data* is an offset into the pixel pack buffer and the pixels are packed into the buffer relative to this offset; otherwise, *data* is a pointer to a block client memory and the pixels are packed into the client memory relative to the

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.

pointer. If a pixel pack buffer object is bound and packing the pixel data according to the pixel pack storage state would access memory beyond the size of the pixel pack buffer's memory size, an `INVALID_OPERATION` error results. If a pixel pack buffer object is bound and *data* is not evenly divisible by the number of basic machine units needed to store in memory the corresponding GL data type from table 3.2 for the *type* parameter, an `INVALID_OPERATION` error results.

Groups of elements are placed in memory just as they are taken from memory when transferring pixel rectangles to the GL. That is, the 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 3.7.1. The only difference is that the storage mode parameters whose names begin with `PACK_` are used instead of those whose names begin with `UNPACK_`. If the *format* is `RED`, `GREEN`, `BLUE`, or `ALPHA`, only the corresponding single element is written. Likewise if the *format* is `RG`, `RGB`, or `BGR`, only the corresponding two or three elements are written. Otherwise all the elements of each group are written.

4.3.2 Copying Pixels

The command

`void` **BlitFramebuffer**(`int` *srcX0*, `int` *srcY0*, `int` *srcX1*,
`int` *srcY1*, `int` *dstX0*, `int` *dstY0*, `int` *dstX1*, `int` *dstY1*,
`bitfield` *mask*, `enum` *filter*);

transfers a rectangle of pixel values from one region of the read framebuffer to another in the draw framebuffer.

mask is the bitwise OR of a number of values indicating which buffers are to be copied. The values are `COLOR_BUFFER_BIT`, `DEPTH_BUFFER_BIT`, and `STENCIL_BUFFER_BIT`, which are described in section 4.2.3. The pixels corresponding to these buffers are copied from the source rectangle bounded by the locations $(srcX0, srcY0)$ and $(srcX1, srcY1)$ to the destination rectangle bounded by the locations $(dstX0, dstY0)$ and $(dstX1, dstY1)$. The lower bounds of the rectangle are inclusive, while the upper bounds are exclusive.

When the color buffer is transferred, values are taken from the read buffer of the read framebuffer and written to each of the draw buffers of the draw framebuffer.

The actual region taken from the read framebuffer is limited to the intersection of the source buffers being transferred, which may include the color buffer selected by the read buffer, the depth buffer, and/or the stencil buffer depending on *mask*. The actual region written to the draw framebuffer is limited to the intersection of the destination buffers being written, which may include multiple draw buffers, the depth buffer, and/or the stencil buffer depending on *mask*. Whether or not the source or destination regions are altered due to these limits, the scaling and offset applied to pixels being transferred is performed as though no such limits were present.

If the source and destination rectangle dimensions do not match, the source image is stretched to fit the destination rectangle. *filter* must be `LINEAR` or `NEAREST`, and specifies the method of interpolation to be applied if the image is stretched. `LINEAR` filtering is allowed only for the color buffer; if *mask* includes `DEPTH_BUFFER_BIT` or `STENCIL_BUFFER_BIT`, and filter is not `NEAREST`, no copy is performed and an `INVALID_OPERATION` error is generated. If the source and destination dimensions are identical, no filtering is applied. If either the source or destination rectangle specifies a negative width or height ($X1 < X0$ 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 a buffer is specified in *mask* and does not exist in both the read and draw framebuffers, the corresponding bit is silently ignored.

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

- The read buffer contains floating-point values and any draw buffer does not contain floating-point values.
- The read buffer contains non-floating-point values and any draw buffer contains floating-point values.
- The read buffer contains unsigned integer values and any draw buffer does not contain unsigned integer values.

- The read buffer contains signed integer values and any draw buffer does not contain signed integer values.

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.

4.3.3 Pixel Draw/Read State

The state required for pixel operations consists of the parameters that are set with **PixelStore**, This state has been summarized in tables 3.1, Additional state includes an integer indicating the current setting of **ReadBuffer**, and a three-valued integer controlling clamping during final conversion. For the default framebuffer, in the initial state the read buffer is `BACK` if there is a back buffer; `FRONT` if there is no back buffer; and `NONE` if no default framebuffer is associated with the context. The initial value of read color clamping is `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 6.20, as well as one set of the state values listed in table 6.21 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 6.41 and 6.23 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`.
- If the attachment sizes are not all identical, rendering will be limited to the largest area that can fit in all of the attachments (an intersection of rectangles having a lower left of $(0,0)$ and an upper right of $(width, height)$ for each attachment).
- If the attachment sizes are not all identical, the values of pixels outside the common intersection area after rendering are undefined.

Framebuffer objects are deleted by calling

`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, just as if they were unused.

The names bound to the draw and read framebuffer bindings can be queried by calling **GetIntegerv** with the symbolic constants `DRAW_FRAMEBUFFER_BINDING` and `READ_FRAMEBUFFER_BINDING`, respectively. `FRAMEBUFFER_BINDING` is equivalent to `DRAW_FRAMEBUFFER_BINDING`.

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 6.21

There are two types of framebuffer-attachable images: the image of a renderbuffer object, and an image of a texture object.

Renderbuffer Objects

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

The name space for renderbuffer objects is the unsigned integers, with zero reserved for the GL. A renderbuffer object is created by binding a name returned by **GenRenderbuffers** (see below) to `RENDERBUFFER`. The binding is effected by calling

`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 6.23. 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, just as if they were unused.

The command

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

`void` **RenderbufferStorageMultisample**(`enum` *target*, `sizei` *samples*, `enum` *internalformat*, `sizei` *width*, `sizei` *height*);

establishes the data storage, format, dimensions, and number of samples of a renderbuffer object's image. *target* must be `RENDERBUFFER`. *internalformat* must be color-renderable, depth-renderable, or stencil-renderable (as defined in section 4.4.4). *width* and *height* are the dimensions in pixels of the renderbuffer. If either *width* or *height* is greater than `MAX_RENDERBUFFER_SIZE`, or if *samples* is greater than `MAX_SAMPLES`, then the error `INVALID_VALUE` is generated. The error `INVALID_OPERATION` may be generated if *internalformat* is a signed or unsigned integer format, *samples* is greater than one, and the implementation does not support multisampled integer renderbuffers (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 3.8.1, with the exception of the color formats labelled "texture-only". Requesting one of these internal formats for a renderbuffer will allocate at least the internal component sizes and exactly the component types shown for that format in tables 3.12- 3.13.

Implementations must support creation of renderbuffers in these required formats with up to the value of `MAX_SAMPLES` multisamples, with the exception that the signed and unsigned integer formats are required only to support creation of renderbuffers with one sample.

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 4.11.

renderbuffertarget must be `RENDERBUFFER` and *renderbuffer* should be set to the name of the renderbuffer object to be attached to the framebuffer. *renderbuffer* must be either zero or the name of an existing renderbuffer object of type *renderbuffertarget*, otherwise an `INVALID_OPERATION` error is generated. If *renderbuffer* is zero, then the value of *renderbuffertarget* is ignored.

If *renderbuffer* is not zero and if **FramebufferRenderbuffer** is successful, then the renderbuffer named *renderbuffer* will be used as the logical buffer identified by *attachment* of the framebuffer currently bound to *target*. The value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` for the specified attachment point is set to `RENDERBUFFER` and the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` is set to *renderbuffer*. All other state values of the attachment point specified by *attachment* are set to their default values listed in table 6.21. 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 6.21.

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 4.11: Framebuffer attachment points. i in `COLOR_ATTACHMENT`i may range from zero to the value of `MAX_COLOR_ATTACHMENTS` - 1.

Attaching Texture Images to a Framebuffer

GL supports copying the rendered contents of the framebuffer into the images of a texture object through the use of the routines **CopyTexImage*** and **CopyTexSubImage***. Additionally, GL supports rendering directly into the images of a texture object.

To render directly into a texture image, a specified image from a texture object can be attached as one of the logical buffers of the currently bound framebuffer object by calling one of the following routines, depending on the type of the texture:

`void` **FramebufferTexture1D**(`enum` *target*, `enum` *attachment*,
  `enum` *textarget*, `uint` *texture*, `int` *level*);
`void` **FramebufferTexture2D**(`enum` *target*, `enum` *attachment*,
  `enum` *textarget*, `uint` *texture*, `int` *level*);
`void` **FramebufferTexture3D**(`enum` *target*, `enum` *attachment*,
  `enum` *textarget*, `uint` *texture*, `int` *level*, `int` *layer*);

In all three routines, *target* must be `DRAW_FRAMEBUFFER`, `READ_FRAMEBUFFER`, or `FRAMEBUFFER`. `FRAMEBUFFER` is equivalent to `DRAW_FRAMEBUFFER`. An `INVALID_OPERATION` error is generated if the value of the corresponding binding is zero. *attachment* must be one of the attachment points of the framebuffer listed in table 4.11.

If *texture* is zero, the image identified by *attachment*, if any, will be detached from the framebuffer currently bound to *target*. *textarget*, *level*, and *layer* are ignored. All state values of the attachment point specified by *attachment* are set to their default values listed in table 6.21.

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`, 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`, or `TEXTURE_CUBE_MAP_NEGATIVE_Z`.

For **FramebufferTexture3D**, if *texture* is not zero, then *textarget* must be `TEXTURE_3D`.

If *texture* is not zero, and if **FramebufferTexture*** is successful, then the specified texture image will be used as the logical buffer identified by *attachment* of the framebuffer currently bound to *target*. The value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` for the specified attachment point is set to `TEXTURE` and the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` is set to *texture*. Additionally, the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL` for the named attachment point is set to *level*. If *texture* is a cube map texture, then the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE` for the named attachment point is set to *textarget*. If *texture* is a 3D texture, then the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER` for the named attachment point is set to *layer*. All other state values of the attachment point specified by *attachment* are set to their default values listed in table 6.21. No change is made to the state of the texture object, and any previous attachment to the *attachment* logical buffer of the framebuffer object bound to framebuffer *target* is broken. If the attachment is not successful, then no change is made to the state of either the texture object or the framebuffer object.

Setting *attachment* to the value `DEPTH_STENCIL_ATTACHMENT` is a special case causing both the depth and stencil attachments of the framebuffer object to be set to *texture*. *texture* must have base internal format `DEPTH_STENCIL`, or the depth and stencil framebuffer attachments will be incomplete (see section 4.4.4).

The command

`void` **FramebufferTextureLayer**(`enum` *target*,
`enum` *attachment*, `uint` *texture*, `int` *level*, `int` *layer*);

operates identically to **FramebufferTexture3D**, except that it attaches a single layer of a three-dimensional texture or a one- or two-dimensional array texture. *layer* is an integer indicating the layer number, and is treated identically to the *layer* parameter in **FramebufferTexture3D**. The error `INVALID_VALUE` is generated if *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 texture or one- or two-dimensional array texture. Unlike **FramebufferTexture3D**, no *textarget* parameter is accepted.

If *texture* is non-zero and the command does not result in an error, the framebuffer attachment state corresponding to *attachment* is updated as in the other **FramebufferTexture** commands, except that `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER` is set to *layer*.

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

4.4.3 Feedback Loops Between Textures and the Framebuffer

A *feedback loop* may exist when a texture object is used as both the source and destination of a GL operation. When a feedback loop exists, undefined behavior results. This section describes *rendering feedback loops* (see section 3.8.8) and *texture copying feedback loops* (see section 3.8.2) in more detail.

Rendering Feedback Loops

The mechanisms for attaching textures to a framebuffer object do not prevent a one- or two-dimensional texture level, a face of a cube map texture level, or a layer of a two-dimensional array or three-dimensional texture from being attached to the draw framebuffer while the same texture is bound to a texture unit. While this conditions holds, texturing operations accessing that image will produce undefined results, as described at the end of section 3.8.8. Conditions resulting in such undefined behavior are defined in more detail below. Such undefined texturing operations are likely to leave the final results of fragment processing operations undefined, and should be avoided.

Special precautions need to be taken to avoid attaching a texture image to the currently bound framebuffer while the texture object is currently bound and enabled for texturing. Doing so could lead to the creation of a rendering feedback loop between the writing of pixels by GL rendering operations and the simultaneous reading of those same pixels when used as texels in the currently bound texture. In this scenario, the framebuffer will be considered framebuffer complete (see section 4.4.4), but the values of fragments rendered while in this state will be undefined. The values of texture samples may be undefined as well, as described under "Rendering Feedback Loops" in section 3.8.8

Specifically, the values of rendered fragments are undefined if all of the following conditions are true:

- an image from texture object *T* is attached to the currently bound draw framebuffer at attachment point *A*
- the texture object *T* is currently bound to a texture unit *U*, and
- the current programmable vertex and/or fragment processing state makes it possible (see below) to sample from the texture object *T* bound to texture unit *U*

while either of the following conditions are true:

- the value of `TEXTURE_MIN_FILTER` for texture object *T* is `NEAREST` or `LINEAR`, and the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL` for attachment point *A* is equal to the value of `TEXTURE_BASE_LEVEL` for the texture object *T*
- 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 3.8.8).

For the purpose of this discussion, it is *possible* to sample from the texture object *T* bound to texture unit *U* if the active fragment or vertex shader contains any instructions that might sample from the texture object *T* bound to *U*, even if those instructions might only be executed conditionally.

Note that if `TEXTURE_BASE_LEVEL` and `TEXTURE_MAX_LEVEL` exclude any levels containing image(s) attached to the currently bound 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 3.8.2. While this condition holds, a texture copying feedback loop between the writing of texels by the copying operation and the reading of those same texels when used as pixels in the read framebuffer may exist. In this scenario, the values of texels written by the copying operation will be undefined (in the same fashion that overlapping copies via **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 3.11 are *color-renderable*: `RED`, `RG`, `RGB`, and `RGBA`. The sized internal formats from table 3.12 that have a color-renderable base internal format are also color-renderable. No other formats, including compressed internal formats, are color-renderable.

- An internal format is *depth-renderable* if it is `DEPTH_COMPONENT` or one of the formats from table 3.13 whose base internal format is `DEPTH_COMPONENT` or `DEPTH_STENCIL`. No other formats are depth-renderable.
- An internal format is *stencil-renderable* if it is `STENCIL_INDEX` or `DEPTH_STENCIL`, if it is one of the `STENCIL_INDEX` formats from table 4.10, or if it is one of the formats from table 3.13 whose base internal format is `DEPTH_STENCIL`. No other formats are stencil-renderable.

Framebuffer Attachment Completeness

If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` for the framebuffer attachment point *attachment* is not `NONE`, then it is said that a framebuffer-attachable image, named *image*, is attached to the framebuffer at the attachment point. *image* is identified by the state in *attachment* as described in section 4.4.2.

The framebuffer attachment point *attachment* is said to be *framebuffer attachment complete* if the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` for *attachment* is `NONE` (i.e., no image is attached), or if all of the following conditions are true:

- *image* is a component of an existing object with the name specified by `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME`, and of the type specified by `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE`.
- The width and height of *image* are non-zero.
- If `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is `TEXTURE` and `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` names a three-dimensional texture, then `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER` must be smaller than the depth of the texture.
- If `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is `TEXTURE` and `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` names a one- or two-dimensional array texture, then `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER` must be smaller than the number of layers in the texture.
- 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; and, if the attached images are a mix of renderbuffers and textures, the value of `RENDERBUFFER_SAMPLES` is zero for all attached renderbuffers.

  { `FRAMEBUFFER_INCOMPLETE_MULTISAMPLE` }

The token in brackets after each clause of the framebuffer completeness rules specifies the return value of **CheckFramebufferStatus** (see below) that is generated when that clause is violated. If more than one clause is violated, it is implementation-dependent which value will be returned by **CheckFramebufferStatus**.

Performing any of the following actions may change whether the framebuffer is considered complete or incomplete:

- Binding to a different framebuffer with **BindFramebuffer**.
- Attaching an image to the framebuffer with **FramebufferTexture*** or **FramebufferRenderbuffer**.
- Detaching an image from the framebuffer with **FramebufferTexture*** or **FramebufferRenderbuffer**.
- Changing the internal format of a texture image that is attached to the framebuffer by calling **CopyTexImage*** or **CompressedTexImage***.
- Changing the internal format of a renderbuffer that is attached to the framebuffer by calling **RenderbufferStorage**.
- Deleting, with **DeleteTextures** or **DeleteRenderbuffers**, an object containing an image that is attached to a framebuffer object that is bound to the framebuffer.
- Changing the read buffer or one of the draw buffers.
- Associating a different window system-provided drawable, or no drawable, with the default framebuffer using a window system binding API such as those described in section 1.6.2.

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** 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 3.8.1 and 4.4.2. The depth attachment may be in any of the required depth or combined depth+stencil formats described in those sections, and the stencil attachment may be in any of the required combined depth+stencil formats.

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

Effects of Framebuffer Completeness on Framebuffer Operations

Attempting to render to or read from a framebuffer which is not framebuffer complete will generate an `INVALID_FRAMEBUFFER_OPERATION` error. This means that rendering commands such as **DrawArrays** or one of the other drawing commands defined in section 2.8.2, 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.

4.4.5 Effects of Framebuffer State on Framebuffer Dependent Values

The values of the state variables listed in table 6.41 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 6.41 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 6.41 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 6.41 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 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 3.6 as follows:

$$i = (x_w - b)$$

$$j = (y_w - b)$$

$$k = (layer - b)$$

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

(x_w, y_w) corresponds to a border texel if x_w, y_w, or $layer$ is less than the border width, or if x_w, y_w, or $layer$ is greater than or equal to the border width plus the width, height, or depth, respectively, of the texture image.

Conversion to Framebuffer-Attachable Image Components

When an enabled color value is written to the framebuffer while the draw framebuffer binding is non-zero, for each draw buffer the R, G, B, and A values are converted to internal components as described in table 3.11, according to the table row corresponding to the internal format of the framebuffer-attachable image attached to the selected logical buffer, and the resulting internal components are written to the image attached to logical buffer. The masking operations described in section 4.2.2 are also effective.

Conversion to RGBA Values

When a color value is read or is used as the source of a logical operation or blending while the read framebuffer binding is non-zero, the components of the framebuffer-attachable image that is attached to the logical buffer selected by `READ_BUFFER` are first converted to R, G, B, and A values according to table 3.20 and the internal format of the attached image.

Chapter 5

Special Functions

This chapter describes additional GL functionality that does not fit easily into any of the preceding chapters. This functionality consists of flushing and finishing (used to synchronize the GL command stream), and hints.

5.1 Flush and Finish

The command

`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.

5.2 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	Hint description
`LINE_SMOOTH_HINT`	Line sampling quality
`POLYGON_SMOOTH_HINT`	Polygon sampling quality
`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 5.1: Hint targets and descriptions.

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 5.1; for each *target*, *hint* must be one of `FASTEST`, indicating that the most efficient option should be chosen; `NICEST`, indicating that the highest quality option should be chosen; and `DONT_CARE`, indicating no preference in the matter.

For the texture compression hint, a *hint* of `FASTEST` indicates that texture images should be compressed as quickly as possible, while `NICEST` indicates that the texture images be compressed with as little image degradation as possible. `FASTEST` should be used for one-time texture compression, and `NICEST` should be used if the compression results are to be retrieved by **GetCompressedTexImage** (section 6.1.4) for reuse.

The interpretation of hints is implementation-dependent. An implementation may ignore them entirely.

The initial value of all hints is `DONT_CARE`.

Chapter 6

State and State Requests

The state required to describe the GL machine is enumerated in section 6.2. Most state is set through the calls described in previous chapters, and can be queried using the calls described in section 6.1.

6.1 Querying GL State

6.1.1 Simple Queries

Much of the GL state is completely identified by symbolic constants. The values of these state variables can be obtained using a set of **Get** commands. There are four commands for obtaining simple state variables:

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

The commands obtain boolean, integer, floating-point, or double-precision state variables. *value* is a symbolic constant indicating the state variable to return. *data* is a pointer to a scalar or array of the indicated type in which to place the returned data.

