

The OpenGL[®] Graphics System:
A Specification
(Version 3.0 - August 11, 2008)

Mark Segal
Kurt Akeley

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

Copyright © 2006-2008 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	Formatting of Optional Features	1
1.2	What is the OpenGL Graphics System?	1
1.3	Programmer's View of OpenGL	2
1.4	Implementor's View of OpenGL	2
1.5	Our View	3
1.6	The Deprecation Model	3
1.7	Companion Documents	3
1.7.1	OpenGL Shading Language	3
1.7.2	Window System Bindings	3
2	OpenGL Operation	5
2.1	OpenGL Fundamentals	5
2.1.1	Floating-Point Computation	7
2.1.2	16-Bit Floating-Point Numbers	8
2.1.3	Unsigned 11-Bit Floating-Point Numbers	9
2.1.4	Unsigned 10-Bit Floating-Point Numbers	9
2.2	GL State	10
2.2.1	Shared Object State	11
2.3	GL Command Syntax	11
2.4	Basic GL Operation	13
2.5	GL Errors	15
2.6	Begin/End Paradigm	16
2.6.1	Begin and End	19
2.6.2	Polygon Edges	23
2.6.3	GL Commands within Begin/End	24
2.7	Vertex Specification	24
2.8	Vertex Arrays	28
2.9	Buffer Objects	38

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

3.4.2	Point Rasterization State	122
3.4.3	Point Multisample Rasterization	122
3.5	Line Segments	122
3.5.1	Basic Line Segment Rasterization	123
3.5.2	Other Line Segment Features	125
3.5.3	Line Rasterization State	128
3.5.4	Line Multisample Rasterization	128
3.6	Polygons	129
3.6.1	Basic Polygon Rasterization	129
3.6.2	Stippling	131
3.6.3	Antialiasing	132
3.6.4	Options Controlling Polygon Rasterization	132
3.6.5	Depth Offset	133
3.6.6	Polygon Multisample Rasterization	134
3.6.7	Polygon Rasterization State	135
3.7	Pixel Rectangles	135
3.7.1	Pixel Storage Modes and Pixel Buffer Objects	135
3.7.2	The Imaging Subset	137
3.7.3	Pixel Transfer Modes	137
3.7.4	Rasterization of Pixel Rectangles	148
3.7.5	Pixel Transfer Operations	162
3.7.6	Pixel Rectangle Multisample Rasterization	172
3.8	Bitmaps	173
3.9	Texturing	175
3.9.1	Texture Image Specification	176
3.9.2	Alternate Texture Image Specification Commands	191
3.9.3	Compressed Texture Images	196
3.9.4	Texture Parameters	200
3.9.5	Depth Component Textures	202
3.9.6	Cube Map Texture Selection	202
3.9.7	Texture Minification	203
3.9.8	Texture Magnification	212
3.9.9	Combined Depth/Stencil Textures	212
3.9.10	Texture Completeness	212
3.9.11	Texture State and Proxy State	214
3.9.12	Texture Objects	215
3.9.13	Texture Environments and Texture Functions	218
3.9.14	Texture Comparison Modes	221
3.9.15	sRGB Texture Color Conversion	224
3.9.16	Shared Exponent Texture Color Conversion	226

3.9.17	Texture Application	226
3.10	Color Sum	227
3.11	Fog	229
3.12	Fragment Shaders	230
3.12.1	Shader Variables	231
3.12.2	Shader Execution	232
3.13	Antialiasing Application	236
3.14	Multisample Point Fade	236
4	Per-Fragment Operations and the Framebuffer	238
4.1	Per-Fragment Operations	240
4.1.1	Pixel Ownership Test	240
4.1.2	Scissor Test	241
4.1.3	Multisample Fragment Operations	241
4.1.4	Alpha Test	242
4.1.5	Stencil Test	243
4.1.6	Depth Buffer Test	245
4.1.7	Occlusion Queries	245
4.1.8	Blending	246
4.1.9	sRGB Conversion	251
4.1.10	Dithering	251
4.1.11	Logical Operation	252
4.1.12	Additional Multisample Fragment Operations	253
4.2	Whole Framebuffer Operations	254
4.2.1	Selecting a Buffer for Writing	254
4.2.2	Fine Control of Buffer Updates	258
4.2.3	Clearing the Buffers	260
4.2.4	The Accumulation Buffer	263
4.3	Drawing, Reading, and Copying Pixels	264
4.3.1	Writing to the Stencil or Depth/Stencil Buffers	265
4.3.2	Reading Pixels	265
4.3.3	Copying Pixels	273
4.3.4	Pixel Draw/Read State	277
4.4	Framebuffer Objects	277
4.4.1	Binding and Managing Framebuffer Objects	278
4.4.2	Attaching Images to Framebuffer Objects	281
4.4.3	Rendering When an Image of a Bound Texture Object is Also Attached to the Framebuffer	288
4.4.4	Framebuffer Completeness	289

4.4.5	Effects of Framebuffer State on Framebuffer Dependent Values	294
4.4.6	Mapping between Pixel and Element in Attached Image	295
5	Special Functions	296
5.1	Evaluators	296
5.2	Selection	302
5.3	Feedback	304
5.4	Display Lists	306
5.5	Flush and Finish	311
5.6	Hints	312
6	State and State Requests	314
6.1	Querying GL State	314
6.1.1	Simple Queries	314
6.1.2	Data Conversions	315
6.1.3	Enumerated Queries	316
6.1.4	Texture Queries	322
6.1.5	Stipple Query	324
6.1.6	Color Matrix Query	324
6.1.7	Color Table Query	325
6.1.8	Convolution Query	325
6.1.9	Histogram Query	326
6.1.10	Minmax Query	327
6.1.11	Pointer and String Queries	328
6.1.12	Asynchronous Queries	329
6.1.13	Buffer Object Queries	331
6.1.14	Vertex Array Object Queries	332
6.1.15	Shader and Program Queries	332
6.1.16	Framebuffer Object Queries	337
6.1.17	Renderbuffer Object Queries	337
6.1.18	Saving and Restoring State	337
6.2	State Tables	340
A	Invariance	391
A.1	Repeatability	391
A.2	Multi-pass Algorithms	392
A.3	Invariance Rules	392
A.4	What All This Means	394

B	Corollaries	395
C	Compressed Texture Image Formats	398
C.1	RGTC Compressed Texture Image Formats	398
C.1.1	Format COMPRESSED_RED_RGTC1	399
C.1.2	Format COMPRESSED_SIGNED_RED_RGTC1	400
C.1.3	Format COMPRESSED_RG_RGTC2	401
C.1.4	Format COMPRESSED_SIGNED_RG_RGTC2	401
D	Shared Objects and Multiple Contexts	402
D.1	Object Deletion Behavior	402
E	The Deprecation Model	403
E.1	Profiles and Deprecated Features of OpenGL 3.0	403
F	Version 1.1	409
F.1	Vertex Array	409
F.2	Polygon Offset	410
F.3	Logical Operation	410
F.4	Texture Image Formats	410
F.5	Texture Replace Environment	410
F.6	Texture Proxies	411
F.7	Copy Texture and Subtexture	411
F.8	Texture Objects	411
F.9	Other Changes	411
F.10	Acknowledgements	412
G	Version 1.2	414
G.1	Three-Dimensional Texturing	414
G.2	BGRA Pixel Formats	414
G.3	Packed Pixel Formats	415
G.4	Normal Rescaling	415
G.5	Separate Specular Color	415
G.6	Texture Coordinate Edge Clamping	415
G.7	Texture Level of Detail Control	416
G.8	Vertex Array Draw Element Range	416
G.9	Imaging Subset	416
G.9.1	Color Tables	416
G.9.2	Convolution	417
G.9.3	Color Matrix	417

G.9.4	Pixel Pipeline Statistics	418
G.9.5	Constant Blend Color	418
G.9.6	New Blending Equations	418
G.10	Acknowledgements	418
H	Version 1.2.1	422
I	Version 1.3	423
I.1	Compressed Textures	423
I.2	Cube Map Textures	423
I.3	Multisample	424
I.4	Multitexture	424
I.5	Texture Add Environment Mode	425
I.6	Texture Combine Environment Mode	425
I.7	Texture Dot3 Environment Mode	425
I.8	Texture Border Clamp	425
I.9	Transpose Matrix	426
I.10	Acknowledgements	426
J	Version 1.4	431
J.1	Automatic Mipmap Generation	431
J.2	Blend Squaring	431
J.3	Changes to the Imaging Subset	432
J.4	Depth Textures and Shadows	432
J.5	Fog Coordinate	432
J.6	Multiple Draw Arrays	432
J.7	Point Parameters	433
J.8	Secondary Color	433
J.9	Separate Blend Functions	433
J.10	Stencil Wrap	433
J.11	Texture Crossbar Environment Mode	433
J.12	Texture LOD Bias	434
J.13	Texture Mirrored Repeat	434
J.14	Window Raster Position	434
J.15	Acknowledgements	434
K	Version 1.5	437
K.1	Buffer Objects	437
K.2	Occlusion Queries	438
K.3	Shadow Functions	438

K.4	Changed Tokens	438
K.5	Acknowledgements	438
L	Version 2.0	443
L.1	Programmable Shading	443
L.1.1	Shader Objects	443
L.1.2	Shader Programs	443
L.1.3	OpenGL Shading Language	444
L.1.4	Changes To Shader APIs	444
L.2	Multiple Render Targets	444
L.3	Non-Power-Of-Two Textures	444
L.4	Point Sprites	445
L.5	Separate Blend Equation	445
L.6	Separate Stencil	445
L.7	Other Changes	445
L.8	Acknowledgements	447
M	Version 2.1	449
M.1	OpenGL Shading Language	449
M.2	Non-Square Matrices	449
M.3	Pixel Buffer Objects	449
M.4	sRGB Textures	450
M.5	Other Changes	450
M.6	Acknowledgements	452
N	Version 3.0	455
N.1	New Features	455
N.2	Deprecation Model	456
N.3	Changed Tokens	457
N.4	Credits and Acknowledgements	457
O	ARB Extensions	460
O.1	Naming Conventions	460
O.2	Promoting Extensions to Core Features	461
O.3	Multitexture	461
O.4	Transpose Matrix	461
O.5	Multisample	461
O.6	Texture Add Environment Mode	462
O.7	Cube Map Textures	462
O.8	Compressed Textures	462