Indexed simple state variables are queried with the commands

```
void GetBooleani_v( enum target, uint index,
   boolean *data );
void GetIntegeri_v( enum target, uint index, int *data );
```

target is the name of the indexed state and *index* is the index of the particular element being queried. *data* is a pointer to a scalar or array of the indicated type in which to place the returned data. An `INVALID_VALUE` error is generated if *index* is outside the valid range for the indexed state *target*.

Finally,

`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 **GetIntegerv** (or any of the **Get** commands below) is called, a boolean value of `TRUE` or `FALSE` is interpreted as 1 or 0, respectively, and a floating-point value is rounded to the nearest integer, unless the value is an RGBA color component, a **DepthRange** value, or a depth buffer clear value. In these cases, the **Get** command converts the floating-point value to an integer according to the `INT` entry of table 4.9; a value not in $[-1, 1]$ converts to an undefined value. If **GetFloatv** is called, a boolean value of `TRUE` or `FALSE` is interpreted as 1.0 or 0.0, respectively, an integer is coerced to floating-point, and a double-precision floating-point value is converted to single-precision. Analogous conversions are carried out in the case of **GetDoublev**. If a value is so large in magnitude that it cannot be represented with the requested type, then the nearest value representable using the requested type is returned.

Unless otherwise indicated, multi-valued state variables return their multiple values in the same order as they are given as arguments to the commands that set them. For instance, the two **DepthRange** parameters are returned in the order n followed by f.

If fragment color clamping is enabled, querying of the texture border color, blend color, and RGBA clear color will clamp the corresponding state values to

$[0, 1]$ before returning them. This behavior provides compatibility with previous versions of the GL that clamped these values when specified.

Most texture state variables are qualified by the value of `ACTIVE_TEXTURE` to determine which server texture state vector is queried. Table 6.12 indicates those state variables which are qualified by `ACTIVE_TEXTURE` during state queries. All other texture state queries will result in an `INVALID_OPERATION` error if the value of `ACTIVE_TEXTURE` is greater than or equal to `MAX_COMBINED_TEXTURE_IMAGE_UNITS`.

Vertex array state variables are qualified by the value of `VERTEX_ARRAY_BINDING` to determine which vertex array object is queried. Tables 6.3 and 6.4 define the set of state stored in a vertex array object.

6.1.3 Enumerated Queries

Other commands exist to obtain state variables that are identified by a category as well as a symbolic constant.

The commands

`void` **GetTexParameter{if}v**(`enum` *target*, `enum` *value*,
`T` *data*);
`void` **GetTexParameterI{i ui}v**(`enum` *target*, `enum` *value*,
`T` *data*);

place information about texture parameter *value* for the specified *target* into *data*. *value* must be one of the symbolic values in table 3.16.

target may be one of `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`, `TEXTURE_RECTANGLE`, or `TEXTURE_CUBE_MAP`, indicating the currently bound one-, two-, three-dimensional, one- or two-dimensional array, rectangular, or cube map texture object.

Querying *value* `TEXTURE_BORDER_COLOR` with **GetTexParameterIiv** or **GetTexParameterIuiv** returns the border color values as signed integers or unsigned integers, respectively; otherwise the values are returned as described in section 6.1.2. If the border color is queried with a type that does not match the original type with which it was specified, the result is undefined.

`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 6.14.

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`, `PROXY_TEXTURE_1D`, `PROXY_TEXTURE_2D`, `PROXY_TEXTURE_3D`, `PROXY_TEXTURE_1D_ARRAY`, `PROXY_TEXTURE_2D_ARRAY`, `PROXY_TEXTURE_RECTANGLE`, or `PROXY_TEXTURE_CUBE_MAP`, 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, or the one-, two-, three-dimensional, one- or two-dimensional array, rectangular, or cube map proxy state vector.

target may also be `TEXTURE_BUFFER`, indicating the texture buffer. In the case *lod* must be zero or an `INVALID_VALUE` error is generated.

Note that `TEXTURE_CUBE_MAP` is not a valid *target* parameter for **GetTexLevelParameter**, because it does not specify a particular cube map face.

lod determines which level-of-detail's state is returned. If *lod* is less than zero or larger than the maximum allowable level-of-detail, then an `INVALID_VALUE` error is generated.

For texture images with uncompressed internal formats, queries of *value* `TEXTURE_RED_TYPE`, `TEXTURE_GREEN_TYPE`, `TEXTURE_BLUE_TYPE`, `TEXTURE_ALPHA_TYPE`, and `TEXTURE_DEPTH_TYPE` return the data type used to store the component. Types `NONE`, `SIGNED_NORMALIZED`, `UNSIGNED_NORMALIZED`, `FLOAT`, `INT`, and `UNSIGNED_INT` respectively indicate missing, signed normalized fixed-point, unsigned normalized fixed-point, floating-point, signed unnormalized integer, and unsigned unnormalized integer components. Queries of *value* `TEXTURE_RED_SIZE`, `TEXTURE_GREEN_SIZE`, `TEXTURE_BLUE_SIZE`, `TEXTURE_ALPHA_SIZE`, `TEXTURE_DEPTH_SIZE`, `TEXTURE_STENCIL_SIZE`, and `TEXTURE_SHARED_SIZE` return the actual resolutions of the stored image array components, not the resolutions specified when the image array was defined. For texture images with a compressed internal format, the resolutions returned specify the component resolution of an uncompressed internal format that produces an image of roughly the same quality as the compressed image in question. Since the quality of the implementation's compression algorithm 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_WIDTH`, `TEXTURE_HEIGHT`, `TEXTURE_DEPTH`, and `TEXTURE_BORDER` return the width, height, depth, and border as specified when the image array was created. The internal format of the image array is queried as `TEXTURE_INTERNAL_FORMAT`.

6.1.4 Texture Queries

The command

```
void GetTexImage( enum tex, int lod, enum format,
   enum type, void *img );
```

is used to obtain texture images. It is somewhat different from the other **Get*** commands; *tex* is a symbolic value indicating which texture (or texture face in the case of a cube map texture target name) is to be obtained. `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`, and `TEXTURE_RECTANGLE` indicate a one-, two-, or three-dimensional, one- or two-dimensional array, or rectangular texture respectively. `TEXTURE_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, and `TEXTURE_CUBE_MAP_NEGATIVE_Z` indicate the respective face of a cube map texture. *lod* is a level-of-detail number, *format* is a pixel format from table 3.3, *type* is a pixel type from table 3.2.

Any of the following mismatches between *format* and the internal format of the texture image will generate an `INVALID_OPERATION` error:

- *format* is a color format (one of the formats in table 3.3 whose target is the color buffer) and the base internal format of the texture image is not a color format.
- *format* is `DEPTH_COMPONENT` and the base internal format is not `DEPTH_COMPONENT` or `DEPTH_STENCIL`.
- *format* is `DEPTH_STENCIL` and the base internal format is not `DEPTH_STENCIL`.
- *format* is one of the integer formats in table 3.3 and the internal format of the texture image is not integer, or *format* is not one of the integer formats in table 3.3 and the internal format is integer.

GetTexImage obtains component groups from a texture image with the indicated level-of-detail. If *format* is a color format then the components are assigned among R, G, B, and A according to table 6.1, starting with the first group in the first row, and continuing by obtaining groups in order from each row and proceeding from the first row to the last, and from the first image to the last for three-dimensional textures. One- and two-dimensional array textures are treated as two- and three-dimensional images, respectively, where the layers are treated as rows or images. If *format* is `DEPTH_COMPONENT`, then each depth component is assigned with the same ordering of rows and images. If *format* is `DEPTH_STENCIL`, then each depth component and each stencil index is assigned with the same ordering of rows and images.

These groups are then packed and placed in client or pixel buffer object memory. If a pixel pack buffer is bound (as indicated by a non-zero value of `PIXEL_PACK_BUFFER_BINDING`), *img* is an offset into the pixel pack buffer; otherwise, *img* is a pointer to client memory. Pixel storage modes that are applicable to **ReadPixels** are applied.

For three-dimensional and two-dimensional array textures, pixel storage operations are applied as if the image were two-dimensional, except that the additional pixel storage state values `PACK_IMAGE_HEIGHT` and `PACK_SKIP_IMAGES` are applied. The correspondence of texels to memory locations is as defined for **TexImage3D** in section 3.8.1.

The row length, number of rows, image depth, and number of images are determined by the size of the texture image (including any borders). Calling **GetTexImage** with *lod* less than zero or larger than the maximum allowable causes the error `INVALID_VALUE`. Calling **GetTexImage** with a *format* of `STENCIL_INDEX` causes the error `INVALID_ENUM`. Calling **GetTexImage** with a non-zero *lod* when *tex* is `TEXTURE_RECTANGLE` causes the error `INVALID_VALUE`. If a pixel pack buffer object is bound and packing the texture image into the buffer's memory would exceed the size of the buffer, an `INVALID_OPERATION` error results. If a pixel pack buffer object is bound and *img* is not evenly divisible by the number of basic machine units needed to store in memory the GL data type corresponding to *type* (see table 3.2), an `INVALID_OPERATION` error results.

The command

`void` **GetCompressedTexImage**(`enum` *target*, `int` *lod*,
`void` **img*);

is used to obtain texture images stored in compressed form. The parameters *target*, *lod*, and *img* are interpreted in the same manner as in **GetTexImage**. When called, **GetCompressedTexImage** writes n `ubytes` of compressed image data to

Base Internal Format	R	G	B	A
`RED`	R_i	0	0	1
`RG`	R_i	G_i	0	1
`RGB`	R_i	G_i	B_i	1
`RGBA`	R_i	G_i	B_i	A_i

Table 6.1: Texture, table, and filter return values. R_i, G_i, B_i, and A_i are components of the internal format that are assigned to pixel values R, G, B, and A. If a requested pixel value is not present in the internal format, the specified constant value is used.

the pixel pack buffer or client memory pointed to by *img*, where n is the value of `TEXTURE_COMPRESSED_IMAGE_SIZE` for the texture. The compressed image data is formatted according to the definition of the texture's internal format. All pixel storage modes are ignored when returning a compressed texture image.

Calling **GetCompressedTexImage** with an *lod* value less than zero or greater than the maximum allowable causes an `INVALID_VALUE` error. Calling **GetCompressedTexImage** with a texture image stored with an uncompressed internal format causes an `INVALID_OPERATION` error. If a pixel pack buffer object is bound and $img + n$ is greater than the size of the buffer, an `INVALID_OPERATION` error results.

The command

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

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

6.1.5 String Queries

String queries return pointers to UTF-8 encoded, NULL-terminated static strings describing properties of the current GL context [1]. The command

`ubyte` ***GetString**(`enum` *name*);

accepts *name* values of `RENDERER`, `VENDOR`, `VERSION`, and `SHADING_LANGUAGE_VERSION`. The format of the `RENDERER` and `VENDOR` strings is

---

[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.

implementation-dependent. The `VERSION` and `SHADING_LANGUAGE_VERSION` strings are laid out as follows:

<version number><space><vendor-specific information>

The version number is either of the form *major_number.minor_number* or *major_number.minor_number.release_number*, where the numbers all have one or more digits. The *release_number* and vendor specific information are optional. However, if present, then they pertain to the server and their format and contents are implementation-dependent.

GetString returns the version number (in the `VERSION` string) that can be supported by the current GL context. Thus, if the client and server support different versions a compatible version is returned.

The GL version may also be queried by calling **GetIntegerv** with *value*s `MAJOR_VERSION` and `MINOR_VERSION`, which respectively return the same values as *major_number* and *minor_number* in the `VERSION` string, and *value* `CONTEXT_FLAGS`, which returns a set of flags defining additional properties of a context. If `CONTEXT_FLAG_FORWARD_COMPATIBLE_BIT` is set in `CONTEXT_FLAGS`, then the context is a forward-compatible context as defined in appendix E, and the deprecated features described in that appendix are not supported; otherwise the context is a full context, and all features described in the specification are supported.

Indexed strings are queried with the command

`ubyte` ***GetStringi**(`enum` *name*, `uint` *index*);

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. There is no defined relationship between any particular extension name and the $index$ values; an extension name may correspond to a different $index$ in different GL contexts and/or implementations.

An `INVALID_VALUE` error is generated if *index* is outside the valid range for the indexed state *name*.

6.1.6 Asynchronous Queries

The command

`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*.

6.1.7 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 command

`void` **GetBufferParameteriv**(`enum` *target*, `enum` *pname*, `int` **data*);

returns information about a bound buffer object. *target* must be one of the targets listed in table 2.5, and *pname* must be one of the buffer object parameters in table 2.6, 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 2.5. *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 2.5 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 **GetIntegeri_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 **GetIntegeri_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.

6.1.8 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.

6.1.9 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` is returned if *shader* is a vertex shader object, and `FRAGMENT_SHADER` is returned if *shader* is a fragment shader object. If *pname* is `DELETE_STATUS`, `TRUE` is returned if the shader has been flagged for deletion and `FALSE` is returned otherwise. If *pname* is `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.

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.

6.1.10 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`, or `BACK_RIGHT`, identifying a color buffer; `DEPTH`, identifying the depth buffer; or `STENCIL`, identifying the stencil buffer.

If a framebuffer object is bound to *target*, then *attachment* must be one of the attachment points of the framebuffer listed in table 4.11.

If *attachment* is `DEPTH_STENCIL_ATTACHMENT`, and different objects are bound to the depth and stencil attachment points of *target*, the query will fail and generate an `INVALID_OPERATION` error. If the same object is bound to both attachment points, information about that object will be returned.

Upon successful return from **GetFramebufferAttachmentParameteriv**, if *pname* is `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE`, then *param* will contain one of `NONE`, `FRAMEBUFFER_DEFAULT`, `TEXTURE`, or `RENDERBUFFER`, identifying the type of object which contains the attached image. Other values accepted for *pname* depend on the type of object, as described below.

If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is `NONE`, no framebuffer is bound to *target*. In this case querying *pname* `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` will return zero, and all other queries will generate an `INVALID_OPERATION` error.

If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is not `NONE`, these queries apply to all other framebuffer types:

- If *pname* is `FRAMEBUFFER_ATTACHMENT_RED_SIZE`, `FRAMEBUFFER_ATTACHMENT_GREEN_SIZE`, `FRAMEBUFFER_ATTACHMENT_BLUE_SIZE`, `FRAMEBUFFER_ATTACHMENT_ALPHA_SIZE`, `FRAMEBUFFER_ATTACHMENT_DEPTH_SIZE`, or `FRAMEBUFFER_ATTACHMENT_STENCIL_SIZE`, then *param* will contain the number of bits in the corresponding red, green, blue, alpha, depth, or stencil component of the specified *attachment*. Zero is returned if the requested component is not present in *attachment*.

- If *pname* is `FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE`, *param* will contain the format of components of the specified attachment, one of `FLOAT`, `INT`, `UNSIGNED_INT`, `SIGNED_NORMALIZED`, or `UNSIGNED_NORMALIZED` for floating-point, signed integer, unsigned integer, signed normalized fixed-point, or unsigned normalized fixed-point components respectively. Only color buffers may have integer components.

- If *pname* is `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING`, *param* will contain the encoding of components of the specified attachment, one of `LINEAR` or `SRGB` for linear or sRGB-encoded components, respectively. Only color buffer components may be sRGB-encoded; such components are treated as described in sections 4.1.7 and 4.1.8. For the default framebuffer, color encoding is determined by the implementation. For framebuffer objects, components are sRGB-encoded if the internal format of a color attachment is one of the color-renderable SRGB formats described in section 3.8.15.

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 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.

Any combinations of framebuffer type and *pname* not described above will generate an `INVALID_ENUM` error.

6.1.11 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 6.23. If the renderbuffer currently bound to *target* is zero, then an `INVALID_OPERATION` error is generated.

Upon successful return from **GetRenderbufferParameteriv**, if *pname* is `RENDERBUFFER_WIDTH`, `RENDERBUFFER_HEIGHT`, `RENDERBUFFER_INTERNAL_FORMAT`, or `RENDERBUFFER_SAMPLES`, then *params* will contain the width in pixels, height in pixels, internal format, or number of samples, respectively, of the image of the renderbuffer currently bound to *target*.

If *pname* is `RENDERBUFFER_RED_SIZE`, `RENDERBUFFER_GREEN_SIZE`, `RENDERBUFFER_BLUE_SIZE`, `RENDERBUFFER_ALPHA_SIZE`, `RENDERBUFFER_DEPTH_SIZE`, or `RENDERBUFFER_STENCIL_SIZE`, then *params* will contain the actual resolutions (not the resolutions specified when the image array was defined) for the red, green, blue, alpha depth, or stencil components, respectively, of the image of the renderbuffer currently bound to *target*.

Otherwise, an `INVALID_ENUM` error is generated.

In the tables that follow, a type is indicated for each variable. Table 6.2 explains these types. The type actually identifies all state associated with the indicated description; in certain cases only a portion of this state is returned. This is the case with textures, where only the selected texture or texture parameter is returned.

The M and m entries for initial minmax table values represent the maximum and minimum possible representable values, respectively.

6.2 State Tables

The tables on the following pages indicate which state variables are obtained with what commands. State variables that can be obtained using any of **GetBooleanv**,

Type code	Explanation
B	Boolean
BMU	Basic machine units
C	Color (floating-point R, G, B, and A values)
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
S	`NULL`-terminated string
I	Image
Y	Pointer (data type unspecified)
$n \times type$	n copies of type $type$ ($n*$ indicates n is minimum)

Table 6.2: State Variable Types

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.