O.9 Texture Border Clamp	462
O.10 Point Parameters	462
O.11 Vertex Blend	462
O.12 Matrix Palette	463
O.13 Texture Combine Environment Mode	463
O.14 Texture Crossbar Environment Mode	463
O.15 Texture Dot3 Environment Mode	463
O.16 Texture Mirrored Repeat	463
O.17 Depth Texture	463
O.18 Shadow	463
O.19 Shadow Ambient	464
O.20 Window Raster Position	464
O.21 Low-Level Vertex Programming	464
O.22 Low-Level Fragment Programming	464
O.23 Buffer Objects	464
O.24 Occlusion Queries	465
O.25 Shader Objects	465
O.26 High-Level Vertex Programming	465
O.27 High-Level Fragment Programming	465
O.28 OpenGL Shading Language	465
O.29 Non-Power-Of-Two Textures	465
O.30 Point Sprites	466
O.31 Fragment Program Shadow	466
O.32 Multiple Render Targets	466
O.33 Rectangular Textures	466
O.34 Floating-Point Color Buffers	466
O.35 Half-Precision Floating Point	467
O.36 Floating-Point Textures	467
O.37 Pixel Buffer Objects	467
Index	468

List of Figures

2.1	Block diagram of the GL.	13
2.2	Creation of a processed vertex from a transformed vertex and current values.	17
2.3	Primitive assembly and processing.	19
2.4	Triangle strips, fans, and independent triangles.	21
2.5	Quadrilateral strips and independent quadrilaterals.	22
2.6	Vertex transformation sequence.	49
2.7	Current raster position.	71
2.8	Processing of RGBA colors.	73
2.9	Processing of color indices.	73
2.10	ColorMaterial operation.	83
3.1	Rasterization.	111
3.2	Rasterization of non-antialiased wide points.	118
3.3	Rasterization of antialiased wide points.	118
3.4	Visualization of Bresenham's algorithm.	123
3.5	Rasterization of non-antialiased wide lines.	126
3.6	The region used in rasterizing an antialiased line segment.	127
3.7	Operation of DrawPixels	148
3.8	Selecting a subimage from an image	153
3.9	A bitmap and its associated parameters.	173
3.10	A texture image and the coordinates used to access it.	189
3.11	Multitexture pipeline.	227
4.1	Per-fragment operations.	240
4.2	Operation of ReadPixels	265
4.3	Operation of CopyPixels	273
5.1	Map Evaluation.	298
5.2	Feedback syntax.	307

List of Tables

2.1	GL command suffixes	12
2.2	GL data types	14
2.3	Summary of GL errors	17
2.4	Vertex array sizes (values per vertex) and data types	30
2.5	Variables that direct the execution of InterleavedArrays	37
2.6	Buffer object parameters and their values.	39
2.7	Buffer object initial state.	41
2.8	Buffer object state set by MapBuffer	42
2.9	Transform feedback modes	64
2.10	Component conversions	74
2.11	Summary of lighting parameters.	76
2.12	Correspondence of lighting parameter symbols to names.	81
2.13	Polygon flatshading color selection.	86
3.1	PixelStore parameters.	136
3.2	PixelTransfer parameters.	138
3.3	PixelMap parameters.	139
3.4	Color table names.	140
3.5	DrawPixels and ReadPixels types.	151
3.6	DrawPixels and ReadPixels formats.	152
3.7	Swap Bytes bit ordering.	153
3.8	Packed pixel formats.	155
3.9	UNSIGNED_BYTE formats. Bit numbers are indicated for each component.	156
3.10	UNSIGNED_SHORT formats	157
3.11	UNSIGNED_INT formats	158
3.12	Packed pixel field assignments.	159
3.13	Color table lookup.	165
3.14	Computation of filtered color components.	166

3.15	Conversion from RGBA, depth, and stencil pixel components to internal texture, table, or filter components.	178
3.16	Sized internal color formats.	183
3.18	Sized internal depth and stencil formats.	184
3.17	Sized internal luminance and intensity formats.	184
3.19	Generic and specific compressed internal formats.	185
3.20	Texture parameters and their values.	201
3.21	Selection of cube map images.	202
3.22	Texel location wrap mode application.	206
3.23	Correspondence of filtered texture components to texture source components.	220
3.24	Texture functions <code>REPLACE</code> , <code>MODULATE</code> , and <code>DECAL</code>	220
3.25	Texture functions <code>BLEND</code> and <code>ADD</code>	221
3.26	<code>COMBINE</code> texture functions.	222
3.27	Arguments for <code>COMBINE_RGB</code> functions.	223
3.28	Arguments for <code>COMBINE_ALPHA</code> functions.	223
3.29	Depth texture comparison functions.	225
4.1	RGB and Alpha blend equations.	248
4.2	Blending functions.	250
4.3	Arguments to LogicOp and their corresponding operations.	253
4.4	Buffer selection for the default framebuffer	256
4.5	Buffer selection for a framebuffer object	256
4.6	DrawBuffers buffer selection for the default framebuffer	257
4.7	PixelStore parameters.	267
4.8	ReadPixels index masks.	271
4.9	ReadPixels GL data types and reversed component conversion formulas.	272
4.10	Effective ReadPixels format for <code>DEPTH_STENCIL</code> CopyPixels operation.	275
4.11	Correspondence of renderbuffer sized to base internal formats.	283
4.12	Framebuffer attachment points.	285
5.1	Values specified by the <i>target</i> to Map1	297
5.2	Correspondence of feedback type to number of values per vertex.	306
5.3	Hint targets and descriptions	313
6.1	Texture, table, and filter return values.	323
6.2	Attribute groups	339
6.3	State Variable Types	341

6.4	GL Internal begin-end state variables (inaccessible)	342
6.5	Current Values and Associated Data	343
6.6	Vertex Array Object State	344
6.7	Vertex Array Object State (cont.)	345
6.8	Vertex Array Object State (cont.)	346
6.9	Vertex Array Object State (cont.)	347
6.10	Vertex Array Data (not in Vertex Array objects)	348
6.11	Buffer Object State	349
6.12	Transformation state	350
6.13	Coloring	351
6.14	Lighting (see also table 2.11 for defaults)	352
6.15	Lighting (cont.)	353
6.16	Rasterization	354
6.17	Rasterization (cont.)	355
6.18	Multisampling	356
6.19	Textures (state per texture unit and binding point)	357
6.20	Textures (state per texture object)	358
6.21	Textures (state per texture image)	359
6.22	Texture Environment and Generation	360
6.23	Texture Environment and Generation (cont.)	361
6.24	Pixel Operations	362
6.25	Pixel Operations (cont.)	363
6.26	Framebuffer Control	364
6.27	Framebuffer (state per target binding point)	365
6.28	Framebuffer (state per framebuffer object)	366
6.29	Framebuffer (state per attachment point)	367
6.30	Renderbuffer (state per target and binding point)	368
6.31	Renderbuffer (state per renderbuffer object)	369
6.32	Pixels	370
6.33	Pixels (cont.)	371
6.34	Pixels (cont.)	372
6.35	Pixels (cont.)	373
6.36	Pixels (cont.)	374
6.37	Pixels (cont.)	375
6.38	Evaluators (GetMap takes a map name)	376
6.39	Shader Object State	377
6.40	Program Object State	378
6.41	Program Object State (cont.)	379
6.42	Vertex Shader State	380
6.43	Query Object State	381

6.44 Transform Feedback State	382
6.45 Hints	383
6.46 Implementation Dependent Values	384
6.47 Implementation Dependent Values (cont.)	385
6.48 Implementation Dependent Values (cont.)	386
6.49 Implementation Dependent Values (cont.)	387
6.50 Implementation Dependent Values (cont.)	388
6.51 Framebuffer Dependent Values	389
6.52 Miscellaneous	390
K.1 New token names	439
N.1 New token names	457

Chapter 1

Introduction

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

1.1 Formatting of Optional Features

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

Portions of the specification which are optional are so described where the optional features are first defined (see section 3.7.2). State table entries which are optional are typeset against a gray background.

1.2 What is the OpenGL Graphics System?

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

Most of OpenGL requires that the graphics hardware contain a framebuffer. Many OpenGL calls pertain to drawing objects such as points, lines, polygons, and bitmaps, but the way that some of this drawing occurs (such as when antialiasing

or texturing is enabled) relies on the existence of a framebuffer. Further, some of OpenGL is specifically concerned with framebuffer manipulation.

1.3 Programmer's View of OpenGL

To the programmer, OpenGL is a set of commands that allow the specification of geometric objects in two or three dimensions, together with commands that control how these objects are rendered into the framebuffer. For the most part, OpenGL provides an immediate-mode interface, meaning that specifying an object causes it to be drawn.

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

1.4 Implementor's View of OpenGL

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

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

1.5 Our View

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

1.6 The Deprecation Model

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

1.7 Companion Documents

1.7.1 OpenGL Shading Language

This specification should be read together with a companion document titled *The OpenGL Shading Language*. The latter document (referred to as the OpenGL Shading Language Specification hereafter) defines the syntax and semantics of the programming language used to write vertex and fragment shaders (see sections 2.20 and 3.12). These sections may include references to concepts and terms (such as shading language variable types) defined in the companion document.

OpenGL 3.0 implementations are guaranteed to support at least versions 1.10, 1.20, and 1.30 of the shading language, although versions 1.10 and 1.20 are deprecated in a forward-compatible context. The actual version supported may be queried as described in section 6.1.11.

1.7.2 Window System Bindings

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

OpenGL Graphics with the X Window System, also called the “GLX Specification”, describes the GLX API for use of OpenGL in the X Window System. It is

primarily directed at Linux and Unix systems, but GLX implementations also exist for Microsoft Windows, MacOS X, and some other platforms where X is available. The GLX Specification is available in the OpenGL Extension Registry (see appendix O).

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

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

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

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

Chapter 2

OpenGL Operation

2.1 OpenGL Fundamentals

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

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

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

Commands are always processed in the order in which they are received, although there may be an indeterminate delay before the effects of a command are realized. This means, for example, that one primitive must be drawn completely before any subsequent one can affect the framebuffer. It also means that queries and pixel read operations return state consistent with complete execution of all

previously invoked GL commands, except where explicitly specified otherwise. In general, the effects of a GL command on either GL modes or the framebuffer must be complete before any subsequent command can have any such effects.

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

The GL provides direct control over the fundamental operations of 3D and 2D graphics. This includes specification of such parameters as vertex and fragment shaders, transformation matrices, lighting equation coefficients, antialiasing methods, and pixel update operators. It does not provide a means for describing or modeling complex geometric objects. Another way to describe this situation is to say that the GL provides mechanisms to describe how complex geometric objects are to be rendered rather than mechanisms to describe the complex objects themselves.