Get value	Type	Get Command	Initial Value	Description	Sec.
VERTEX_ATTRIB_ARRAY_ENABLED	$16 * \times B$	**GetVertexAttribiv**	`FALSE`	Vertex attrib array enable	2.8
VERTEX_ATTRIB_ARRAY_SIZE	$16 * \times Z$	**GetVertexAttribiv**	4	Vertex attrib array size	2.8
VERTEX_ATTRIB_ARRAY_STRIDE	$16 * \times Z^+$	**GetVertexAttribiv**	0	Vertex attrib array stride	2.8
VERTEX_ATTRIB_ARRAY_TYPE	$16 * \times Z_9$	**GetVertexAttribiv**	`FLOAT`	Vertex attrib array type	2.8
VERTEX_ATTRIB_ARRAY_NORMALIZED	$16 * \times B$	**GetVertexAttribiv**	`FALSE`	Vertex attrib array normalized	2.8
VERTEX_ATTRIB_ARRAY_INTEGER	$16 * \times B$	**GetVertexAttribiv**	`FALSE`	Vertex attrib array has unconverted integers	2.8
VERTEX_ATTRIB_ARRAY_POINTER	$16 * \times Y$	**GetVertex-AttribPointerv**	`NULL`	Vertex attrib array pointer	2.8

Table 6.3. Vertex Array Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.
ELEMENT_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Element array buffer binding	2.9.5
VERTEX_ATTRIB_ARRAY_BUFFER_BINDING	$16 * \times Z^+$	**GetVertexAttribiv**	0	Attribute array buffer binding	2.9

Table 6.4. Vertex Array Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.
ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Current buffer binding	2.9
VERTEX_ARRAY_BINDING	Z^+	**GetIntegerv**	0	Current vertex array object binding	2.10
PRIMITIVE_RESTART	B	**IsEnabled**	`FALSE`	Primitive restart enable	2.8
PRIMITIVE_RESTART_INDEX	Z^+	**GetIntegerv**	0	Primitive restart index	2.8

Table 6.5. Vertex Array Data (not in Vertex Array objects)

Get value	Type	Get Command	Initial Value	Description	Sec.
–	$n \times BMU$	**GetBufferSubData**	-	Buffer data	2.9
BUFFER_SIZE	$n \times Z^+$	**GetBufferParameteriv**	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	$n \times Z^+$	**GetBufferParameteriv**	0	Start of mapped buffer range	2.9
BUFFER_MAP_LENGTH	$n \times Z^+$	**GetBufferParameteriv**	0	Size of mapped buffer range	2.9

Table 6.6. Buffer Object State

Get value	Type	Get Command	Initial Value	Description	Sec.
VIEWPORT	$4 \times Z$	**GetIntegerv**	see 2.12.1	Viewport origin & extent	2.12.1
DEPTH_RANGE	$2 \times R^{+}$	**GetFloatv**	0,1	Depth range near & far	2.12.1
CLIP_DISTANCEi	$6 * \times B$	**IsEnabled**	`FALSE`	ith user clipping plane enabled	2.17

Table 6.7. Transformation state

Get value	Type	Get Command	Initial Value	Description	Sec.
CLAMP_READ_COLOR	Z_3	**GetIntegerv**	`FIXED_ONLY`	Read color clamping	4.3.1

Table 6.8. Coloring

Get value	Type	Get Command	Initial Value	Description	Sec.
POINT_SIZE	R^+	**GetFloatv**	1.0	Point size	3.4
POINT_FADE_THRESHOLD_SIZE	R^+	**GetFloatv**	1.0	Threshold for alpha attenuation	3.4
POINT_SPRITE_COORD_ORIGIN	Z_2	**GetIntegerv**	`UPPER_LEFT`	Origin orientation for point sprites	3.4
LINE_WIDTH	R^+	**GetFloatv**	1.0	Line width	3.5
LINE_SMOOTH	B	**IsEnabled**	`FALSE`	Line antialiasing on	3.5

Table 6.9. Rasterization

Get value	Type	Get Command	Initial Value	Description	Sec.
CULL_FACE	B	**IsEnabled**	`FALSE`	Polygon culling enabled	3.6.1
CULL_FACE_MODE	Z_3	**GetIntegerv**	`BACK`	Cull front-/back-facing polygons	3.6.1
FRONT_FACE	Z_2	**GetIntegerv**	`CCW`	Polygon frontface CW/CCW indicator	3.6.1
POLYGON_SMOOTH	B	**IsEnabled**	`FALSE`	Polygon antialiasing on	3.6
POLYGON_OFFSET_FACTOR	R	**GetFloatv**	0	Polygon offset factor	3.6.4
POLYGON_OFFSET_UNITS	R	**GetFloatv**	0	Polygon offset units	3.6.4
POLYGON_OFFSET_POINT	B	**IsEnabled**	`FALSE`	Polygon offset enable for `POINT` mode rasterization	3.6.4
POLYGON_OFFSET_LINE	B	**IsEnabled**	`FALSE`	Polygon offset enable for `LINE` mode rasterization	3.6.4
POLYGON_OFFSET_FILL	B	**IsEnabled**	`FALSE`	Polygon offset enable for `FILL` mode rasterization	3.6.4

Table 6.10. Rasterization (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.
MULTISAMPLE	B	**IsEnabled**	`TRUE`	Multisample rasterization	3.3.1
SAMPLE_ALPHA_TO_COVERAGE	B	**IsEnabled**	`FALSE`	Modify coverage from alpha	4.1.3
SAMPLE_ALPHA_TO_ONE	B	**IsEnabled**	`FALSE`	Set alpha to maximum	4.1.3
SAMPLE_COVERAGE	B	**IsEnabled**	`FALSE`	Mask to modify coverage	4.1.3
SAMPLE_COVERAGE_VALUE	R^+	**GetFloatv**	1	Coverage mask value	4.1.3
SAMPLE_COVERAGE_INVERT	B	**GetBooleanv**	`FALSE`	Invert coverage mask value	4.1.3

Table 6.11. Multisampling

Get value	Type	Get Command	Initial Value	Description	Sec.
TEXTURE_BINDING_xD	$32 * \times 3 \times Z^+$	**GetIntegerv**	0	Texture object bound to `TEXTURE_`x`D`	3.8.13
TEXTURE_BINDING_1D_ARRAY	$32 * \times Z^+$	**GetIntegerv**	0	Texture object bound to `TEXTURE_1D_ARRAY`	3.8.13
TEXTURE_BINDING_2D_ARRAY	$32 * \times Z^+$	**GetIntegerv**	0	Texture object bound to `TEXTURE_2D_ARRAY`	3.8.13
TEXTURE_BINDING_RECTANGLE	$32 * \times Z^+$	**GetIntegerv**	0	Texture object bound to `TEXTURE_RECTANGLE`	3.8.13
TEXTURE_BINDING_BUFFER	$32 * \times Z^+$	**GetIntegerv**	0	Texture object bound to `TEXTURE_BUFFER`	3.8.13
TEXTURE_BINDING_CUBE_MAP	$32 * \times Z^+$	**GetIntegerv**	0	Texture object bound to `TEXTURE_CUBE_MAP`	3.8.13
TEXTURE_xD	$n \times I$	**GetTexImage**	see 3.8	xD texture image at l.o.d. i	3.8
TEXTURE_1D_ARRAY	$n \times I$	**GetTexImage**	see 3.8	1D texture array image at row i	3.8
TEXTURE_2D_ARRAY	$n \times I$	**GetTexImage**	see 3.8	2D texture array image at slice i	3.8
TEXTURE_RECTANGLE	$n \times I$	**GetTexImage**	see 3.8	Rectangular texture image at l.o.d. zero	3.8
TEXTURE_CUBE_MAP_POSITIVE_X	$n \times I$	**GetTexImage**	see 3.8.1	$+x$ face cube map texture image at l.o.d. i	3.8.1
TEXTURE_CUBE_MAP_NEGATIVE_X	$n \times I$	**GetTexImage**	see 3.8.1	$-x$ face cube map texture image at l.o.d. i	3.8.1
TEXTURE_CUBE_MAP_POSITIVE_Y	$n \times I$	**GetTexImage**	see 3.8.1	$+y$ face cube map texture image at l.o.d. i	3.8.1
TEXTURE_CUBE_MAP_NEGATIVE_Y	$n \times I$	**GetTexImage**	see 3.8.1	$-y$ face cube map texture image at l.o.d. i	3.8.1
TEXTURE_CUBE_MAP_POSITIVE_Z	$n \times I$	**GetTexImage**	see 3.8.1	$+z$ face cube map texture image at l.o.d. i	3.8.1
TEXTURE_CUBE_MAP_NEGATIVE_Z	$n \times I$	**GetTexImage**	see 3.8.1	$-z$ face cube map texture image at l.o.d. i	3.8.1

Table 6.12. Textures (state per texture unit and binding point)

Get value	Type	Get Command	Initial Value	Description	Sec.
TEXTURE_BORDER_COLOR	$n \times C$	**GetTexParameter**	0,0,0,0	Border color	3.8
TEXTURE_MIN_FILTER	$n \times Z_6$	**GetTexParameter**	see sec. 3.8.12	Minification function	3.8.8
TEXTURE_MAG_FILTER	$n \times Z_2$	**GetTexParameter**	`LINEAR`	Magnification function	3.8.9
TEXTURE_WRAP_S	$n \times Z_4$	**GetTexParameter**	see sec. 3.8.12	Texcoord s wrap mode	3.8.8
TEXTURE_WRAP_T	$n \times Z_4$	**GetTexParameter**	see sec. 3.8.12	Texcoord t wrap mode (2D, 3D, cube map textures only)	3.8.8
TEXTURE_WRAP_R	$n \times Z_4$	**GetTexParameter**	see sec. 3.8.12	Texcoord r wrap mode (3D textures only)	3.8.8
TEXTURE_MIN_LOD	$n \times R$	**GetTexParameterfv**	-1000	Minimum level of detail	3.8
TEXTURE_MAX_LOD	$n \times R$	**GetTexParameterfv**	1000	Maximum level of detail	3.8
TEXTURE_BASE_LEVEL	$n \times Z^+$	**GetTexParameterfv**	0	Base texture array	3.8
TEXTURE_MAX_LEVEL	$n \times Z^+$	**GetTexParameterfv**	1000	Max. texture array level	3.8
TEXTURE_LOD_BIAS	$n \times R$	**GetTexParameterfv**	0.0	Texture level of detail bias ($bias_{texobj}$)	3.8.8
TEXTURE_COMPARE_MODE	$n \times Z_2$	**GetTexParameteriv**	`NONE`	Comparison mode	3.8.14
TEXTURE_COMPARE_FUNC	$n \times Z_8$	**GetTexParameteriv**	`LEQUAL`	Comparison function	3.8.14

Table 6.13. Textures (state per texture object)

Get value	Type	Get Command	Initial Value	Description	Sec.
TEXTURE_WIDTH	$n \times Z^+$	**GetTexLevelParameter**	0	Specified width	3.8
TEXTURE_HEIGHT	$n \times Z^+$	**GetTexLevelParameter**	0	Specified height (2D/3D)	3.8
TEXTURE_DEPTH	$n \times Z^+$	**GetTexLevelParameter**	0	Specified depth (3D)	3.8
TEXTURE_INTERNAL_FORMAT	$n \times Z_{68*}$	**GetTexLevelParameter**	`RGBA` or `R8`	Internal format (see section 3.8.12)	3.8
TEXTURE_x_SIZE	$n \times 6 \times Z^+$	**GetTexLevelParameter**	0	Component resolution (x is `RED`, `GREEN`, `BLUE`, `ALPHA`, `DEPTH`, or `STENCIL`)	3.8
TEXTURE_SHARED_SIZE	$n \times Z^+$	**GetTexLevelParameter**	0	Shared exponent field resolution	3.8
TEXTURE_x_TYPE	$n \times Z_5$	**GetTexLevelParameter**	`NONE`	Component type (x is `RED`, `GREEN`, `BLUE`, `ALPHA`, or `DEPTH`)	6.1.3
TEXTURE_COMPRESSED	$n \times B$	**GetTexLevelParameter**	`FALSE`	True if image has a compressed internal format	3.8.3
TEXTURE_COMPRESSED_IMAGE_SIZE	$n \times Z^+$	**GetTexLevelParameter**	0	Size (in `ubytes`) of compressed image	3.8.3
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	3.8.13

Table 6.14. Textures (state per texture image)

Get value	Type	Get Command	Initial Value	Description	Sec.
ACTIVE_TEXTURE	Z_{32*}	**GetIntegerv**	`TEXTURE0`	Active texture unit selector	2.7

Table 6.15. Texture Environment and Generation

Get value	Type	Get Command	Initial Value	Description	Sec.
SCISSOR_TEST	B	**IsEnabled**	`FALSE`	Scissoring enabled	4.1.2
SCISSOR_BOX	$4 \times Z$	**GetIntegerv**	see 4.1.2	Scissor box	4.1.2
STENCIL_TEST	B	**IsEnabled**	`FALSE`	Stenciling enabled	4.1.4
STENCIL_FUNC	Z_8	**GetIntegerv**	`ALWAYS`	Front stencil function	4.1.4
STENCIL_VALUE_MASK	Z^+	**GetIntegerv**	see 4.1.4	Front stencil mask	4.1.4
STENCIL_REF	Z^+	**GetIntegerv**	0	Front stencil reference value	4.1.4
STENCIL_FAIL	Z_8	**GetIntegerv**	`KEEP`	Front stencil fail action	4.1.4
STENCIL_PASS_DEPTH_FAIL	Z_8	**GetIntegerv**	`KEEP`	Front stencil depth buffer fail action	4.1.4
STENCIL_PASS_DEPTH_PASS	Z_8	**GetIntegerv**	`KEEP`	Front stencil depth buffer pass action	4.1.4
STENCIL_BACK_FUNC	Z_8	**GetIntegerv**	`ALWAYS`	Back stencil function	4.1.4
STENCIL_BACK_VALUE_MASK	Z^+	**GetIntegerv**	see 4.1.4	Back stencil mask	4.1.4
STENCIL_BACK_REF	Z^+	**GetIntegerv**	0	Back stencil reference value	4.1.4
STENCIL_BACK_FAIL	Z_8	**GetIntegerv**	`KEEP`	Back stencil fail action	4.1.4
STENCIL_BACK_PASS_DEPTH_FAIL	Z_8	**GetIntegerv**	`KEEP`	Back stencil depth buffer fail action	4.1.4
STENCIL_BACK_PASS_DEPTH_PASS	Z_8	**GetIntegerv**	`KEEP`	Back stencil depth buffer pass action	4.1.4

Table 6.16. Pixel Operations

Get value	Type	Get Command	Initial Value	Description	Sec.
DEPTH_TEST	B	**IsEnabled**	`FALSE`	Depth buffer enabled	4.1.5
DEPTH_FUNC	Z_8	**GetIntegerv**	`LESS`	Depth buffer test function	4.1.5
BLEND	$1 * \times B$	**IsEnabledi**	`FALSE`	Blending enabled for draw buffer i	4.1.7
BLEND_SRC_RGB (v1.3:BLEND_SRC)	Z_{15}	**GetIntegerv**	`ONE`	Blending source RGB function	4.1.7
BLEND_SRC_ALPHA	Z_{15}	**GetIntegerv**	`ONE`	Blending source A function	4.1.7
BLEND_DST_RGB (v1.3:BLEND_DST)	Z_{14}	**GetIntegerv**	`ZERO`	Blending dest. RGB function	4.1.7
BLEND_DST_ALPHA	Z_{14}	**GetIntegerv**	`ZERO`	Blending dest. A function	4.1.7
BLEND_EQUATION_RGB (v1.5: BLEND_EQUATION)	Z_5	**GetIntegerv**	`FUNC_ADD`	RGB blending equation	4.1.7
BLEND_EQUATION_ALPHA	Z_5	**GetIntegerv**	`FUNC_ADD`	Alpha blending equation	4.1.7
BLEND_COLOR	C	**GetFloatv**	0,0,0,0	Constant blend color	4.1.7
FRAMEBUFFER_SRGB	B	**IsEnabled**	`FALSE`	sRGB update and blending enable	4.1.7
DITHER	B	**IsEnabled**	`TRUE`	Dithering enabled	4.1.9
COLOR_LOGIC_OP	B	**IsEnabled**	`FALSE`	Color logic op enabled	4.1.10
LOGIC_OP_MODE	Z_{16}	**GetIntegerv**	`COPY`	Logic op function	4.1.10

Table 6.17. Pixel Operations (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.
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
DEPTH_WRITEMASK	B	**GetBooleanv**	`TRUE`	Depth buffer enabled for writing	4.2.2
STENCIL_WRITEMASK	Z^+	**GetIntegerv**	1's	Front stencil buffer writemask	4.2.2
STENCIL_BACK_WRITEMASK	Z^+	**GetIntegerv**	1's	Back stencil buffer writemask	4.2.2
COLOR_CLEAR_VALUE	C	**GetFloatv**	0,0,0,0	Color buffer clear value	4.2.3
DEPTH_CLEAR_VALUE	R^+	**GetIntegerv**	1	Depth buffer clear value	4.2.3
STENCIL_CLEAR_VALUE	Z^+	**GetIntegerv**	0	Stencil clear value	4.2.3

Table 6.18. Framebuffer Control

Get value	Type	Get Command	Initial Value	Description	Sec.
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 6.19. Framebuffer (state per target binding point)

Get value	Type	Get Command	Initial Value	Description	Sec.
DRAW_BUFFERi	$1 * \times Z_{11*}$	**GetIntegerv**	see 4.2.1	Draw buffer selected for color output i	4.2.1
READ_BUFFER	Z_{11*}	**GetIntegerv**	see 4.3.1	Read source buffer	4.3.1

Table 6.20. Framebuffer (state per framebuffer object)

Get value	Type	Get Command	Initial Value	Description	Sec.
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_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 6.21. Framebuffer (state per attachment point)

Get value	Type	Get Command	Initial Value	Description	Sec.
RENDERBUFFER_BINDING	Z	**GetIntegerv**	0	Renderbuffer object bound to `RENDERBUFFER`	4.4.2

Table 6.22. Renderbuffer (state per target and binding point)

Get value	Type	Get Command	Initial Value	Description	Sec.
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 6.23. Renderbuffer (state per renderbuffer object)

Get value	Type	Get Command	Initial Value	Description	Sec.
UNPACK_SWAP_BYTES	B	**GetBooleanv**	FALSE	Value of UNPACK_SWAP_BYTES	3.7.1
UNPACK_LSB_FIRST	B	**GetBooleanv**	FALSE	Value of UNPACK_LSB_FIRST	3.7.1
UNPACK_IMAGE_HEIGHT	Z^+	**GetIntegerv**	0	Value of UNPACK_IMAGE_HEIGHT	3.7.1
UNPACK_SKIP_IMAGES	Z^+	**GetIntegerv**	0	Value of UNPACK_SKIP_IMAGES	3.7.1
UNPACK_ROW_LENGTH	Z^+	**GetIntegerv**	0	Value of UNPACK_ROW_LENGTH	3.7.1
UNPACK_SKIP_ROWS	Z^+	**GetIntegerv**	0	Value of UNPACK_SKIP_ROWS	3.7.1
UNPACK_SKIP_PIXELS	Z^+	**GetIntegerv**	0	Value of UNPACK_SKIP_PIXELS	3.7.1
UNPACK_ALIGNMENT	Z^+	**GetIntegerv**	4	Value of UNPACK_ALIGNMENT	3.7.1
PACK_SWAP_BYTES	B	**GetBooleanv**	FALSE	Value of PACK_SWAP_BYTES	4.3.1
PACK_LSB_FIRST	B	**GetBooleanv**	FALSE	Value of PACK_LSB_FIRST	4.3.1
PACK_IMAGE_HEIGHT	Z^+	**GetIntegerv**	0	Value of PACK_IMAGE_HEIGHT	4.3.1
PACK_SKIP_IMAGES	Z^+	**GetIntegerv**	0	Value of PACK_SKIP_IMAGES	4.3.1
PACK_ROW_LENGTH	Z^+	**GetIntegerv**	0	Value of PACK_ROW_LENGTH	4.3.1
PACK_SKIP_ROWS	Z^+	**GetIntegerv**	0	Value of PACK_SKIP_ROWS	4.3.1
PACK_SKIP_PIXELS	Z^+	**GetIntegerv**	0	Value of PACK_SKIP_PIXELS	4.3.1
PACK_ALIGNMENT	Z^+	**GetIntegerv**	4	Value of PACK_ALIGNMENT	4.3.1
PIXEL_PACK_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Pixel pack buffer binding	4.3.1
PIXEL_UNPACK_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Pixel unpack buffer binding	6.1.7

Table 6.24. Pixels

Get value	Type	Get Command	Initial Value	Description	Sec.
SHADER_TYPE	Z_2	**GetShaderiv**	-	Type of shader (vertex or fragment)	2.11.1
DELETE_STATUS	B	**GetShaderiv**	`FALSE`	Shader flagged for deletion	2.11.1
COMPILE_STATUS	B	**GetShaderiv**	`FALSE`	Last compile succeeded	2.11.1
-	S	**GetShaderInfoLog**	empty string	Info log for shader objects	6.1.9
INFO_LOG_LENGTH	Z^+	**GetShaderiv**	0	Length of info log	6.1.9
-	S	**GetShaderSource**	empty string	Source code for a shader	2.11.1
SHADER_SOURCE_LENGTH	Z^+	**GetShaderiv**	0	Length of source code	6.1.9