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

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

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

Allocation and configuration of the default framebuffer occurs outside of the GL in conjunction with the window system, using companion APIs such as GLX,

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

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

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

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

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

2.1.1 Floating-Point Computation

The GL must perform a number of floating-point operations during the course of its operation. In some cases, the representation and/or precision of such operations is defined or limited; by the OpenGL Shading Language Specification for operations in shaders, and in some cases implicitly limited by the specified format of vertex, texture, or renderbuffer data consumed by the GL. Otherwise, the representation of such floating-point numbers, and the details of how operations on them are performed, is not specified. We require simply that numbers' floating-point parts contain enough bits and that their exponent fields are large enough so that individual results of floating-point operations are accurate to about 1 part in 10^5 . The maximum representable magnitude of a floating-point number used to represent positional, normal, or texture coordinates must be at least 2^{32} ; the maximum representable magnitude for colors must be at least 2^{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 *Inf*s and *NaN*s in their floating-point computations.

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

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

2.1.2 16-Bit Floating-Point Numbers

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

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

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

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

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

2.1.3 Unsigned 11-Bit Floating-Point Numbers

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

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

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

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

$$M = N \bmod 64.$$

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:

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

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

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

$$M = N \bmod 32.$$

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.

2.2 GL State

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

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

complete set of GL server state; each connection from a client to a server implies a set of both GL client state and GL server state.

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

2.2.1 Shared Object State

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

2.3 GL Command Syntax

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

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

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

and

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

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

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

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.

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

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 char` is abbreviated `uchar`).

For example,

```
void Normal3{fd}( T arg );
```

indicates the two declarations

```
void Normal3f( float arg1, float arg2, float arg3 );
void Normal3d( double arg1, double arg2, double arg3 );
```

while

```
void Normal3{fd}v( T arg );
```

means the two declarations

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

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

2.4 Basic GL Operation

Figure 2.1 shows a schematic diagram of the GL. Commands enter the GL on the left. Some commands specify geometric objects to be drawn while others control how the objects are handled by the various stages. Most commands may be accumulated in a *display list* for processing by the GL at a later time. Otherwise, commands are effectively sent through a processing pipeline.

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

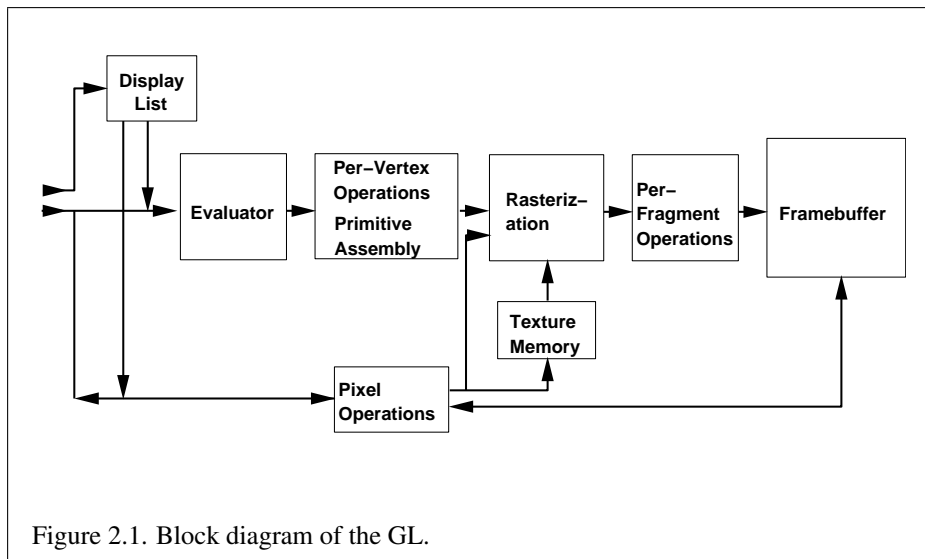
Finally, there is a way to bypass the vertex processing portion of the pipeline to send a block of fragments directly to the individual fragment operations, eventually causing a block of pixels to be written to the framebuffer; values may also be read back from the framebuffer or copied from one portion of the framebuffer to another. These transfers may include some type of decoding or encoding.

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

GL 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	<i>ptrbits</i>	Signed 2's complement binary integer
sizeiptr	<i>ptrbits</i>	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 $[0, 1]$
time	64	Unsigned binary representing an absolute absolute or relative time interval. Precision is nanoseconds but accuracy is implementation-dependent.

Table 2.2: GL data types. GL types are not C types. Thus, for example, GL type `int` is referred to as `GLint` outside this document, and is not necessarily equivalent to the C type `int`. An implementation may use more bits than the number indicated in the table to represent a GL type. Correct interpretation of integer values outside the minimum range is not required, however.

ptrbits is the number of bits required to represent a pointer type; in other words, types `intptr` and `sizeiptr` must be sufficiently large as to store any address.



2.5 GL Errors

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

The command

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

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

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

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

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

- If a command that requires an enumerated value is passed a symbolic constant that is not one of those specified as allowable for that command, the error `INVALID_ENUM` is generated. This is the case even if the argument is a pointer to a symbolic constant, if the value pointed to is not allowable for the given command.
- If a negative number is provided where an argument of type `sizei` or `sizeiptr` is specified, the error `INVALID_VALUE` is generated.
- If memory is exhausted as a side effect of the execution of a command, the error `OUT_OF_MEMORY` may be generated.

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

2.6 Begin/End Paradigm

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

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

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

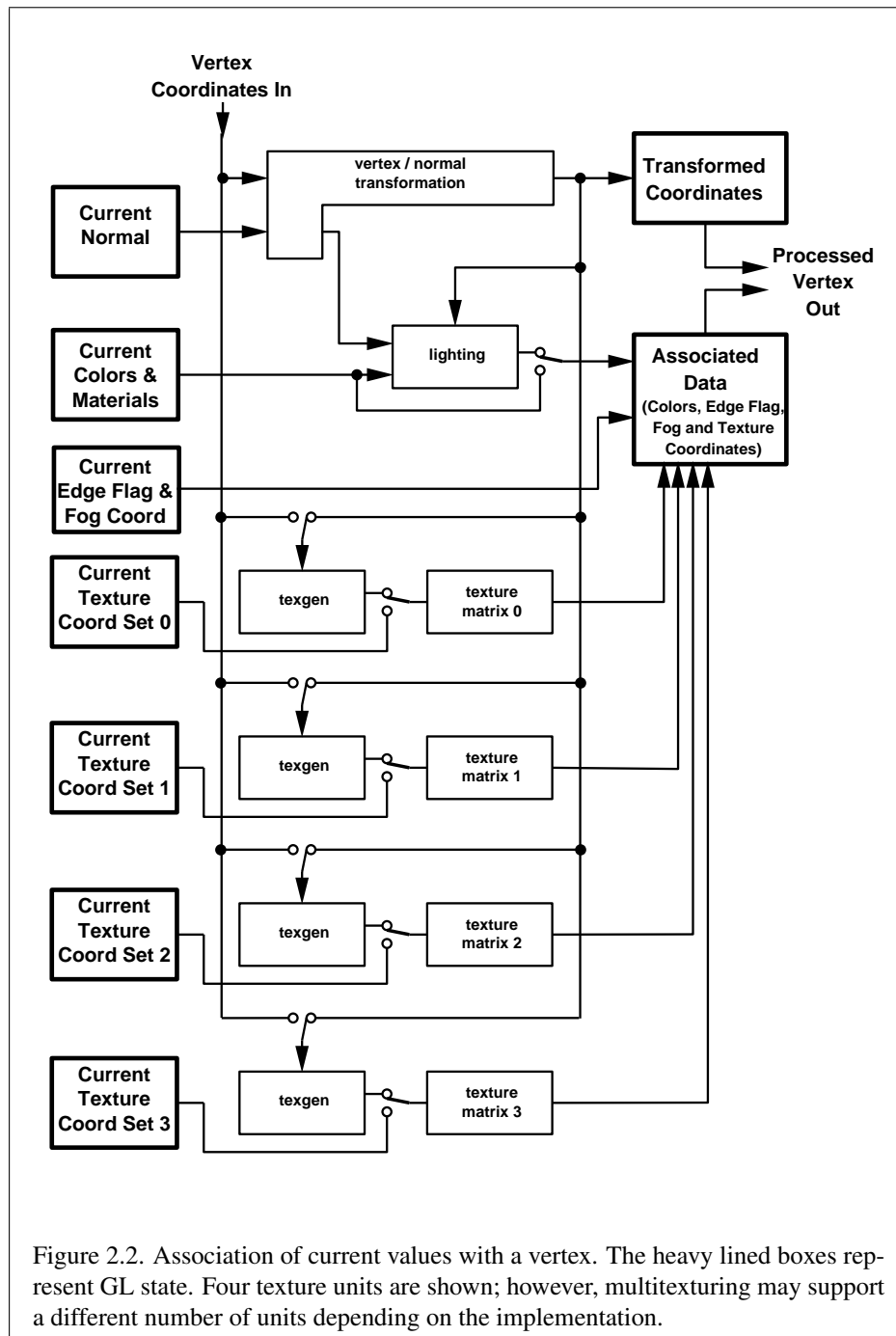
Table 2.3: Summary of GL errors

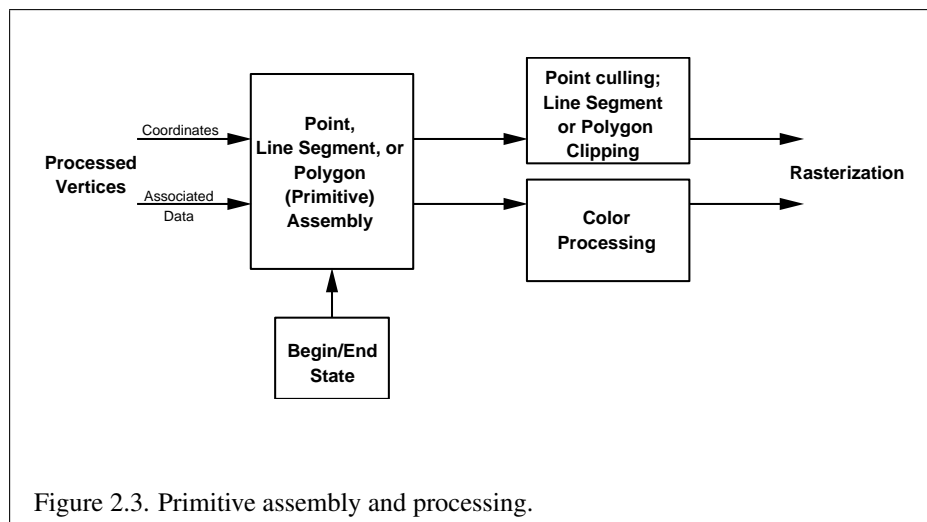
a texture image is mapped onto a primitive. Multiple sets of texture coordinates may be used to specify how multiple texture images are mapped onto a primitive. The number of texture units supported is implementation dependent but must be at least two. The number of texture units supported can be queried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

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

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

The methods by which vertices, normals, texture coordinates, fog coordinate, generic attributes, and colors are sent to the GL, as well as how normals are trans-





formed and how vertices are mapped to the two-dimensional screen, are discussed later.