Table 6.25. Shader Object State

Get value	Type	Get Command	Initial Value	Description	Sec.
CURRENT_PROGRAM	Z^+	**GetIntegerv**	0	Name of current program object	2.11.2
DELETE_STATUS	B	**GetProgramiv**	`FALSE`	Program object deleted	2.11.2
LINK_STATUS	B	**GetProgramiv**	`FALSE`	Last link attempt succeeded	2.11.2
VALIDATE_STATUS	B	**GetProgramiv**	`FALSE`	Last validate attempt succeeded	2.11.2
ATTACHED_SHADERS	Z^+	**GetProgramiv**	0	Number of attached shader objects	6.1.9
-	$0 * \times Z^+$	**GetAttachedShaders**	empty	Shader objects attached	6.1.9
-	S	**GetProgramInfoLog**	empty	Info log for program object	6.1.9
INFO_LOG_LENGTH	Z^+	**GetProgramiv**	0	Length of info log	2.11.4
ACTIVE_UNIFORMS	Z^+	**GetProgramiv**	0	Number of active uniforms	2.11.4
-	$0 * \times Z$	**GetUniformLocation**	–	Location of active uniforms	6.1.9
-	$0 * \times Z^+$	**GetActiveUniform**	–	Size of active uniform	2.11.4
-	$0 * \times Z^+$	**GetActiveUniform**	–	Type of active uniform	2.11.4
-	$0 * \times$`char`	**GetActiveUniform**	empty	Name of active uniform	2.11.4
ACTIVE_UNIFORM_MAX_LENGTH	Z^+	**GetProgramiv**	0	Maximum active uniform name length	6.1.9
	$512 * \times R$	**GetUniform**	0	Uniform value	2.11.4
ACTIVE_ATTRIBUTES	Z^+	**GetProgramiv**	0	Number of active attributes	2.11.3

Table 6.26. Program Object State

Get value	Type	Get Command	Initial Value	Description	Sec.
-	$0*\times Z$	**GetAttribLocation**	–	Location of active generic attribute	2.11.3
-	$0*\times Z^+$	**GetActiveAttrib**	–	Size of active attribute	2.11.3
-	$0*\times Z^+$	**GetActiveAttrib**	–	Type of active attribute	2.11.3
-	$0*\times$ `char`	**GetActiveAttrib**	empty	Name of active attribute	2.11.3
ACTIVE_ATTRIBUTE_MAX_LENGTH	Z^+	**GetProgramiv**	0	Maximum active attribute name length	6.1.9
TRANSFORM_FEEDBACK_BUFFER_MODE	Z_2	**GetProgramiv**	`INTERLEAVED_ATTRIBS`	Transform feedback mode for the program	6.1.9
TRANSFORM_FEEDBACK_VARYINGS	Z^+	**GetProgramiv**	0	Number of varyings to stream to buffer object(s)	6.1.9
TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH	Z^+	**GetProgramiv**	0	Maximum transform feedback varying name length	6.1.9
-	Z^+	**GetTransformFeedbackVarying**	-	Size of each transform feedback varying variable	2.11.6
-	Z^+	**GetTransformFeedbackVarying**	-	Type of each transform feedback varying variable	2.11.6
-	$0^+\times$ `char`	**GetTransformFeedbackVarying**	-	Name of each transform feedback varying variable	2.11.6

Table 6.27. Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.
UNIFORM_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Uniform buffer object bound to the context for buffer object manipulation	2.11.4
UNIFORM_BUFFER_BINDING	$n \times Z^+$	**GetIntegeri_v**	0	Uniform buffer object bound to the specified context binding point	2.11.4
ACTIVE_UNIFORM_BLOCKS	Z^+	**GetProgramiv**	0	Number of active uniform blocks in a program	2.11.4
ACTIVE_UNIFORM_BLOCK_MAX_-NAME_LENGTH	Z^+	**GetProgramiv**	0	Length of longest active uniform block name	2.11.4
UNIFORM_TYPE	$0 * \times Z_{27}$	**GetActiveUniformsiv**	-	Type of active uniform	2.11.4
UNIFORM_SIZE	$0 * \times Z^+$	**GetActiveUniformsiv**	-	Size of active uniform	2.11.4
UNIFORM_NAME_LENGTH	$0 * \times Z^+$	**GetActiveUniformsiv**	-	Uniform name length	2.11.4
UNIFORM_BLOCK_INDEX	$0 * \times Z$	**GetActiveUniformsiv**	-	Uniform block index	2.11.4
UNIFORM_OFFSET	$0 * \times Z$	**GetActiveUniformsiv**	-	Uniform buffer offset	2.11.4

Table 6.28. Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.
UNIFORM_ARRAY_STRIDE	$0 * \times Z$	**GetActiveUniformsiv**	-	Uniform buffer array stride	2.11.4
UNIFORM_MATRIX_STRIDE	$0 * \times Z$	**GetActiveUniformsiv**	-	Uniform buffer intra-matrix stride	2.11.4
UNIFORM_IS_ROW_MAJOR	$0 * \times Z^+$	**GetActiveUniformsiv**	-	Whether uniform is a row-major matrix	2.11.4
UNIFORM_BLOCK_BINDING	Z^+	**GetActiveUniformBlockiv**	0	Uniform buffer binding points associated with the specified uniform block	2.11.4
UNIFORM_BLOCK_DATA_SIZE	Z^+	**GetActiveUniformBlockiv**	-	Size of the storage needed to hold this uniform block's data	2.11.4
UNIFORM_BLOCK_ACTIVE_UNIFORMS	Z^+	**GetActiveUniformBlockiv**	-	Count of active uniforms in the specified uniform block	2.11.4
UNIFORM_BLOCK_ACTIVE_UNIFORM_INDICES	$n \times Z^+$	**GetActiveUniformBlockiv**	-	Array of active uniform indices of the specified uniform block	2.11.4
UNIFORM_BLOCK_REFERENCED_BY_VERTEX_SHADER	B	**GetActiveUniformBlockiv**	0	True if uniform block is actively referenced by the vertex stage	2.11.4
UNIFORM_BLOCK_REFERENCED_BY_FRAGMENT_SHADER	B	**GetActiveUniformBlockiv**	0	True if uniform block is actively referenced by the fragment stage	2.11.4

Table 6.29. Program Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.
CURRENT_VERTEX_ATTRIB	$16 * \times R^4$	**GetVertexAttribfv**	0.0,0.0,0.0,1.0	Current generic vertex attribute values	2.7
VERTEX_PROGRAM_POINT_SIZE	B	**IsEnabled**	`FALSE`	Point size mode	3.4

Table 6.30. Vertex Shader State

Get value	Type	Get Command	Initial Value	Description	Sec.
QUERY_RESULT	Z^+	**GetQueryObjectuiv**	0	Query object result	6.1.6
QUERY_RESULT_AVAILABLE	B	**GetQueryObjectiv**	`FALSE`	Is the query object result available?	6.1.6

Table 6.31. Query Object State

Get value	Type	Get Command	Initial Value	Description	Sec.
TRANSFORM_FEEDBACK_BUFFER_BINDING	Z^+	**GetIntegerv**	0	Buffer object bound to generic bind point for transform feedback	6.1.7
TRANSFORM_FEEDBACK_BUFFER_BINDING	nxZ^+	**GetIntegeri_v**	0	Buffer object bound to each transform feedback attribute stream	6.1.7
TRANSFORM_FEEDBACK_BUFFER_START	nxZ^+	**GetIntegeri_v**	0	Start offset of binding range for each transform feedback attrib. stream	6.1.7
TRANSFORM_FEEDBACK_BUFFER_SIZE	$n \times Z^+$	**GetIntegeri_v**	0	Size of binding range for each transform feedback attrib. stream	6.1.7
MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS	Z^+	**GetIntegerv**	64	Max number of components to write to a single buffer in interleaved mode	2.15
MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS	Z^+	**GetIntegerv**	4	Max number of separate attributes or vayings that can be captured in transform feedback	2.15
MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS	Z^+	**GetIntegerv**	4	Max number of components per attribute or varying in separate mode	2.15

Table 6.32. Transform Feedback State

Get value	Type	Get Command	Initial Value	Description	Sec.
LINE_SMOOTH_HINT	Z_3	**GetIntegerv**	`DONT_CARE`	Line smooth hint	5.2
POLYGON_SMOOTH_HINT	Z_3	**GetIntegerv**	`DONT_CARE`	Polygon smooth hint	5.2
TEXTURE_COMPRESSION_HINT	Z_3	**GetIntegerv**	`DONT_CARE`	Texture compression quality hint	5.2
FRAGMENT_SHADER_DERIVATIVE_HINT	Z_3	**GetIntegerv**	`DONT_CARE`	Fragment shader derivative accuracy hint	5.2

Table 6.33. Hints

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_CLIP_DISTANCES	Z^+	**GetIntegerv**	8	Maximum number of user clipping planes	2.17
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	3.8.1
MAX_TEXTURE_SIZE	Z^+	**GetIntegerv**	1024	Maximum 2D/1D texture image dimension	3.8.1
MAX_ARRAY_TEXTURE_LAYERS	Z^+	**GetIntegerv**	256	Maximum number of layers for texture arrays	3.8.1
MAX_TEXTURE_LOD_BIAS	R^+	**GetFloatv**	2.0	Maximum absolute texture level of detail bias	3.8.8
MAX_CUBE_MAP_TEXTURE_SIZE	Z^+	**GetIntegerv**	1024	Maximum cube map texture image dimension	3.8.1
MAX_RENDERBUFFER_SIZE	Z^+	**GetIntegerv**	1024	Maximum width and height of renderbuffers	4.4.2

Table 6.34. Implementation Dependent Values

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_VIEWPORT_DIMS	$2 \times Z^+$	**GetIntegerv**	see 2.12.1	Maximum viewport dimensions	2.12.1
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 6.35. Implementation Dependent Values (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.
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_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	3.8.3
NUM_COMPRESSED_TEXTURE_FORMATS	Z	**GetIntegerv**	4	Number of compressed texture formats	3.8.3
MAX_TEXTURE_BUFFER_SIZE	Z^+	**GetIntegerv**	65536	No. of addressable texels for buffer textures	3.8.4
MAX_RECTANGLE_TEXTURE_SIZE	Z^+	**GetIntegerv**	1024	Max. width & height of rectangular textures	3.8.1

Table 6.36. Implementation Dependent Values (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.
QUERY_COUNTER_BITS	$3 \times Z^+$	**GetQueryiv**	see 6.1.6	Asynchronous query counter bits	6.1.6
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**	–	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
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 6.37. Implementation Dependent Values (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.
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	2.11.4
MAX_VARYING_COMPONENTS	Z^+	**GetIntegerv**	64	Number of components for varying variables	2.11.6
MAX_COMBINED_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	32	Total number of texture units accessible by the GL	2.11.7
MAX_VERTEX_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	16	Number of texture image units accessible by a vertex shader	2.11.7
MAX_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	16	Number of texture image units accessible by fragment processing	2.11.7
MAX_FRAGMENT_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1024	Number of components for frag. shader uniform variables	3.9.1
MIN_PROGRAM_TEXEL_OFFSET	Z	**GetIntegerv**	-8	Minimum texel offset allowed in lookup	2.11.7
MAX_PROGRAM_TEXEL_OFFSET	Z	**GetIntegerv**	7	Maximum texel offset allowed in lookup	2.11.7

Table 6.38. Implementation Dependent Values (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_VERTEX_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	12	Max number of vertex uniform buffers per program	2.11.4
MAX_FRAGMENT_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	12	Max number of fragment uniform buffers per program	2.11.4
MAX_COMBINED_UNIFORM_BLOCKS	Z^+	**GetIntegerv**	24	Max number of uniform buffers per program	2.11.4
MAX_UNIFORM_BUFFER_BINDINGS	Z^+	**GetIntegerv**	24	Max number of uniform buffer binding points on the context	2.11.4
MAX_UNIFORM_BLOCK_SIZE	Z^+	**GetIntegerv**	16384	Max size in basic machine units of a uniform block	2.11.4
UNIFORM_BUFFER_OFFSET_ALIGNMENT	Z^+	**GetIntegerv**	1	Minimum required alignment for uniform buffer sizes and offsets	2.11.4

Table 6.39. Implementation Dependent Values (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.
MAX_VERTEX_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1	Number of words for vertex shader uniform variables in default uniform block	2.11.4
MAX_FRAGMENT_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1	Number of words for fragment shader uniform variables in default uniform block	2.11.4
MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1	Number of words for vertex shader uniform variables in all uniform blocks (including default)	2.11.4
MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS	Z^+	**GetIntegerv**	1	Number of words for fragment shader uniform variables in all uniform blocks (including default)	2.11.4

Table 6.40. Implementation Dependent Values (cont.)
(1) 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.
MAX_DRAW_BUFFERS	Z^+	**GetIntegerv**	8	Maximum number of active draw buffers	4.2.1
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
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

Table 6.41. Framebuffer Dependent Values

Get value	Type	Get Command	Initial Value	Description	Sec.
–	$n \times Z_5$	**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	4.1.6
CURRENT_QUERY	$3 \times Z^+$	**GetQueryiv**	0	Active query object names	6.1.6
COPY_READ_BUFFER	Z^+	**GetIntegerv**	0	Buffer object bound to copy buffer “read” bind point	2.9.3
COPY_WRITE_BUFFER	Z^+	**GetIntegerv**	0	Buffer object bound to copy buffer “write” bind point	2.9.3
TEXTURE_BUFFER	Z^+	**GetIntegerv**	0	Buffer object bound to texture buffer bind point	3.8.13

Table 6.42. Miscellaneous

Appendix A

Invariance

The OpenGL specification is not pixel exact. It therefore does not guarantee an exact match between images produced by different GL implementations. However, the specification does specify exact matches, in some cases, for images produced by the same implementation. The purpose of this appendix is to identify and provide justification for those cases that require exact matches.

A.1 Repeatability

The obvious and most fundamental case is repeated issuance of a series of GL commands. For any given GL and framebuffer state *vector*, and for any GL command, the resulting GL and framebuffer state must be identical whenever the command is executed on that initial GL and framebuffer state.

One purpose of repeatability is avoidance of visual artifacts when a double-buffered scene is redrawn. If rendering is not repeatable, swapping between two buffers rendered with the same command sequence may result in visible changes in the image. Such false motion is distracting to the viewer. Another reason for repeatability is testability.

Repeatability, while important, is a weak requirement. Given only repeatability as a requirement, two scenes rendered with one (small) polygon changed in position might differ at every pixel. Such a difference, while within the law of repeatability, is certainly not within its spirit. Additional invariance rules are desirable to ensure useful operation.

A.2 Multi-pass Algorithms

Invariance is necessary for a whole set of useful multi-pass algorithms. Such algorithms render multiple times, each time with a different GL mode vector, to eventually produce a result in the framebuffer. Examples of these algorithms include:

- "Erasing" a primitive from the framebuffer by redrawing it, either in a different color or using the XOR logical operation.
- Using stencil operations to compute capping planes.

On the other hand, invariance rules can greatly increase the complexity of high-performance implementations of the GL. Even the weak repeatability requirement significantly constrains a parallel implementation of the GL. Because GL implementations are required to implement ALL GL capabilities, not just a convenient subset, those that utilize hardware acceleration are expected to alternate between hardware and software modules based on the current GL mode vector. A strong invariance requirement forces the behavior of the hardware and software modules to be identical, something that may be very difficult to achieve (for example, if the hardware does floating-point operations with different precision than the software).

What is desired is a compromise that results in many compliant, high-performance implementations, and in many software vendors choosing to port to OpenGL.

A.3 Invariance Rules

For a given instantiation of an OpenGL rendering context:

Rule 1 *For any given GL and framebuffer state vector, and for any given GL command, the resulting GL and framebuffer state must be identical each time the command is executed on that initial GL and framebuffer state.*

Rule 2 *Changes to the following state values have no side effects (the use of any other state value is not affected by the change):*

Required:

- *Framebuffer contents (all bitplanes)*
- *The color buffers enabled for writing*
- *Scissor parameters (other than enable)*

- *Writemasks (color, depth, stencil)*
- *Clear values (color, depth, stencil)*

Strongly suggested:

- *Stencil parameters (other than enable)*
- *Depth test parameters (other than enable)*
- *Blend parameters (other than enable)*
- *Logical operation parameters (other than enable)*
- *Pixel storage state*
- *Polygon offset parameters (other than enables, and except as they affect the depth values of fragments)*

Corollary 1 *Fragment generation is invariant with respect to the state values marked with • in Rule 2.*

Rule 3 *The arithmetic of each per-fragment operation is invariant except with respect to parameters that directly control it.*

Corollary 2 *Images rendered into different color buffers sharing the same framebuffer, either simultaneously or separately using the same command sequence, are pixel identical.*

Rule 4 *The same vertex or fragment shader will produce the same result when run multiple times with the same input. The wording 'the same shader' means a program object that is populated with the same source strings, which are compiled and then linked, possibly multiple times, and which program object is then executed using the same GL state vector.*

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 error semantics of upward compatible OpenGL revisions may change, and features deprecated in a previous revision may be removed. Otherwise, only additions can be made to upward compatible revisions.
2. GL query commands are not required to satisfy the semantics of the **Flush** or the **Finish** commands. All that is required is that the queried state be consistent with complete execution of all previously executed GL commands.
3. Application specified point size and line width must be returned as specified when queried. Implementation-dependent clamping affects the values only while they are in use.
4. The mask specified as the third argument to **StencilFunc** affects the operands of the stencil comparison function, but has no direct effect on the update of the stencil buffer. The mask specified by **StencilMask** has no effect on the stencil comparison function; it limits the effect of the update of the stencil buffer.
5. 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.
6. There is no atomicity requirement for OpenGL rendering commands, even at the fragment level.

7. 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.

8. OpenGL does not force left- or right-handedness on any of its coordinates systems.

9. (No pixel dropouts or duplicates.) Let two polygons share an identical edge (that is, there exist vertices A and B of an edge of one polygon, and vertices C and D of an edge of the other polygon, and the coordinates of vertex A (resp. B) are identical to those of vertex C (resp. D), and the state of the the coordinate transfomations is identical when A, B, C, and D are specified). Then, when the fragments produced by rasterization of both polygons are taken together, each fragment intersecting the interior of the shared edge is produced exactly once.

10. Dithering algorithms may be different for different components. In particular, alpha may be dithered differently from red, green, or blue, and an implementation may choose to not dither alpha at all.

Appendix C

Compressed Texture Image Formats

C.1 RGTC Compressed Texture Image Formats

Compressed texture images stored using the RGTC compressed image encodings are represented as a collection of 4×4 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

$bits$ 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:

$$code(x,y) = bits\,[3 \times (4 \times y + x) + 2 \ldots 3 \times (4 \times y + x) + 0]$$

where bit 47 is the most significant and bit 0 is the least significant bit.

The red value R for a texel at location (x, y) in the block is given by:

$$R = \begin{cases} RED_0, & red_0 > red_1, code(x,y) = 0 \\ RED_1, & red_0 > red_1, code(x,y) = 1 \\ \frac{6RED_0 + RED_1}{7}, & red_0 > red_1, code(x,y) = 2 \\ \frac{5RED_0 + 2RED_1}{7}, & red_0 > red_1, code(x,y) = 3 \\ \frac{4RED_0 + 3RED_1}{7}, & red_0 > red_1, code(x,y) = 4 \\ \frac{3RED_0 + 4RED_1}{7}, & red_0 > red_1, code(x,y) = 5 \\ \frac{2RED_0 + 5RED_1}{7}, & red_0 > red_1, code(x,y) = 6 \\ \frac{RED_0 + 6RED_1}{7}, & red_0 > red_1, code(x,y) = 7 \\ RED_0, & red_0 \leq red_1, code(x,y) = 0 \\ RED_1, & red_0 \leq red_1, code(x,y) = 1 \\ \frac{4RED_0 + RED_1}{5}, & red_0 \leq red_1, code(x,y) = 2 \\ \frac{3RED_0 + 2RED_1}{5}, & red_0 \leq red_1, code(x,y) = 3 \\ \frac{2RED_0 + 3RED_1}{5}, & red_0 \leq red_1, code(x,y) = 4 \\ \frac{RED_0 + 4RED_1}{5}, & red_0 \leq red_1, code(x,y) = 5 \\ RED_{min}, & red_0 \leq red_1, code(x,y) = 6 \\ RED_{max}, & red_0 \leq red_1, code(x,y) = 7 \end{cases}$$

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 (two's 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

State that can be shared between contexts includes pixel and vertex buffer objects, program and shader objects, and texture objects (except for the texture objects named zero).

Framebuffer, query, and vertex array objects are not shared.

D.1 Object Deletion Behavior

After an object is deleted, its name is immediately marked unused. Caution should be taken when deleting an object attached to a container object (such as a buffer object attached to a vertex array object, or a renderbuffer or texture attached to a framebuffer object), or a shared object bound in multiple contexts. Following its deletion, the object's name may be returned by **Gen*** commands, even though the underlying object state and data may still be referred to by container objects, or in use by contexts other than the one in which the object was deleted. Such a container or other context may continue using the object, and may still contain state identifying its name as being currently bound, until such time as the container object is deleted, the attachment point of the container object is changed to refer to another object, or another attempt to bind or attach the name is made in that context. Since the name is marked unused, binding the name will create a new object with the same name, and attaching the name will generate an error. The underlying storage backing a deleted object will not be reclaimed by the GL until all references to the object from container object attachment points or context binding points are removed.

D.2 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**.

The object T is considered to have been changed once such a command has completed. Completion of a command [1] may be determined only by calling **Finish**.

---

[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.2.1 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.2.2 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 *must be completed in that other context (by calling* **Finish***) prior to* C *being bound. Changes made in another* *context without calling* **Finish***, or after* C *is bound in the current context, are not guaranteed to be seen.*

Rule 3 *State changes to shared objects are not automatically propagated between contexts. If the state of a shared object* T *is changed in a context other than the current context, and* T *is already directly or indirectly attached to the current context, any operations on the current context involving* T *via those attachments are not guaranteed to use its new state.*

Rule 4 *If the state of a shared object* T *is changed in a context other than the current context, and* T *is already directly or indirectly attached to the current context at multiple attachment or bind points, it must be attached or re-attached to at least one binding point in the current context in order for the new state of* T *to be visible in the current context.*

Note: "Attached or re-attached" *means either attaching an object to a binding point it wasn't already attached to, or attaching an object again to a binding point it was already attached.*

Note: *This rule also applies to the pointer to the data store of an object. The pointer itself is state, while the content of the data store are data, not state. To guarantee that another context sees data updates to an object, you should attach or re-attach the object in that context, since the pointer to the data store could have changed.*

Note: *To be sure that a data update, as the result of a transform-feedback operation in another context, is visible in the current context, the app needs to make sure that the command* **EndTransformFeedback** *has completed (using* **Finish***).*

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

The Deprecation Model

OpenGL 3.0 introduces a deprecation model in which certain features may be marked as *deprecated*. Deprecated features are expected to be completely removed from a future version of OpenGL. Deprecated features are summarized in section E.1.

To aid developers in writing applications which will run on such future versions, it is possible to create an OpenGL 3.0 context which does not support deprecated features. Such a context is called a *forward compatible* context, while a context supporting all OpenGL 3.0 features is called a *full* context. Forward compatible contexts cannot restore deprecated functionality through extensions, but they may support additional, non-deprecated functionality through extensions.

Profiles allow defining subsets of OpenGL functionality targeted to specific application domains. While OpenGL 3.0 only defines a single profile, future versions may introduce profiles addressing domains such as workstation, gaming, and embedded. Implementations are not required to support all defined profiles, but must support at least one profile.

To enable application control of deprecation and profiles, new *context creation APIs* have been defined as extensions to GLX and WGL. These APIs allow specifying a particular version, profile, and full or forward compatible status, and will either create a context compatible with the request, or fail (if, for example, requesting an OpenGL version or profile not supported by the implementation),

Only the ARB may define OpenGL profiles and deprecated features.

E.1 Profiles and Deprecated Features of OpenGL 3.0

OpenGL 3.0 defines a single profile, and all OpenGL 3.0 implementations must support that profile.

The features deprecated in OpenGL 3.0 are summarized below, together with the sections of the specification in which they are defined. Functions which are completely deprecated will generate an `INVALID_OPERATION` error if called in a forward-compatible context. Functions which are partially deprecated (e.g. no longer accept some parameter values) will generate the errors appropriate for any other unrecognized value of that parameter when a deprecated value is passed in a forward-compatible context.

- Application-generated object names - the names of all object types, such as buffer, query, and texture objects, must be generated using the corresponding **Gen*** commands. Trying to bind an object name not returned by a **Gen*** command will result in an `INVALID_OPERATION` error. This behavior is already the case for framebuffer, renderbuffer, and vertex array objects. Object types which have default objects (objects named zero) , such as vertex array, framebuffer, and texture objects, may also bind the default object, even though it is not returned by **Gen***.

- Color index mode - No color index visuals are supplied by the window system-binding APIs such as GLX and WGL, so the default framebuffer is always in RGBA mode. All language and state related to color index mode vertex, rasterization, and fragment processing behavior is removed. `COLOR_INDEX` formats are also deprecated.

- OpenGL Shading Language versions 1.10 and 1.20. These versions of the shading language depend on many API features that have also been deprecated.

- **Begin / End** primitive specification - **Begin**, **End**, and **EdgeFlag***; **Color***, **FogCoord***, **Index***, **Normal3***, **SecondaryColor3***, **TexCoord***, **Vertex*** **Vertex***; and all associated state. Vertex arrays and array drawing commands must be used to draw primitives. However, **VertexAttrib*** and the current vertex attribute state are retained in order to provide default attribute values for disabled attribute arrays.

- Edge flags and fixed-function vertex processing - **ColorPointer**, **EdgeFlagPointer**, **FogCoordPointer**, **IndexPointer**, **NormalPointer**, **SecondaryColorPointer**, **TexCoordPointer**, **VertexPointer**, **EnableClientState**, **DisableClientState**, and **InterleavedArrays**, **ClientActiveTexture**; **Frustum**, **LoadIdentity**, **LoadMatrix**, **LoadTransposeMatrix**, **MatrixMode**, **MultMatrix**, **MultTransposeMatrix**, **Ortho**, **PopMatrix**, **PushMatrix**, **Rotate**, **Scale**, and **Translate**; **Enable/Disable** targets `RESCALE_NORMAL` and `NORMALIZE`; **TexGen*** and **Enable/Disable** targets `TEXTURE_GEN_*`, **Material***, **Light***, **LightModel***, and **ColorMaterial**, **ShadeModel**, and **Enable/Disable** targets `LIGHTING`. `VERTEX_PROGRAM_TWO_SIDE`, `LIGHT`*i*, and `COLOR_MATERIAL`; **ClipPlane**; and all associated fixed-function vertex array, multitexture, matrix and matrix stack, normal and texture coordinate, lighting, and clipping state. A vertex shader must be defined in order to draw primitives.

Language referring to edge flags in the current specification is modified as though all edge flags are `TRUE`.

Note that the **FrontFace** and **ClampColor** commands are **not** deprecated, as they still affect other non-deprecated functionality; however, the **ClampColor** *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. The default vertex array object (the name zero) is also deprecated. Calling **VertexAttribPointer** when no buffer object or no vertex array object is bound will generate an `INVALID_OPERATION` error, as will calling any array drawing command when no vertex array object is bound.
- Rectangles - **Rect***.
- Current raster position - **RasterPos*** and **WindowPos***, and all associated state.
- Two-sided color selection - **Enable** target `VERTEX_PROGRAM_TWO_SIDE`; OpenGL Shading Language builtins `gl_BackColor` and `gl_BackSecondaryColor`; and all associated state.
- Non-sprite points - **Enable/Disable** targets `POINT_SMOOTH` and `POINT_SPRITE`, and all associated state. Point rasterization is always performed as though `POINT_SPRITE` were enabled.
- Wide lines and line stipple - **LineWidth** is not deprecated, but values greater than 1.0 will generate an `INVALID_VALUE` error; **LineStipple** and **Enable/Disable** target `LINE_STIPPLE`, and all associated state.
- Quadrilateral and polygon primitives - vertex array drawing modes `POLYGON`, `QUADS`, and `QUAD_STRIP`, related descriptions of rasterization of non-triangle polygons, and all associated state.

- Separate polygon draw mode - **PolygonMode** *face* values of `FRONT` and `BACK`; polygons are always drawn in the same mode, no matter which face is being rasterized.

- Polygon Stipple - **PolygonStipple** and **Enable/Disable** target `POLYGON_STIPPLE`, and all associated state.

- Pixel transfer modes and operations - all pixel transfer modes, including pixel maps, shift and bias, color table lookup, color matrix, and convolution commands and state, and all associated state and commands defining that state.

- Pixel drawing - **DrawPixels** and **PixelZoom**. However, the language describing pixel rectangles in section 3.7 is retained as it is required for **TexImage*** and **ReadPixels**.

- Bitmaps - **Bitmap** and the `BITMAP` external format.

- Legacy OpenGL 1.0 pixel formats - the values 1, 2, 3, and 4 are no longer accepted as internal formats by **TexImage*** or any other command taking an internal format argument. The initial internal format of a texel array is `RGBA` instead of 1. `TEXTURE_COMPONENTS` is deprecated; always use `TEXTURE_INTERNAL_FORMAT`.

- Legacy pixel formats - all `ALPHA`, `LUMINANCE`, `LUMINANCE_ALPHA`, and `INTENSITY` external and internal formats, including compressed, floating-point, and integer variants; all references to luminance and intensity formats elsewhere in the specification, including conversion to and from those formats; and all associated state. including state describing the allocation or format of luminance and intensity texture or framebuffer components.

- Depth texture mode - `DEPTH_TEXTURE_MODE`. Section 3.8.14 is to be changed so that r is returned to texture samplers directly, and the OpenGL Shading Language 1.30 Specification is to be changed so that $(r, r, r, 1)$ is always returned from depth texture samplers in this case.

- Texture wrap mode `CLAMP` - `CLAMP` is no longer accepted as a value of texture parameters `TEXTURE_WRAP_S`, `TEXTURE_WRAP_T`, or `TEXTURE_WRAP_R`.

- Texture borders - the *border* value to **TexImage*** must always be zero, or an `INVALID_VALUE` error is generated (section 3.8.1); all language in section 3.8 referring to nonzero border widths during texture image specification and texture sampling; and all associated state.

- Automatic mipmap generation - **TexParameter*** *target* `GENERATE_MIPMAP`, and all associated state.

- Fixed-function fragment processing - **AreTexturesResident**, **PrioritizeTextures**, and **TexParameter** *target* `TEXTURE_PRIORITY`; **TexEnv** *target* `TEXTURE_ENV`, and all associated parameters; **TexEnv** *target* `TEXTURE_FILTER_CONTROL`, and parameter name `TEXTURE_LOD_BIAS`; **Enable** *targets* of all dimensionalities (`TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`, and `TEXTURE_CUBE_MAP`); **Enable** target `COLOR_SUM`; **Enable** *target* `FOG`, **Fog**, and all associated parameters; the implementation-dependent values `MAX_TEXTURE_UNITS` and `MAX_TEXTURE_COORDS`; and all associated state.

- Alpha test - **AlphaFunc** and **Enable/Disable** target `ALPHA_TEST`, and all associated state.

- Accumulation buffers - **ClearAccum**, and `ACCUM_BUFFER_BIT` is not valid as a bit in the argument to **Clear** (section 4.2.3); **Accum**; the `ACCUM_*_BITS` framebuffer state describing the size of accumulation buffer components; and all associated state.

  Window system-binding APIs such as GLX and WGL may choose to either not expose window configs containing accumulation buffers, or to ignore accumulation buffers when the default framebuffer bound to a GL context contains them.

- Pixel copying - **CopyPixels** (the comments also applying to **CopyTexImage** will be moved to section 3.8.2).

- Auxiliary color buffers, including `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, and all associated state.

- Selection and feedback modes - **RenderMode**, **InitNames**, **PopName**, **PushName**, **LoadName**, and **SelectBuffer**; **FeedbackBuffer** and **PassThrough**; and all associated state.

- Display lists - **NewList**, **EndList**, **CallList**, **CallLists**, **ListBase**, **GenLists**, **IsList**, and **DeleteLists**; all references to display lists and behavior when compiling commands into display lists elsewhere in the specification; and all associated state.

- Hints - the `PERSPECTIVE_CORRECTION_HINT`, `POINT_SMOOTH_HINT`, `FOG_HINT`, and `GENERATE_MIPMAP_HINT` targets to **Hint** (section 5.2).
- Attribute stacks - **PushAttrib**, **PushClientAttrib**, **PopAttrib**, **PopClientAttrib**, the `MAX_ATTRIB_STACK_DEPTH`, `MAX_CLIENT_ATTRIB_STACK_DEPTH`, `ATTRIB_STACK_DEPTH`, and `CLIENT_ATTRIB_STACK_DEPTH` state, the client and server attribute stacks, and the values `ALL_ATTRIB_BITS` and `CLIENT_ALL_ATTRIB_BITS`.
- Unified extension string - `EXTENSIONS` target to **GetString**.
- Token names and queries - all token names and queries not otherwise mentioned above for deprecated state, as well as all query entry points where all valid targets of that query are deprecated state (chapter 6 and the state tables)

Appendix F

Version 3.0 and Before

OpenGL version 3.0, released on August 11, 2008, is the eighth revision since the original version 1.0. When using a *full* 3.0 context, OpenGL 3.0 is upward compatible with earlier versions, meaning that any program that runs with a 2.1 or earlier GL implementation will also run unchanged with a 3.0 GL implementation. OpenGL 3.0 context creation is done using a window system binding API, and on most platforms a new command, defined by extensions introduced along with OpenGL 3.0, must be called to create a 3.0 context. Calling the older context creation commands will return an OpenGL 2.1 context. When using a *forward compatible* context, many OpenGL 2.1 features are not supported.

Following are brief descriptions of changes and additions to OpenGL 3.0. Descriptions of changes and additions in earlier versions of OpenGL (versions 1.1, 1.2, 1.2.1, 1.3, 1.4, 1.5, 2.0, and 2.1) are omitted in this specification, but may be found in the OpenGL 3.0 Specification, available on the World Wide Web at URL

http://www.opengl.org/registry/

F.1 New Features

New features in OpenGL 3.0, including the extension or extensions if any on which they were based, include:

- API support for the new texture lookup, texture format, and integer and unsigned integer capabilities of the OpenGL Shading Language 1.30 specification (`GL_EXT_gpu_shader4`).
- Conditional rendering (`GL_NV_conditional_render`).

- 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 6.1.10 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 2.13 that **BeginQuery** will generate an `INVALID_OPERATION` error when called with an existing query object name whose type does not match the specified *target* (bug 3712).
- Add description of `gl_ClipDistance` to shader outputs in section 2.11.7 and note that only one of `gl_ClipVertex` and `gl_ClipDistance` should be written by a shader (bug 3898).

- Changed **ClearBuffer*** in section 4.2.3 to indirect through the draw buffer state by specifying the buffer type and draw buffer number, rather than the attachment name; also changed to accept `DEPTH_BUFFER` / `DEPTH_ATTACHMENT` and `STENCIL_BUFFER` / `STENCIL_ATTACHMENT` interchangeably, to reduce inconsistency between clearing the default framebuffer and framebuffer objects. Likewise changed **GetFramebufferAttachmentParameteriv** in section 6.1.10 to accept `DEPTH_BUFFER` / `DEPTH_ATTACHMENT` and `STENCIL_BUFFER` / `STENCIL_ATTACMENT` interchangeably (bug 3744).

- Add proper type suffix to query commands in tables 6.3 and 6.30 (Mark Kilgard).

- Update deprecation list in section E.1 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, 2.9.1; tables 2.6, 2.7, and 6.6 - 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 2.12.1, 4.1.2, 4.2.1, and 4.3.3 - define initial state when a context is bound with no default framebuffer - null viewport and scissor region, draw buffer = read buffer = `NONE`, max viewport dims = max(display size - if any, max renderbuffer size). Viewport/scissor language added to the GLX and WGL create context extension specs as well (bug 2941).

- Section 2.15 - define “word-aligned” to be a multiple of 4 (e.g. 32 bits) (bug 3624).

- Section 6.1.7 - 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 6.1.10 - 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 6.1.11 - 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.1 - 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 2.17 and table 6.34, to match GLSL (bug 4803).
- Accept null pointers in **CompressedTexImage*** (section 3.8.3) and treat them the same as for the corresponding **TexImage*** commands (bug 4863).

- Relax error conditions when specifying RGTC format texture images (section 3.8.2) and subimages (section 3.8.3) so that non-power-of-two RGTC images may be specified (also see section C.1), and edits to partial tiles at the edge of such an image made (bug 4856).
- Relaxed texture magnification switch-over point calculation in section 3.8.9 (bug 4392).
- Clarify initial value of stencil value masks in section 4.1.4 and table 6.16 (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 6.21 to `NONE` (bug 4407).
- Brought extension list in appendix H.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

Extension Registry, Header Files, and ARB Extensions

H.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/

H.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 H.1). The combination of `<GL/gl.h>` and `<GL/glext.h>` always defines APIs for the latest core OpenGL version as well as for all extensions defined in the Registry.

With the introduction of OpenGL 3.1, many features were removed from the core API. The deprecation model does not allow reintroduction of these features except via the special `GL_ARB_compatibility` extension (see section G.2). While it is possible to continue using `<GL/gl.h>` and `<GL/glext.h>`, new header files are defined for OpenGL 3.1 and future versions. The ARB recommends using these headers for any new application which is written to the OpenGL 3.1 core without using any of the features removed from OpenGL 3.0. The header files so defined are:

- `<GL3/gl3.h>`, which will always define the core API of the current version of OpenGL, and only that API. It does not include APIs for features removed by OpenGL 3.1. Initially it contains only the APIs in OpenGL 3.1.
- `<GL3/gl3ext.h>`, which will always define APIs for registered extensions which may be provided by an OpenGL 3.1 implementation that does not support the `GL_ARB_compatibility` extension. Most currently defined extensions cannot be provided by such an implementation, since they depend on features no longer present in 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 OpenGL 3.1. This can assist in developing applications on a GL implementation that supports `GL_ARB_compatibility` when the application is also intended to run on other platforms supporting only the core OpenGL 3.1 API.

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.

H.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.

H.3.1 Naming Conventions

To distinguish ARB extensions from core OpenGL features and from vendor-specific extensions, the following naming conventions are used:

- A unique *name string* of the form ”`GL_ARB_`*`name`*” is associated with each extension. If the extension is supported by an implementation, this string will be among the `EXTENSIONS` strings returned by **GetStringi**, as described in section 6.1.4.
- All functions defined by the extension will have names of the form ***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.

H.3.2 Promoting Extensions to Core Features

ARB extensions can be *promoted* to required core features in later revisions of OpenGL. When this occurs, the extension specifications are merged into the core specification. Functions and enumerants that are part of such promoted extensions will have the **ARB** affix removed.

GL implementations of such later revisions should continue to export the name strings of promoted extensions in the `EXTENSIONS` strings and continue to support the **ARB**-affixed versions of functions and enumerants as a transition aid.

For descriptions of extensions promoted to core features in OpenGL 1.3 and beyond, see the corresponding version of the OpenGL specification, or the descriptions of that version in version-specific appendices to later versions of the specification.

H.3.3 Multitexture

The name string for multitexture is `GL_ARB_multitexture`. It was promoted to a core feature in OpenGL 1.3.

H.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.

H.3.5 Multisample

The name string for multisample is `GL_ARB_multisample`. It was promoted to a core feature in OpenGL 1.3.

H.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.

H.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.

H.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.

H.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.

H.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.

H.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`.

H.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`.

H.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.

H.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.

H.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.

H.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.

H.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.

H.3.18 Shadow

The name string for shadow is `GL_ARB_shadow`. It was promoted to a core feature in OpenGL 1.4.