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

Figure 2.3 shows the sequence of operations that builds a *primitive* (point, line segment, or polygon) from a sequence of vertices. After a primitive is formed, it is clipped to a viewing volume. This may alter the primitive by altering vertex coordinates, texture coordinates, and colors. In the case of line and polygon primitives, clipping may insert new vertices into the primitive. The vertices defining a primitive to be rasterized have texture coordinates and colors associated with them.

2.6.1 Begin and End

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

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

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

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

Line Strips. A series of one or more connected line segments is specified by enclosing a series of two or more endpoints within a **Begin/End** pair when **Begin** is called with `LINE_STRIP`. In this case, the first vertex specifies the first segment's start point while the second vertex specifies the first segment's endpoint and the second segment's start point. In general, the i th vertex (for $i > 1$) specifies the beginning of the i th segment and the end of the $i - 1$ st. The last vertex specifies the end of the last segment. If only one vertex is specified between the **Begin/End** pair, then no primitive is generated.

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

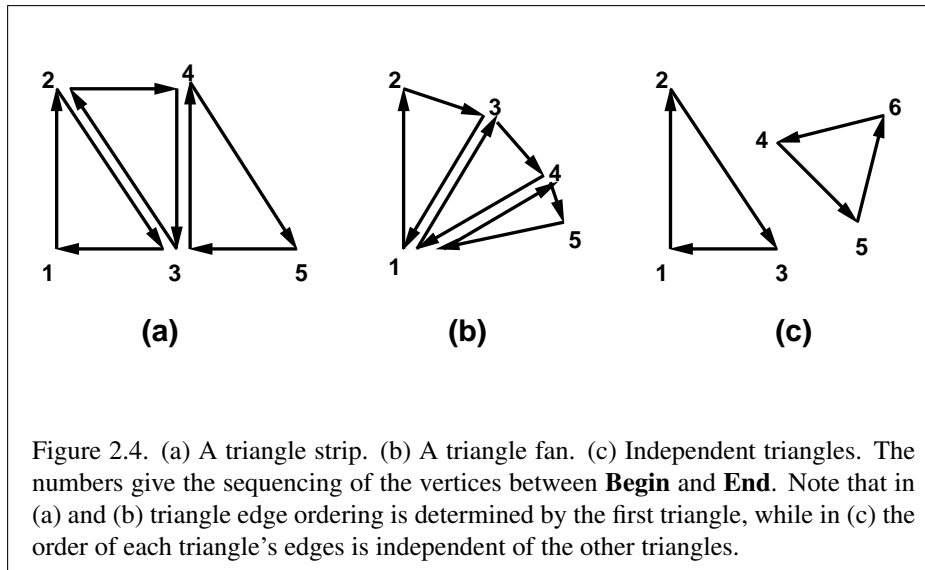
Line Loops. Line loops, specified with the `LINE_LOOP` argument value to **Begin**, are the same as line strips except that a final segment is added from the final specified vertex to the first vertex. The additional state consists of the processed first vertex.

Separate Lines. Individual line segments, each specified by a pair of vertices, are generated by surrounding vertex pairs with **Begin** and **End** when the value of the argument to **Begin** is `LINES`. In this case, the first two vertices between a **Begin** and **End** pair define the first segment, with subsequent pairs of vertices each defining one more segment. If the number of specified vertices is odd, then the last one is ignored. The state required is the same as for lines but it is used differently: a vertex holding the first vertex of the current segment, and a boolean flag indicating whether the current vertex is odd or even (a segment start or end).

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

Only convex polygons are guaranteed to be drawn correctly by the GL. If a specified polygon is nonconvex when projected onto the window, then the rendered polygon need only lie within the convex hull of the projected vertices defining its boundary.

The state required to support polygons consists of at least two processed vertices (more than two are never required, although an implementation may use

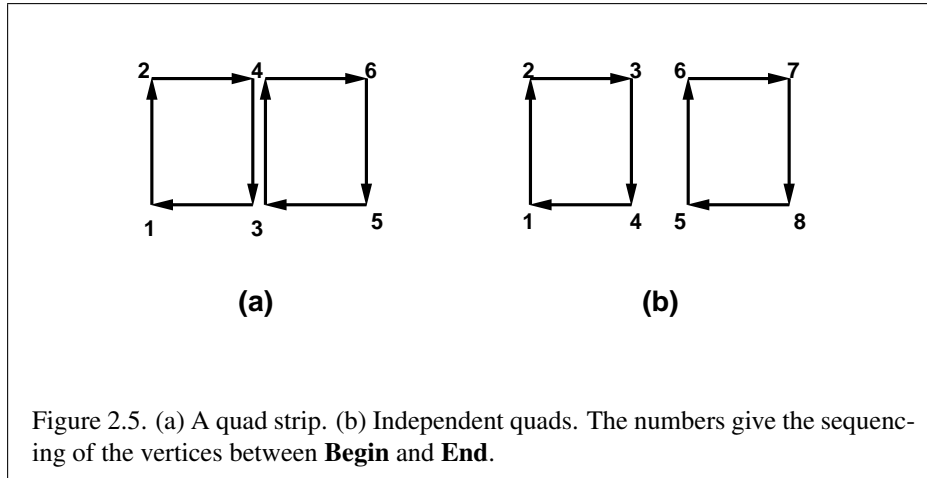


more); this is because a convex polygon can be rasterized as its vertices arrive, before all of them have been specified. The order of the vertices is significant in lighting and polygon rasterization (see sections 2.19.1 and 3.6.1).