H.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`.

H.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.

H.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`.

H.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`.

H.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.

H.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.

H.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.

H.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.

H.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.

H.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.

H.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.

H.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.

H.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`.

H.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.

H.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.

H.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.

H.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.

H.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.

H.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.

H.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.

H.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.

H.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.

H.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.

H.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`.

H.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.

H.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`.

H.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.

H.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.

H.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.

H.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.

H.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.

H.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.

H.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.

H.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`.

H.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.

H.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.

Index

*CopyBufferSubData, 38
*GetString, 226
*GetStringi, 227
*MapBuffer, 37
*MapBufferRange, 34
*Pointer, 41
-, 262–264
–, 279
279

Accum, 298
ACCUM_*_BITS, 298
ACCUM_BUFFER_BIT, 298
ACTIVE_ATTRIBUTE_MAX_LENGTH, 47, 232, 264
ACTIVE_ATTRIBUTES, 47, 232, 263
ACTIVE_TEXTURE, 115, 157, 222, 252
ACTIVE_UNIFORM_BLOCK_MAX_NAME_LENGTH, 232, 265
ACTIVE_UNIFORM_BLOCKS, 51, 52, 232, 265
ACTIVE_UNIFORM_MAX_LENGTH, 54, 55, 232, 263
ACTIVE_UNIFORMS, 53, 54, 232, 263
ActiveTexture, 64, 115
ALIASED_LINE_WIDTH_RANGE, 273
ALL_ATTRIB_BITS, 299
ALPHA, 177, 192, 196, 251, 258, 297
ALPHA_BITS, 298
ALPHA_TEST, 298
AlphaFunc, 298
ALWAYS, 142, 158, 171, 172, 253
AND, 180
AND_INVERTED, 180
AND_REVERSE, 180
Antialiasing, 95
AreTexturesResident, 298
ARRAY_BUFFER, 31, 39, 41
ARRAY_BUFFER_BINDING, 39, 242
ArrayElement, 302
ATTACHED_SHADERS, 232, 233, 263
AttachShader, 44
ATTRIB_STACK_DEPTH, 299
AUXi, 298

BACK, 96, 171, 174, 182–186, 188, 192, 199, 247, 297
BACK_LEFT, 183, 236
BACK_RIGHT, 183, 236
Begin, 295
BeginConditionalRender, 77
BeginQuery, 75, 76, 81, 173, 302
BeginTransformFeedback, 78–80
BGR, 106, 192, 196
BGR_INTEGER, 106
BGRA, 106, 108, 113, 192
BGRA_INTEGER, 106
BindAttribLocation, 48
BindBuffer, 30, 32, 40, 63, 79, 139
BindBufferBase, 63, 79, 80

BindBufferRange, 63, 64, 79, 80
BindFragDataLocation, 164
BindFramebuffer, 200, 201, 214
BindRenderbuffer, 202, 203
BindTexture, 64, 115, 155–157
BindVertexArray, 41
BITMAP, 297
Bitmap, 297
BLEND, 174, 178, 179, 254
BLEND_COLOR, 254
BLEND_DST_ALPHA, 254
BLEND_DST_RGB (v1.3:BLEND_DST), 254
BLEND_EQUATION_ALPHA, 254
BLEND_EQUATION_RGB (v1.5: BLEND_EQUATION), 254
BLEND_SRC_ALPHA, 254
BLEND_SRC_RGB (v1.3:BLEND_SRC), 254
BlendColor, 176
BlendEquation, 174
BlendEquationSeparate, 174
BlendFunc, 175
BlendFuncSeparate, 175
BlitFramebuffer, 189, 196, 198, 211
BLUE, 106, 192, 196, 251, 258
BLUE_BITS, 298
BLUE_INTEGER, 106
BOOL, 56
bool, 56, 60
BOOL_VEC2, 56
BOOL_VEC3, 56
BOOL_VEC4, 56
BUFFER_ACCESS, 31, 33, 36, 243
BUFFER_ACCESS_FLAGS, 31, 33, 36, 38, 243, 303
BUFFER_MAP_LENGTH, 31, 33, 36, 38, 243, 303
BUFFER_MAP_OFFSET, 31, 33, 36, 38, 243, 303
BUFFER_MAP_POINTER, 31, 33, 36, 38, 229, 230, 243
BUFFER_MAPPED, 31, 33, 36, 38, 243
BUFFER_SIZE, 31, 33, 34, 36, 37, 63, 79, 243
BUFFER_USAGE, 31, 33, 35, 243
BufferData, 32, 33, 50, 291
BufferSubData, 34, 50, 291
bvec2, 56, 58
bvec3, 56
bvec4, 56
BYTE, 24, 105, 194, 195

CallList, 298
CallLists, 298
CCW, 96, 247
CheckFramebufferStatus, 214, 215
CLAMP, 297
CLAMP_FRAGMENT_COLOR, 296
CLAMP_READ_COLOR, 193, 245
CLAMP_TO_BORDER, 142, 147, 304
CLAMP_TO_EDGE, 142, 147, 155, 197
CLAMP_VERTEX_COLOR, 296
ClampColor, 193, 296
CLEAR, 180
Clear, 77, 85, 187, 189, 298
ClearAccum, 298
ClearBuffer, 189
ClearBuffer*, 77, 85, 302, 303
ClearBuffer{if ui}v, 188, 189
ClearBufferfi, 188, 189
ClearBufferfv, 188, 189
ClearBufferiv, 188, 189
ClearBufferuiv, 188
ClearColor, 187, 188
ClearDepth, 187, 188
ClearStencil, 187, 188
CLIENT_ALL_ATTRIB_BITS, 299

CLIENT_ATTRIB_STACK_DEPTH, 299
ClientActiveTexture, 295
CLIP_DISTANCEi, 82, 244, 302
CLIP_DISTANCE0, 82
CLIP_PLANEi, 302
ClipPlane, 296
COLOR, 129, 188, 189
Color*, 295
COLOR_ATTACHMENTi, 182, 183, 192, 206, 212
COLOR_ATTACHMENTm, 182, 184
COLOR_ATTACHMENTn, 201
COLOR_ATTACHMENT0, 182, 185, 192, 201
COLOR_BUFFER_BIT, 187, 189, 196, 197
COLOR_CLEAR_VALUE, 255
COLOR_INDEX, 295
COLOR_LOGIC_OP, 179, 254
COLOR_MATERIAL, 296
COLOR_SUM, 298
COLOR_WRITEMASK, 185, 186, 255
ColorMask, 185, 186
ColorMaski, 185
ColorMaterial, 296
ColorPointer, 295
COMPARE_R_TO_TEXTURE, 302
COMPARE_REF_TO_TEXTURE, 142, 158, 302
COMPILE_STATUS, 43, 231, 262
CompileShader, 43
COMPRESSED_RED, 124
COMPRESSED_RED_RGTC1, 119, 124, 287, 288
COMPRESSED_RG, 124
COMPRESSED_RG_RGTC2, 119, 124, 288
COMPRESSED_RGB, 124
COMPRESSED_RGBA, 124
COMPRESSED_SIGNED_RED_RGTC1, 119, 124, 288, 289
COMPRESSED_SIGNED_RG_RGTC2, 119, 124, 289
COMPRESSED_SRGB, 124, 159
COMPRESSED_SRGB_ALPHA, 124, 159
COMPRESSED_TEXTURE_FORMATS, 118, 273
CompressedTexImage, 137
CompressedTexImage$n$D, 134
CompressedTexImage*, 214, 308
CompressedTexImage1D, 134–136
CompressedTexImage2D, 134–136
CompressedTexImage3D, 134–136
CompressedTexSubImage$n$D, 136
CompressedTexSubImage1D, 136–138
CompressedTexSubImage2D, 136–138
CompressedTexSubImage3D, 136–138
CONSTANT_ALPHA, 177
CONSTANT_COLOR, 177
CONTEXT_FLAG_FORWARD_COMPATIBLE_BIT, 227
CONTEXT_FLAGS, 227, 274
COPY, 179, 180, 254
COPY_INVERTED, 180
COPY_READ_BUFFER, 31, 39, 279
COPY_WRITE_BUFFER, 31, 39, 279
CopyPixels, 298
CopyTexImage, 215, 298
CopyTexImage*, 207, 211, 214
CopyTexImage1D, 129–131, 133, 134, 150
CopyTexImage2D, 127, 129–131, 133, 134, 150
CopyTexImage3D, 131
CopyTexSubImage, 215
CopyTexSubImage*, 133, 138, 207
CopyTexSubImage1D, 130–134

CopyTexSubImage2D, 130–134
CopyTexSubImage3D, 130, 131, 133, 134
CreateProgram, 44
CreateShader, 42
CULL_FACE, 96, 247
CULL_FACE_MODE, 247
CullFace, 96, 100
CURRENT_PROGRAM, 263
CURRENT_QUERY, 228, 279
CURRENT_VERTEX_ATTRIB, 234, 267
CW, 96

DECR, 171
DECR_WRAP, 171
DELETE_STATUS, 44, 231, 232, 262, 263
DeleteBuffers, 30, 32
DeleteFramebuffers, 200, 201
DeleteLists, 298
DeleteProgram, 46
DeleteQueries, 76
DeleteRenderbuffers, 203, 214
DeleteShader, 43, 44
DeleteTextures, 156, 157, 214
DeleteVertexArrays, 41
DEPTH, 129, 188, 189, 236, 251, 258, 302
DEPTH24_STENCIL8, 119, 123
DEPTH32F_STENCIL8, 119, 123
DEPTH_ATTACHMENT, 201, 206, 212, 302, 303
DEPTH_BITS, 298
DEPTH_BUFFER, 303
DEPTH_BUFFER_BIT, 187, 189, 196–198
DEPTH_CLEAR_VALUE, 255
DEPTH_COMPONENT, 70, 106, 117, 123, 158, 162, 190, 193, 212, 224, 225
DEPTH_COMPONENT16, 119, 123
DEPTH_COMPONENT24, 119, 123
DEPTH_COMPONENT32, 123
DEPTH_COMPONENT32F, 119, 123
DEPTH_FUNC, 254
DEPTH_RANGE, 244
DEPTH_STENCIL, 70, 104, 106, 108, 113, 114, 116, 117, 123, 129, 153, 158, 162, 188, 189, 191, 193, 206, 208, 212, 224, 225
DEPTH_STENCIL_ATTACHMENT, 206, 208, 236
DEPTH_TEST, 172, 254
DEPTH_TEXTURE_MODE, 297
DEPTH_WRITEMASK, 255
DepthFunc, 172
DepthMask, 186
DepthRange, 74, 221
DetachShader, 44
dFdx, 219
dFdy, 219
Disable, 26, 82, 85, 88, 89, 91, 95, 96, 100, 169, 170, 172, 174, 179, 295–298
DisableClientState, 295
Disablei, 174
DisableVertexAttribArray, 25, 234
DITHER, 179, 254
DONT_CARE, 219, 270
DOUBLE, 24
DOUBLEBUFFER, 278
DRAW_BUFFER, 182, 185, 192
DRAW_BUFFERi, 174, 185, 188, 213, 257
DRAW_BUFFER0, 185
DRAW_FRAMEBUFFER, 199–201, 205, 207, 215, 236, 256
DRAW_FRAMEBUFFER_BINDING,

149, 181, 182, 198, 202, 215, 216, 256
DrawArrays, 19, 21, 26, 27, 29, 39, 41, 70, 78, 215
DrawArraysInstanced, 29
DrawBuffer, 180–184, 186, 189
DrawBuffers, 181–184
DrawElements, 26–29, 40, 41
DrawElementsInstanced, 29, 40
DrawPixels, 297
DrawRangeElements, 28, 40, 273
DST_ALPHA, 177
DST_COLOR, 177
DYNAMIC_COPY, 31, 33
DYNAMIC_DRAW, 31, 33
DYNAMIC_READ, 31, 33

EdgeFlag*, 295
EdgeFlagPointer, 295
ELEMENT_ARRAY_BUFFER, 31, 40
ELEMENT_ARRAY_BUFFER_BINDING, 241
Enable, 26, 82, 85, 88, 89, 91, 95, 96, 100, 169, 170, 172, 174, 179, 221, 295–298
EnableClientState, 295
Enablei, 174
EnableVertexAttribArray, 25, 41, 234
End, 295
EndConditionalRender, 77
EndList, 298
EndQuery, 76, 173
EndTransformFeedback, 78, 292, 293
EQUAL, 142, 158, 171, 172
EQUIV, 180
EvalCoord*, 298
EvalMesh*, 298
EvalPoint*, 298
EXTENSIONS, 227, 274, 299, 314

FALSE, 31, 33, 38, 43, 45, 58, 71–73, 101, 102, 154, 162, 170, 191, 193, 221, 226, 228, 229, 231, 232, 234, 235, 238, 240, 242–244, 246–248, 251, 253, 254, 261–263, 267, 268, 279
FASTEST, 219
FeedbackBuffer, 298
FILL, 98, 100, 101, 247, 285
Finish, 218, 284, 291–293
FIXED_ONLY, 193, 199, 245
FLAT, 284
FLOAT, 24, 30, 47, 56, 104, 105, 117, 192, 193, 195, 223, 236, 240
float, 46, 56, 60
FLOAT_32_UNSIGNED_INT_24_8_REV, 104, 105, 108, 112, 191, 194, 195
FLOAT_MAT2, 47, 56
FLOAT_MAT2x3, 47, 56
FLOAT_MAT2x4, 47, 56
FLOAT_MAT3, 47, 56
FLOAT_MAT3x2, 47, 56
FLOAT_MAT3x4, 47, 56
FLOAT_MAT4, 47, 56
FLOAT_MAT4x2, 47, 56
FLOAT_MAT4x3, 47, 56
FLOAT_VEC2, 47, 56
FLOAT_VEC3, 47, 56
FLOAT_VEC4, 47, 56
Flush, 218, 284
FlushMappedBufferRange, 35, 37, 291
FOG, 298
Fog, 298
FOG_HINT, 299
FogCoord*, 295
FogCoordPointer, 295
FRAGMENT_SHADER, 160, 231
FRAGMENT_SHADER_DERIVATIVE_HINT, 219, 270

FRAMEBUFFER, 200, 205, 207, 215, 236
FRAMEBUFFER_ATTACHMENT_x_SIZE, 258
FRAMEBUFFER_ATTACHMENT_ALPHA_SIZE, 236
FRAMEBUFFER_ATTACHMENT_BLUE_SIZE, 236
FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING, 175, 178, 237, 258
FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE, 236, 258
FRAMEBUFFER_ATTACHMENT_DEPTH_SIZE, 236
FRAMEBUFFER_ATTACHMENT_GREEN_SIZE, 236
FRAMEBUFFER_ATTACHMENT_OBJECT_NAME, 206, 208, 212, 236, 237, 258
FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE, 206, 208, 212, 213, 216, 236, 237, 258
FRAMEBUFFER_ATTACHMENT_RED_SIZE, 236
FRAMEBUFFER_ATTACHMENT_STENCIL_SIZE, 236
FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE, 208, 237, 258, 309
FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER, 208, 209, 212, 217, 237, 258
FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL, 149, 208, 210, 237, 258
FRAMEBUFFER_BINDING, 202
FRAMEBUFFER_COMPLETE, 215
FRAMEBUFFER_DEFAULT, 236
FRAMEBUFFER_INCOMPLETE_ATTACHMENT, 213
FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER, 213
FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT, 213
FRAMEBUFFER_INCOMPLETE_MULTISAMPLE, 213
FRAMEBUFFER_INCOMPLETE_READ_BUFFER, 213
FRAMEBUFFER_SRGB, 175, 178, 254
FRAMEBUFFER_UNDEFINED, 213
FRAMEBUFFER_UNSUPPORTED, 213, 214
FramebufferRenderbuffer, 205, 206, 214
FramebufferTexture, 209
FramebufferTexture*, 208, 209, 214
FramebufferTexture1D, 207, 208
FramebufferTexture2D, 207, 208
FramebufferTexture3D, 207–209
FramebufferTextureLayer, 208, 309
FRONT, 96, 171, 174, 182–186, 188, 192, 199, 297
FRONT_AND_BACK, 96, 98, 171, 174, 183–186, 188, 192
FRONT_FACE, 247
FRONT_LEFT, 183, 236
FRONT_RIGHT, 183, 236
FrontFace, 96, 162, 296
Frustum, 295
FUNC_ADD, 174, 176, 254
FUNC_REVERSE_SUBTRACT, 174, 176
FUNC_SUBTRACT, 174, 176
fwidth, 219

Gen*, 290, 295
GenBuffers, 30, 32
GENERATE_MIPMAP, 298
GENERATE_MIPMAP_HINT, 299
GenerateMipmap, 152
GenFramebuffers, 199–202
GenLists, 298
GenQueries, 76
GenRenderbuffers, 202, 203
GenTextures, 156, 157, 226
GenVertexArrays, 40, 41
GEQUAL, 142, 158, 171, 172
Get, 75, 220, 221
GetActiveUniformBlockiv, 266
GetActiveAttrib, 46, 47, 67, 264
GetActiveUniform, 54–56, 59, 263
GetActiveUniformBlockiv, 51
GetActiveUniformBlockName, 51
GetActiveUniformName, 53, 54
GetActiveUniformsiv, 54–56, 265, 266
GetAttachedShaders, 233, 263
GetAttribLocation, 47, 48, 264
GetBooleani_v, 185, 220, 255
GetBooleanv, 170, 186, 220, 221, 238, 239, 248, 255, 261, 278
GetBufferParameteriv, 229, 243, 303
GetBufferPointerv, 230, 243
GetBufferSubData, 229, 230, 243
GetCompressedTexImage, 135, 137, 219, 223, 225, 226
GetDoublev, 220, 221, 239
GetError, 18, 279
GetFloatv, 13, 170, 220, 221, 239, 244, 246–248, 254, 255, 271–273
GetFragDataLocation, 164, 165
GetFramebufferAttachment-Parameteriv, 258
GetFramebufferAttachmentiv, 303
GetFramebufferAttachmentParameteriv, 216, 236, 302, 303
GetIntegeri_v, 220, 230, 265, 269
GetIntegerv, 28, 52, 59, 60, 63, 87, 184, 185, 202, 203, 220, 221, 227, 239, 241, 242, 244–247, 249, 252–257, 259, 261, 263, 265, 269–279
GetProgramInfoLog, 45, 233, 263
GetProgramiv, 45, 47, 51, 53, 54, 66, 67, 71, 232, 233, 263–265
GetQueryiv, 228, 274, 279
GetQueryObject[u]iv, 229
GetQueryObjectiv, 228, 268
GetQueryObjectuiv, 228, 268
GetRenderbufferParameteriv, 260
GetRenderbufferParameteriv, 216, 238, 304
GetShaderInfoLog, 43, 233, 262
GetShaderiv, 43, 44, 231, 233, 234, 262
GetShaderSource, 233, 262
GetString, 227, 274, 299
GetStringi, 274, 314
GetTexImage, 155, 191, 224, 225, 249, 309
GetTexLevelParameter, 222, 223, 251
GetTexParameter, 216, 222, 250
GetTexParameterfv, 155, 250
GetTexParameterI, 222
GetTexParameterIiv, 222
GetTexParameterIuiv, 222
GetTexParameteriv, 155, 250
GetTransformFeedbackVarying, 264
GetTransformFeedbackVarying, 66, 67
GetUniform, 263
GetUniform*, 235
GetUniformBlockIndex, 50, 51
GetUniformfv, 235
GetUniformIndices, 53–55
GetUniformiv, 235
GetUniformLocation, 50, 54, 55, 64, 263

GetUniformuiv, 235
GetVertexAttribPointerv, 240
GetVertexAttribdv, 234
GetVertexAttribfv, 234, 267
GetVertexAttribIiv, 234
GetVertexAttribIuiv, 234
GetVertexAttribiv, 234, 240, 241
GetVertexAttribPointerv, 235
GL_APPLE_flush_buffer_range, 301, 321
GL_APPLE_vertex_array_object, 301, 322
GL_ARB_color_buffer_float, 301, 319
GL_ARB_compatibility, 307, 308, 313, 323
GL_ARB_copy_buffer, 307, 323
GL_ARB_depth_buffer_float, 320
GL_ARB_depth_texture, 316
GL_ARB_draw_buffers, 319
GL_ARB_draw_instanced, 307, 320, 321
GL_ARB_fragment_program, 317, 318
GL_ARB_fragment_program_shadow, 318
GL_ARB_fragment_shader, 318
GL_ARB_framebuffer_object, 320
GL_ARB_framebuffer_sRGB, 321
GL_ARB_geometry_shader4, 321
GL_ARB_half_float_pixel, 301, 320
GL_ARB_half_float_vertex, 321
GL_ARB_instanced_arrays, 321
GL_ARB_map_buffer_range, 321
GL_ARB_matrix_palette, 316
GL_ARB_multisample, 315
GL_ARB_multitexture, 314
GL_ARB_occlusion_query, 317
GL_ARB_pixel_buffer_object, 320
GL_ARB_point_parameters, 315
GL_ARB_point_sprite, 318
GL_ARB_shader_objects, 318
GL_ARB_shader_texture_lod, 323
GL_ARB_shading_language_100, 318
GL_ARB_shadow, 316, 318
GL_ARB_shadow_ambient, 317
GL_ARB_texture_border_clamp, 315
GL_ARB_texture_buffer_object, 308, 322
GL_ARB_texture_compression, 315
GL_ARB_texture_compression_rgtc, 322
GL_ARB_texture_cube_map, 315
GL_ARB_texture_env_add, 315
GL_ARB_texture_env_combine, 316
GL_ARB_texture_env_crossbar, 316
GL_ARB_texture_env_dot3, 316
GL_ARB_texture_float, 301, 320
GL_ARB_texture_mirrored_repeat, 316
GL_ARB_texture_non_power_of_two, 318
GL_ARB_texture_rectangle, 308, 319
GL_ARB_texture_rg, 322
GL_ARB_transpose_matrix, 315
GL_ARB_uniform_buffer_object, 308, 322
GL_ARB_vertex_array_object, 322
GL_ARB_vertex_blend, 316
GL_ARB_vertex_buffer_object, 317
GL_ARB_vertex_program, 317
GL_ARB_vertex_shader, 318
GL_ARB_window_pos, 317
GL_ARB_*name*, 314
gl_BackColor, 296
gl_BackSecondaryColor, 296
gl_ClipDistance, 70, 302
gl_ClipDistance[], 81
gl_ClipVertex, 302
GL_EXT_draw_buffers2, 301
GL_EXT_framebuffer_blit, 301, 320
GL_EXT_framebuffer_multisample, 301, 320

GL_EXT_framebuffer_object, 301, 320
GL_EXT_framebuffer_sRGB, 301, 321
GL_EXT_geometry_shader4, 304
GL_EXT_gpu_shader4, 300
GL_EXT_packed_depth_stencil, 301
GL_EXT_packed_float, 301
GL_EXT_texture_array, 301, 304
GL_EXT_texture_compression_rgtc, 301, 322
GL_EXT_texture_integer, 301
GL_EXT_texture_shared_exponent, 301
GL_EXT_transform_feedback, 301
gl_FragColor, 163, 164, 184
gl_FragCoord, 162
gl_FragCoord.z, 282
gl_FragData, 164, 184
gl_FragData[n], 163
gl_FragDepth, 163, 164, 282
gl_FrontFacing, 162
gl_InstanceID, 28, 70
GL_NV_conditional_render, 300
GL_NV_depth_buffer_float, 301, 320
GL_NV_half_float, 301, 321
GL_NV_primitive_restart, 307
gl_PointCoord, 89
gl_PointSize, 88
gl_Position, 65, 70, 73
gl_PrimitiveID, 163
gl_VertexID, 70, 163
GLX_ARB_create_context, 322
GLX_ARB_fbconfig_float, 319
GLX_ARB_framebuffer_sRGB, 321
GREATER, 142, 158, 171, 172
GREEN, 106, 192, 196, 251, 258
GREEN_BITS, 298
GREEN_INTEGER, 106

HALF, 193
HALF_FLOAT, 24, 105, 117, 192, 193, 195
Hint, 218, 299

INCR, 171
INCR_WRAP, 171
Index*, 295
IndexPointer, 295
INFO_LOG_LENGTH, 231–233, 262, 263
InitNames, 298
INT, 24, 47, 56, 105, 194, 195, 223, 236
int, 56, 60
INT_SAMPLER_1D, 56
INT_SAMPLER_1D_ARRAY, 56
INT_SAMPLER_2D, 56
INT_SAMPLER_2D_ARRAY, 56
INT_SAMPLER_3D, 56
INT_SAMPLER_CUBE, 56
INT_VEC2, 47, 56
INT_VEC3, 47, 56
INT_VEC4, 47, 56
INTENSITY, 297
INTERLEAVED_ATTRIBS, 65, 66, 79, 232, 264
InterleavedArrays, 295
INVALID_ENUM, 18, 19, 37, 104, 115, 129, 134, 136, 138, 141, 156, 182, 189, 191, 192, 225, 237, 238, 304
INVALID_FRAMEBUFFER_OPERATION, 19, 129, 133, 192, 198, 215, 216
INVALID_INDEX, 51, 53
INVALID_OPERATION, 19, 32, 34, 36–39, 41, 42, 44, 45, 48, 50, 58, 59, 64, 66, 71, 76–78, 80, 104, 108, 115, 117, 125, 129, 133, 135–139, 152, 156, 164, 165, 182, 184, 190–192, 196–198, 200, 203–205, 207, 209, 222, 224–226, 228, 231, 235,

236, 238, 295, 296, 302, 303
INVALID_VALUE, 18, 19, 24–28, 34, 36, 37, 39, 42, 47, 48, 51–54, 63, 65, 66, 75, 77, 79, 88, 89, 91, 101, 117, 124–126, 129–132, 135, 136, 151, 164, 169, 174, 182, 184, 185, 187, 189, 204, 208, 209, 221, 223, 225–227, 231, 234, 235, 296, 297, 309
INVERT, 171, 180
isampler1D, 56
isampler1DArray, 56
isampler2D, 56
isampler2DArray, 56
isampler3D, 56
isamplerCube, 56
IsBuffer, 229
IsEnabled, 169, 178, 221, 239, 242, 244, 246–248, 253, 254, 267
IsEnabledi, 178, 221, 254
IsFramebuffer, 235
IsList, 298
IsProgram, 232
IsQuery, 227, 228
IsRenderbuffer, 238
IsShader, 231
IsTexture, 226
IsVertexArray, 231
ivec2, 56
ivec3, 56
ivec4, 56

KEEP, 171, 172, 253

layout, 61
LEFT, 174, 182–185, 188, 192
LEQUAL, 142, 155, 158, 171, 172, 250
LESS, 142, 158, 171, 172, 254
LIGHTi, 296
Light*, 296
LIGHTING, 296
LightModel*, 296
LINE, 98, 100, 247
LINE_LOOP, 21, 78
LINE_SMOOTH, 91, 95, 246
LINE_SMOOTH_HINT, 219, 270
LINE_STIPPLE, 296
LINE_STRIP, 21, 78
LINE_WIDTH, 246
LINEAR, 68, 141, 142, 148, 149, 151, 152, 155, 197, 198, 210, 237, 250
LINEAR_MIPMAP_LINEAR, 142, 149–151, 210
LINEAR_MIPMAP_NEAREST, 142, 149–151, 210
LINES, 21, 78
LineStipple, 296
LineWidth, 91, 296, 307
LINK_STATUS, 45, 232, 263
LinkProgram, 44, 45, 47, 48, 51, 53, 54, 63, 64, 66, 80, 164
ListBase, 298
LoadIdentity, 295
LoadMatrix, 295
LoadName, 298
LoadTransposeMatrix, 295
LOGIC_OP_MODE, 254
LogicOp, 179, 180
LOWER_LEFT, 89, 90
LUMINANCE, 297
LUMINANCE_ALPHA, 297

MAJOR_VERSION, 227, 274
Map*, 298
MAP_FLUSH_EXPLICIT_BIT, 35–37
MAP_INVALIDATE_BUFFER_BIT, 35, 36

MAP_INVALIDATE_RANGE_BIT, 35, 36
MAP_READ_BIT, 34–37
MAP_UNSYNCHRONIZED_BIT, 35, 36
MAP_WRITE_BIT, 34–37
MapBuffer, 34, 37, 50, 63, 79, 80, 303
MapBufferRange, 34–37, 303
MapGrid*, 298
mat$C$, 60, 61
mat$C$xR, 60, 61
mat2, 46, 56
mat2x3, 46, 56
mat2x4, 46, 56
mat3, 46, 56
mat3x2, 46, 56
mat3x4, 46, 56
mat4, 46, 56
mat4x2, 46, 56
mat4x3, 46, 56
Material*, 296
MatrixMode, 295
MAX, 174, 176
MAX_3D_TEXTURE_SIZE, 125, 207, 208, 271
MAX_ARRAY_TEXTURE_LAYERS, 125, 271
MAX_ATTRIB_STACK_DEPTH, 299
MAX_CLIENT_ATTRIB_STACK_DEPTH, 299
MAX_CLIP_DISTANCES, 271, 302, 308
MAX_CLIP_PLANES, 302
MAX_COLOR_ATTACHMENTS, 182–184, 200, 206, 215, 278
MAX_COMBINED_FRAGMENT_UNIFORM_COMPONENTS, 160, 277
MAX_COMBINED_TEXTURE_IMAGE_UNITS, 69, 115, 222, 275
MAX_COMBINED_UNIFORM_BLOCKS, 60, 276
MAX_COMBINED_VERTEX_UNIFORM_COMPONENTS, 49, 277
MAX_CUBE_MAP_TEXTURE_SIZE, 125, 208, 271
MAX_DRAW_BUFFERS, 164, 174, 176, 184, 185, 189, 278
MAX_ELEMENTS_INDICES, 28, 273
MAX_ELEMENTS_VERTICES, 28, 273
MAX_FRAGMENT_UNIFORM_BLOCKS, 59, 276
MAX_FRAGMENT_UNIFORM_COMPONENTS, 160, 275, 277
MAX_PROGRAM_TEXEL_OFFSET, 145, 275
MAX_RECTANGLE_TEXTURE_SIZE, 125, 273
MAX_RENDERBUFFER_SIZE, 204, 271
MAX_SAMPLES, 204, 205, 278
MAX_TEXTURE_BUFFER_SIZE, 139, 273
MAX_TEXTURE_COORDS, 298
MAX_TEXTURE_IMAGE_UNITS, 69, 162, 275
MAX_TEXTURE_LOD_BIAS, 144, 271
MAX_TEXTURE_SIZE, 125, 208, 271
MAX_TEXTURE_UNITS, 298
MAX_TRANSFORM_FEEDBACK_INTERLEAVED_COMPONENTS, 66, 269
MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS, 65, 79, 80, 230, 269

MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS, 66, 269
MAX_UNIFORM_BLOCK_SIZE, 52, 276
MAX_UNIFORM_BUFFER_BINDINGS, 63, 230, 276
MAX_VARYING_COMPONENTS, 65, 275, 302
MAX_VARYING_FLOATS, 302
MAX_VERTEX_ATTRIBS, 23–26, 29, 46, 48, 234, 235, 275
MAX_VERTEX_TEXTURE_IMAGE_UNITS, 69, 275
MAX_VERTEX_UNIFORM_BLOCKS, 59, 276
MAX_VERTEX_UNIFORM_COMPONENTS, 49, 275, 277
MAX_VIEWPORT_DIMS, 228, 272
MIN, 174, 176
MIN_PROGRAM_TEXEL_OFFSET, 145, 275
MINOR_VERSION, 227, 274
MIRRORED_REPEAT, 141, 142, 147
MultiDrawArrays, 27
MultiDrawElements, 28, 40
MULTISAMPLE, 88, 90, 95, 100, 169, 180, 248
MultMatrix, 295
MultTransposeMatrix, 295

NAND, 180
NEAREST, 68, 141, 142, 147, 149, 151–153, 159, 197, 210
NEAREST_MIPMAP_LINEAR, 142, 149–152, 155, 210
NEAREST_MIPMAP_NEAREST, 142, 149–153, 159, 210
NEVER, 142, 158, 171, 172
NewList, 298
NICEST, 219
NO_ERROR, 18
NONE, 70, 142, 154, 155, 158, 162, 180, 182–185, 189, 192, 199, 212, 213, 223, 236, 250, 251, 258, 303, 309
NOOP, 180
noperspective, 83
NOR, 180
Normal3*, 295
NORMALIZE, 296
NormalPointer, 295
NOTEQUAL, 142, 158, 171, 172
NULL, 30, 31, 33, 36, 40, 41, 43, 47, 51, 54, 67, 230, 233, 239, 240, 243
NUM_COMPRESSED_TEXTURE_FORMATS, 118, 273
NUM_EXTENSIONS, 227, 274
NV, 308

ONE, 176, 177, 254
ONE_MINUS_CONSTANT_ALPHA, 177
ONE_MINUS_CONSTANT_COLOR, 177
ONE_MINUS_DST_ALPHA, 177
ONE_MINUS_DST_COLOR, 177
ONE_MINUS_SRC_ALPHA, 177
ONE_MINUS_SRC_COLOR, 177
OR, 180
OR_INVERTED, 180
OR_REVERSE, 180
Ortho, 295
OUT_OF_MEMORY, 18, 19, 34, 36, 204

PACK_ALIGNMENT, 191, 261

PACK_IMAGE_HEIGHT, 191, 225, 261
PACK_LSB_FIRST, 191, 261
PACK_ROW_LENGTH, 191, 261
PACK_SKIP_IMAGES, 191, 225, 261
PACK_SKIP_PIXELS, 191, 261
PACK_SKIP_ROWS, 191, 261
PACK_SWAP_BYTES, 191, 261
PassThrough, 298
PERSPECTIVE_CORRECTION_HINT, 299
PIXEL_PACK_BUFFER, 31, 102, 189
PIXEL_PACK_BUFFER_BINDING, 194, 225, 261
PIXEL_UNPACK_BUFFER, 31, 102
PIXEL_UNPACK_BUFFER_BINDING, 104, 134, 261
PixelStore, 101, 102, 191, 198, 199
PixelZoom, 297
POINT, 98, 100, 247
POINT_FADE_THRESHOLD_SIZE, 89, 246
POINT_SIZE, 246
POINT_SIZE_GRANULARITY, 272
POINT_SIZE_RANGE, 272
POINT_SMOOTH, 296
POINT_SMOOTH_HINT, 299
POINT_SPRITE, 296
POINT_SPRITE_COORD_ORIGIN, 89, 90, 246
PointParameter, 89
PointParameter*, 89
POINTS, 21, 78, 98
PointSize, 88
POLYGON, 296
POLYGON_OFFSET_FACTOR, 247
POLYGON_OFFSET_FILL, 100, 247
POLYGON_OFFSET_LINE, 100, 247
POLYGON_OFFSET_POINT, 100, 247
POLYGON_OFFSET_UNITS, 247
POLYGON_SMOOTH, 95, 100, 247
POLYGON_SMOOTH_HINT, 219, 270
POLYGON_STIPPLE, 297
PolygonMode, 98, 100, 101, 297
PolygonOffset, 99
PolygonStipple, 297
PopAttrib, 299
PopClientAttrib, 299
PopMatrix, 295
PopName, 298
PRIMITIVE_RESTART, 26, 242, 307, 308
PRIMITIVE_RESTART_INDEX, 242, 308
PrimitiveRestartIndex, 26
PRIMITIVES_GENERATED, 81, 228
PrioritizeTextures, 298
PROXY_TEXTURE_1D, 117, 127, 155, 223
PROXY_TEXTURE_1D_ARRAY, 117, 126, 155, 223
PROXY_TEXTURE_2D, 117, 126, 155, 223
PROXY_TEXTURE_2D_ARRAY, 116, 117, 155, 223
PROXY_TEXTURE_3D, 116, 155, 223
PROXY_TEXTURE_CUBE_MAP, 117, 126, 155, 223
PROXY_TEXTURE_RECTANGLE, 117, 126, 134, 136, 155, 223
PushAttrib, 299
PushClientAttrib, 299
PushMatrix, 295
PushName, 298

QUAD_STRIP, 296
QUADS, 296
QUERY_BY_REGION_NO_WAIT, 77
QUERY_BY_REGION_WAIT, 77
QUERY_COUNTER_BITS, 228, 274

QUERY_NO_WAIT, 77
QUERY_RESULT, 228, 268
QUERY_RESULT_AVAILABLE, 229, 268
QUERY_WAIT, 77

R, 301
R11F_G11F_B10F, 119, 122
R16, 119, 121, 140
R16_SNORM, 119, 121
R16F, 119, 121, 140
R16I, 119, 122, 140
R16UI, 119, 122, 140
R32F, 119, 121, 140
R32I, 119, 122, 140
R32UI, 119, 122, 140
R3_G3_B2, 121
R8, 119, 121, 140, 154, 251
R8_SNORM, 119, 121
R8I, 119, 122, 140
R8UI, 119, 122, 140
RASTERIZER_DISCARD, 85
RasterPos*, 296
READ_BUFFER, 192, 213, 217, 257
READ_FRAMEBUFFER, 199–201, 205, 207, 215, 236, 256
READ_FRAMEBUFFER_BINDING, 129, 133, 190, 192, 193, 198, 202, 256
READ_ONLY, 31, 36, 37
READ_WRITE, 31, 33, 36, 37, 243
ReadBuffer, 183, 191, 192, 198
ReadPixels, 80, 101, 107, 129, 189–192, 194, 215, 225, 297
Rect*, 296
RED, 106, 117, 121, 122, 124, 141, 143, 161, 192, 196, 211, 226, 251, 258
RED_BITS, 298
RED_INTEGER, 106
RENDERBUFFER, 202–206, 216, 236–238, 259
RENDERBUFFER_ALPHA_SIZE, 238, 260
RENDERBUFFER_BINDING, 203, 259
RENDERBUFFER_BLUE_SIZE, 238, 260
RENDERBUFFER_DEPTH_SIZE, 238, 260
RENDERBUFFER_GREEN_SIZE, 238, 260
RENDERBUFFER_HEIGHT, 204, 238, 260
RENDERBUFFER_INTERNAL_FORMAT, 204, 238, 260
RENDERBUFFER_RED_SIZE, 238, 260
RENDERBUFFER_SAMPLES, 204, 213, 215, 238, 260
RENDERBUFFER_STENCIL_SIZE, 238, 260
RENDERBUFFER_WIDTH, 204, 238, 260
RenderbufferStorage, 204, 205, 214
RenderbufferStorageMultisample, 204, 205, 309
RENDERER, 226, 274
RenderMode, 298
REPEAT, 141, 142, 147, 155
REPLACE, 171
RESCALE_NORMAL, 295
RG, 106, 117, 121, 122, 124, 161, 192, 196, 211, 226, 301
RG16, 119, 121, 140
RG16_SNORM, 119, 121
RG16F, 119, 121, 140
RG16I, 119, 122, 140
RG16UI, 119, 122, 140

RG32F, 119, 121, 140
RG32I, 119, 122, 140
RG32UI, 119, 122, 140
RG8, 119, 121, 140
RG8_SNORM, 119, 121
RG8I, 119, 122, 140
RG8UI, 119, 122, 140
RG_INTEGER, 106
RGB, 106, 108, 113, 117, 120–122, 124, 161, 177, 192, 194, 196, 211, 226
RGB10, 121
RGB10_A2, 119, 121
RGB12, 121
RGB16, 119, 121
RGB16_SNORM, 119, 121
RGB16F, 119, 121
RGB16I, 119, 122
RGB16UI, 119, 122
RGB32F, 119, 121
RGB32I, 119, 122
RGB32UI, 119, 122
RGB4, 121
RGB5, 121
RGB5_A1, 121
RGB8, 119, 121
RGB8_SNORM, 119, 121
RGB8I, 119, 122
RGB8UI, 119, 122
RGB9_E5, 119, 122, 159, 194
RGB_INTEGER, 106
RGBA, 106, 108, 113, 117, 121, 122, 124, 154, 161, 192, 211, 226, 251, 260, 297
RGBA12, 121
RGBA16, 119, 121, 140
RGBA16_SNORM, 119, 121
RGBA16F, 119, 121, 140
RGBA16I, 119, 122, 140
RGBA16UI, 119, 122, 140
RGBA2, 121
RGBA32F, 119, 122, 140
RGBA32I, 119, 122, 140
RGBA32UI, 119, 122, 140
RGBA4, 121
RGBA8, 119, 121, 140
RGBA8_SNORM, 119, 121
RGBA8I, 119, 122, 140
RGBA8UI, 119, 122, 140
RGBA_INTEGER, 106
RIGHT, 174, 182–185, 188, 192
Rotate, 295

SAMPLE_ALPHA_TO_COVERAGE, 169, 248
SAMPLE_ALPHA_TO_ONE, 169, 170, 248
SAMPLE_BUFFERS, 87, 90, 95, 100, 129, 169, 173, 180, 186, 190, 191, 198, 215, 278
SAMPLE_COVERAGE, 169, 170, 248
SAMPLE_COVERAGE_INVERT, 169, 170, 248
SAMPLE_COVERAGE_VALUE, 169, 170, 248
SampleCoverage, 170
sampler1D, 56
sampler1DArray, 56
sampler1DArrayShadow, 56
sampler1DShadow, 56, 69, 161
sampler2D, 56, 64
sampler2DArray, 56
sampler2DArrayShadow, 56
sampler2DRect, 56
sampler2DRectShadow, 56, 69, 162
sampler2DShadow, 56, 69, 162
sampler3D, 56
SAMPLER_1D, 56
SAMPLER_1D_ARRAY, 56

SAMPLER_1D_ARRAY_SHADOW, 56
SAMPLER_1D_SHADOW, 56
SAMPLER_2D, 56
SAMPLER_2D_ARRAY, 56
SAMPLER_2D_ARRAY_SHADOW, 56
SAMPLER_2D_RECT, 56
SAMPLER_2D_RECT_SHADOW, 56
SAMPLER_2D_SHADOW, 56
SAMPLER_3D, 56
SAMPLER_CUBE, 56
SAMPLER_CUBE_SHADOW, 56
samplerCube, 56
samplerCubeShadow, 56
SAMPLES, 87, 88, 173, 198, 215, 278
SAMPLES_PASSED, 77, 173, 228
Scale, 295
Scissor, 169
SCISSOR_BOX, 253
SCISSOR_TEST, 169, 253
SecondaryColor3*, 295
SecondaryColorPointer, 295
SelectBuffer, 298
SEPARATE_ATTRIBS, 65, 66, 79, 80, 232
SET, 180
ShadeModel, 296
SHADER_SOURCE_LENGTH, 231, 234, 262
SHADER_TYPE, 72, 231, 262
ShaderSource, 43, 234
SHADING_LANGUAGE_VERSION, 226, 227, 274
SHORT, 24, 105, 194, 195
SIGNED_NORMALIZED, 223, 236
SMOOTH_LINE_WIDTH_GRANULARITY (v1.1: LINE_WIDTH_GRANULARITY), 273
SMOOTH_LINE_WIDTH_RANGE (v1.1: LINE_WIDTH_RANGE), 273
SRC_ALPHA, 177
SRC_ALPHA_SATURATE, 177
SRC_COLOR, 177
SRGB, 159, 175, 178, 237
SRGB8, 119, 121, 159
SRGB8_ALPHA8, 119, 121, 159
SRGB_ALPHA, 159
STATIC_COPY, 31, 33
STATIC_DRAW, 31, 33, 243
STATIC_READ, 31, 33
std140, 52, 61
STENCIL, 188, 189, 236, 251, 258, 302
STENCIL_ATTACHMENT, 201, 206, 212, 303
STENCIL_ATTACMENT, 302, 303
STENCIL_BACK_FAIL, 253
STENCIL_BACK_FUNC, 253
STENCIL_BACK_PASS_DEPTH_FAIL, 253
STENCIL_BACK_PASS_DEPTH_PASS, 253
STENCIL_BACK_REF, 253
STENCIL_BACK_VALUE_MASK, 253
STENCIL_BACK_WRITEMASK, 255
STENCIL_BITS, 298
STENCIL_BUFFER, 303
STENCIL_BUFFER_BIT, 187, 189, 196–198
STENCIL_CLEAR_VALUE, 255
STENCIL_FAIL, 253
STENCIL_FUNC, 253
STENCIL_INDEX, 106, 116, 191, 193, 204, 212, 225
STENCIL_INDEX1, 204
STENCIL_INDEX16, 204
STENCIL_INDEX4, 204
STENCIL_INDEX8, 204

STENCIL_PASS_DEPTH_FAIL, 253
STENCIL_PASS_DEPTH_PASS, 253
STENCIL_REF, 253
STENCIL_TEST, 170, 253
STENCIL_VALUE_MASK, 253
STENCIL_WRITEMASK, 255
StencilFunc, 170–172, 284
StencilFuncSeparate, 170, 171
StencilMask, 186, 284
StencilMaskSeparate, 186
StencilOp, 170–172
StencilOpSeparate, 170–172
STEREO, 278
STREAM_COPY, 31, 32
STREAM_DRAW, 31, 32
STREAM_READ, 31, 32
SUBPIXEL_BITS, 271

TexBuffer, 138
TexCoord*, 295
TexCoordPointer, 295
TexEnv, 298
TexGen*, 296
TexImage, 115, 131
TexImage*, 107, 291, 297, 308
TexImage*D, 101, 102
TexImage1D, 102, 123, 126, 127, 129–131, 134, 136, 150, 155
TexImage2D, 102, 123, 126, 127, 129, 131, 134, 136, 150, 155
TexImage3D, 102, 115, 116, 123, 124, 126, 127, 131, 134, 136, 150, 155, 225
TexParameter, 115, 141, 291, 298
TexParameter*, 298
TexParameter[if], 144, 151
TexParameterI, 141
TexParameterIiv, 141
TexParameterIuiv, 141
TexParameteriv, 141
TexSubImage, 131
TexSubImage*, 133, 138, 291
TexSubImage*D, 101
TexSubImage1D, 102, 130–133, 136
TexSubImage2D, 102, 130–133, 136
TexSubImage3D, 102, 130, 131, 133, 136
TEXTURE, 208, 212, 216, 236, 237
TEXTUREi, 115
TEXTURE0, 115, 252
TEXTURE_x_SIZE, 251
TEXTURE_x_TYPE, 251
TEXTURE_xD, 249
TEXTURE_1D, 117, 127, 130, 141, 152, 156, 157, 208, 222–224, 298
TEXTURE_1D_ARRAY, 117, 126, 129, 130, 141, 152, 156, 157, 222–224, 249, 298
TEXTURE_2D, 64, 117, 126, 129, 130, 141, 152, 156, 157, 208, 222–224, 298
TEXTURE_2D_ARRAY, 116, 117, 124, 130, 136, 138, 141, 152, 156, 157, 222–224, 249, 298
TEXTURE_3D, 116, 124, 130, 141, 152, 155–157, 207, 208, 222–224, 298
TEXTURE_ALPHA_SIZE, 223
TEXTURE_ALPHA_TYPE, 223
TEXTURE_BASE_LEVEL, 141, 142, 149, 151, 155, 210, 250
TEXTURE_BINDING_xD, 249
TEXTURE_BINDING_1D_ARRAY, 249
TEXTURE_BINDING_2D_ARRAY, 249
TEXTURE_BINDING_BUFFER, 249
TEXTURE_BINDING_CUBE_MAP, 249

TEXTURE_BINDING_RECTANGLE, 249
TEXTURE_BLUE_SIZE, 223
TEXTURE_BLUE_TYPE, 223
TEXTURE_BORDER, 135, 137, 224
TEXTURE_BORDER_COLOR, 141, 142, 148, 154, 155, 222, 250, 304
TEXTURE_BUFFER, 31, 139, 156, 157, 223, 249, 279
TEXTURE_BUFFER_DATA_STORE_BINDING, 251
TEXTURE_COMPARE_FAIL_VALUE_ARB, 317
TEXTURE_COMPARE_FUNC, 142, 155, 157, 158, 250
TEXTURE_COMPARE_MODE, 69, 70, 142, 155, 157, 158, 162, 250
TEXTURE_COMPONENTS, 297
TEXTURE_COMPRESSED, 251
TEXTURE_COMPRESSED_IMAGE_SIZE, 135, 137, 223, 226, 251
TEXTURE_COMPRESSION_HINT, 219, 270
TEXTURE_CUBE_MAP, 117, 126, 141, 152, 156, 157, 222, 223, 249, 298
TEXTURE_CUBE_MAP_*, 126
TEXTURE_CUBE_MAP_NEGATIVE_X, 126, 129, 130, 143, 207, 208, 223, 224, 249
TEXTURE_CUBE_MAP_NEGATIVE_Y, 126, 129, 130, 143, 207, 208, 223, 224, 249
TEXTURE_CUBE_MAP_NEGATIVE_Z, 126, 129, 130, 143, 207, 208, 223, 224, 249
TEXTURE_CUBE_MAP_POSITIVE_X, 126, 129, 130, 143, 207, 208, 223, 224, 249
TEXTURE_CUBE_MAP_POSITIVE_Y, 126, 129, 130, 143, 207, 208, 223, 224, 249
TEXTURE_CUBE_MAP_POSITIVE_Z, 126, 129, 130, 143, 207, 208, 223, 224, 249
TEXTURE_DEPTH, 135, 137, 224, 251
TEXTURE_DEPTH_SIZE, 223
TEXTURE_DEPTH_TYPE, 223
TEXTURE_ENV, 298
TEXTURE_FILTER_CONTROL, 298
TEXTURE_GEN_*, 296
TEXTURE_GREEN_SIZE, 223
TEXTURE_GREEN_TYPE, 223
TEXTURE_HEIGHT, 133, 135, 137, 138, 224, 251
TEXTURE_INTERNAL_FORMAT, 135, 137, 224, 251, 297
TEXTURE_LOD_BIAS, 142, 144, 250, 298
TEXTURE_MAG_FILTER, 142, 152, 153, 155, 159, 250
TEXTURE_MAX_LEVEL, 141, 142, 151, 155, 210, 250
TEXTURE_MAX_LOD, 141, 142, 144, 155, 250
TEXTURE_MIN_FILTER, 141, 142, 146–150, 152, 153, 155, 159, 210, 250
TEXTURE_MIN_LOD, 141, 142, 144, 155, 250
TEXTURE_PRIORITY, 298
TEXTURE_RECTANGLE, 117, 126, 129–131, 134, 136, 141, 156, 157, 207, 208, 222–225, 249
TEXTURE_RECTANGLE_ARB, 319
TEXTURE_RED_SIZE, 223
TEXTURE_RED_TYPE, 223

TEXTURE_SHARED_SIZE, 223, 251
TEXTURE_STENCIL_SIZE, 223
TEXTURE_WIDTH, 133, 135, 137, 138, 224, 251
TEXTURE_WRAP_R, 141, 142, 147, 250, 297
TEXTURE_WRAP_S, 141, 142, 147, 250, 297
TEXTURE_WRAP_T, 141, 142, 147, 250, 297
TRANSFORM_FEEDBACK_BUFFER, 31, 79, 80
TRANSFORM_FEEDBACK_BUFFER_BINDING, 230, 269
TRANSFORM_FEEDBACK_BUFFER_MODE, 232, 264
TRANSFORM_FEEDBACK_BUFFER_SIZE, 230, 269
TRANSFORM_FEEDBACK_BUFFER_START, 230, 269
TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, 80, 81, 228
TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH, 67, 232, 264
TRANSFORM_FEEDBACK_VARYINGS, 66, 232, 264
TransformFeedbackVaryings, 65, 66, 80
Translate, 295
TRIANGLE_FAN, 22, 78
TRIANGLE_STRIP, 21, 22, 78
TRIANGLES, 22, 78
TRUE, 25, 31, 36, 38, 43, 45, 58, 71, 101, 102, 162, 170, 185, 191, 193, 221, 226, 228, 229, 231, 232, 234, 235, 238, 248, 254, 255, 296
uint, 60
Uniform, 14, 57
Uniform*, 49, 50, 58, 59, 64
Uniform*f{v}, 58
Uniform*i{v}, 58
Uniform*ui{v}, 58
Uniform1f, 15
Uniform1i, 14
Uniform1i{v}, 58, 64
Uniform1iv, 58
Uniform2{if ui}*, 58
Uniform2f, 15
Uniform2i, 15
Uniform3f, 15
Uniform3i, 15
Uniform4f, 13, 15
Uniform4f{v}, 59
Uniform4i, 15
Uniform4i{v}, 59
UNIFORM_ARRAY_STRIDE, 57, 61, 266
UNIFORM_BLOCK_ACTIVE_UNIFORM_INDICES, 52, 266
UNIFORM_BLOCK_ACTIVE_UNIFORMS, 52, 266
UNIFORM_BLOCK_BINDING, 52, 266
UNIFORM_BLOCK_DATA_SIZE, 52, 64, 266
UNIFORM_BLOCK_INDEX, 57, 265
UNIFORM_BLOCK_NAME_LENGTH, 52
UNIFORM_BLOCK_REFERENCED_BY_FRAGMENT_SHADER, 52, 266
UNIFORM_BLOCK_REFERENCED_BY_VERTEX_SHADER, 52, 266
UNIFORM_BUFFER, 31, 63
UNIFORM_BUFFER_BINDING, 230,

265
UNIFORM_BUFFER_OFFSET_ALIGNMENT, 63, 276
UNIFORM_BUFFER_SIZE, 230
UNIFORM_BUFFER_START, 230
UNIFORM_IS_ROW_MAJOR, 57, 266
UNIFORM_MATRIX_STRIDE, 57, 60, 61, 266
UNIFORM_NAME_LENGTH, 55, 265
UNIFORM_OFFSET, 57, 265
UNIFORM_SIZE, 55, 265
UNIFORM_TYPE, 55, 265
Uniform{1,2,3,4}ui, 57
Uniform{1,2,3,4}uiv, 57
UniformBlockBinding, 63, 64
UniformMatrix2x4fv, 58
UniformMatrix3fv, 59
UniformMatrix{234}fv, 57, 58
UniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}fv, 57, 58
UnmapBuffer, 35, 37, 38, 50, 291
UNPACK_ALIGNMENT, 102, 107, 116, 261
UNPACK_IMAGE_HEIGHT, 102, 116, 261
UNPACK_LSB_FIRST, 102, 261
UNPACK_ROW_LENGTH, 102, 104, 107, 116, 261
UNPACK_SKIP_IMAGES, 102, 116, 126, 261
UNPACK_SKIP_PIXELS, 102, 107, 261
UNPACK_SKIP_ROWS, 102, 107, 261
UNPACK_SWAP_BYTES, 102, 104, 106, 261
unsigned int, 56
UNSIGNED_BYTE, 24, 27, 105, 194, 195
UNSIGNED_BYTE_2_3_3_REV, 105, 108, 109, 195
UNSIGNED_BYTE_3_3_2, 105, 108, 109, 195
UNSIGNED_INT, 24, 27, 47, 56, 105, 194, 195, 223, 236
UNSIGNED_INT_10_10_10_2, 105, 108, 111, 195
UNSIGNED_INT_10F_11F_11F_REV, 105, 108, 111, 113, 117, 193–195
UNSIGNED_INT_24_8, 104, 105, 108, 111, 191, 194, 195
UNSIGNED_INT_2_10_10_10_REV, 105, 108, 111, 195
UNSIGNED_INT_5_9_9_9_REV, 105, 108, 111, 113, 117, 120, 193–195
UNSIGNED_INT_8_8_8_8, 105, 108, 111, 195
UNSIGNED_INT_8_8_8_8_REV, 105, 108, 111, 195
UNSIGNED_INT_SAMPLER_1D, 56
UNSIGNED_INT_SAMPLER_1D_ARRAY, 56
UNSIGNED_INT_SAMPLER_2D, 56
UNSIGNED_INT_SAMPLER_2D_ARRAY, 56
UNSIGNED_INT_SAMPLER_3D, 56
UNSIGNED_INT_SAMPLER_CUBE, 56
UNSIGNED_INT_VEC2, 47, 56
UNSIGNED_INT_VEC3, 47, 56
UNSIGNED_INT_VEC4, 47, 56
UNSIGNED_NORMALIZED, 223, 236
UNSIGNED_SHORT, 24, 27, 105, 194, 195
UNSIGNED_SHORT_1_5_5_5_REV, 105, 108, 110, 195
UNSIGNED_SHORT_4_4_4_4, 105, 108, 110, 195
UNSIGNED_SHORT_4_4_4_4_REV,

105, 108, 110, 195
UNSIGNED_SHORT_5_5_5_1, 105, 108, 110, 195
UNSIGNED_SHORT_5_6_5, 105, 108, 110, 195
UNSIGNED_SHORT_5_6_5_REV, 105, 108, 110, 195
UPPER_LEFT, 89, 90, 246
usampler1D, 56
usampler1DArray, 56
usampler2D, 56
usampler2DArray, 56
usampler3D, 56
usamplerCube, 56
UseProgram, 45, 67, 80
uvec2, 56
uvec3, 56
uvec4, 56

VALIDATE_STATUS, 71, 232, 263
ValidateProgram, 71, 232
vec2, 46, 56
vec3, 46, 56
vec4, 46, 56, 59
VENDOR, 226, 274
VERSION, 226, 227, 274
Vertex*, 295
VERTEX_ARRAY_BINDING, 222, 234, 242
VERTEX_ATTRIB_ARRAY_BUFFER_BINDING, 39, 234, 241
VERTEX_ATTRIB_ARRAY_ENABLED, 234, 240
VERTEX_ATTRIB_ARRAY_INTEGER, 234, 240
VERTEX_ATTRIB_ARRAY_NORMALIZED, 234, 240
VERTEX_ATTRIB_ARRAY_POINTER, 235, 240
VERTEX_ATTRIB_ARRAY_SIZE, 234, 240
VERTEX_ATTRIB_ARRAY_STRIDE, 234, 240
VERTEX_ATTRIB_ARRAY_TYPE, 234, 240
VERTEX_PROGRAM_POINT_SIZE, 89, 267
VERTEX_PROGRAM_TWO_SIDE, 296
VERTEX_SHADER, 42, 231
VertexAttrib, 23, 77
VertexAttrib*, 23, 24, 46, 295
VertexAttrib1*, 23
VertexAttrib2*, 23
VertexAttrib3*, 23
VertexAttrib4, 23
VertexAttrib4*, 23
VertexAttrib4N, 23
VertexAttrib4Nub, 23
VertexAttribI, 23
VertexAttribI4, 24
VertexAttribIPointer, 24, 25, 234
VertexAttribPointer, 24, 25, 39, 41, 234, 296
VertexPointer, 295
VIEWPORT, 244
Viewport, 74

WGL_ARB_create_context, 322
WGL_ARB_framebuffer_sRGB, 321
WGL_ARB_pixel_format_float, 319
WindowPos*, 296
WRITE_ONLY, 31, 36, 37

XOR, 180

ZERO, 171, 176, 177, 254