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

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

Triangle fans. A triangle fan is the same as a triangle strip with one exception: each vertex after the first always replaces vertex B of the two stored vertices. The



vertices of a triangle fan are enclosed between **Begin** and **End** when the value of the argument to **Begin** is `TRIANGLE_FAN`.

Separate Triangles. Separate triangles are specified by placing vertices between **Begin** and **End** when the value of the argument to **Begin** is `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 between the **Begin** and **End**. k is either 0, 1, or 2; if k is not zero, the final k vertices are ignored. For each triangle, vertex A is vertex $3i$ and vertex B is vertex $3i + 1$. Otherwise, separate triangles are the same as a triangle strip.

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

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

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

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

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

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

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

2.6.2 Polygon Edges

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

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

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

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

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

2.6.3 GL Commands within Begin/End

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

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

2.7 Vertex Specification

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

```
void Vertex{234}{sifd}( T coords );
void Vertex{234}{sifd}v( T coords );
```

A call to any **Vertex** command specifies four coordinates: x , y , z , and w . The x coordinate is the first coordinate, y is second, z is third, and w is fourth. A call to **Vertex2** sets the x and y coordinates; the z coordinate is implicitly set to zero and the w coordinate to one. **Vertex3** sets x , y , and z to the provided values and w to one. **Vertex4** sets all four coordinates, allowing the specification of an arbitrary point in projective three-space. Invoking a **Vertex** command outside of a **Begin/End** pair results in undefined behavior.

Current values are used in associating auxiliary data with a vertex as described in section 2.6. A current value may be changed at any time by issuing an appropriate command. The commands

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

specify the current homogeneous texture coordinates, named s , t , r , and q . The **TexCoord1** family of commands set the s coordinate to the provided single argument while setting t and r to 0 and q to 1. Similarly, **TexCoord2** sets s and t to the specified values, r to 0 and q to 1; **TexCoord3** sets s , t , and r , with q set to 1, and **TexCoord4** sets all four texture coordinates.

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

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

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

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

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

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

The current normal is set using

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

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

The current fog coordinate is set using

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

There are several ways to set the current color and secondary color. The GL stores a current single-valued *color index*, as well as a current four-valued RGBA color and secondary color. Either the index or the color and secondary color are significant depending as the GL is in *color index mode* or *RGBA mode*. The mode selection is made when the GL is initialized.

The commands to set RGBA colors are

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

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

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

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

The command

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

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

Vertex shaders (see section 2.20) can be written to access an array of 4-component generic vertex attributes in addition to the conventional attributes specified previously. The first slot of this array is numbered 0, and the size of the array is specified by the implementation-dependent constant `MAX_VERTEX_ATTRIBS`.

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

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

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

The **VertexAttrib4N*** commands specify fixed-point values that are converted to a normalized $[0, 1]$ or $[-1, 1]$ range as shown in table 2.10, while the other commands specify values that are converted directly to the internal floating-point 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 values into a generic shader attribute declared as a signed or unsigned scalar or vector, use the commands

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

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

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

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

Setting generic vertex attribute zero specifies a vertex; the four vertex coordinates are taken from the values of attribute zero. A **Vertex2**, **Vertex3**, or **Vertex4**

command is completely equivalent to the corresponding **VertexAttrib*** command with an *index* of zero. Setting any other generic vertex attribute updates the current values of the attribute. There are no current values for vertex attribute zero.

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

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

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

2.8 Vertex Arrays

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

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

```

void ColorPointer( int size, enum type, sizei stride,
    void *pointer );
void SecondaryColorPointer( int size, enum type,
    sizei stride, void *pointer );
void IndexPointer( enum type, sizei stride, void *pointer );
void EdgeFlagPointer( sizei stride, void *pointer );
void FogCoordPointer( enum type, sizei stride,
    void *pointer );
void TexCoordPointer( int size, enum type, sizei stride,
    void *pointer );
void VertexAttribPointer( uint index, int size, enum type,
    boolean normalized, sizei stride, const
    void *pointer );
void VertexAttribIPointer( uint index, int size, enum type,
    sizei stride, const void *pointer );

```

describe the locations and organizations of these arrays. For each command, *type* specifies the data type of the values stored in the array. Because edge flags are always type `boolean`, **EdgeFlagPointer** has no *type* argument. *size*, when present, indicates the number of values per vertex that are stored in the array. Because normals are always specified with three values, **NormalPointer** has no *size* argument. Likewise, because color indices and edge flags are always specified with a single value, **IndexPointer** and **EdgeFlagPointer** also have no *size* argument. Table 2.4 indicates the allowable values for *size* and *type* (when present). For *type* the values `BYTE`, `SHORT`, `INT`, `FLOAT`, `HALF_FLOAT`, and `DOUBLE` indicate types `byte`, `short`, `int`, `float`, `half`, and `double`, respectively; and the values `UNSIGNED_BYTE`, `UNSIGNED_SHORT`, and `UNSIGNED_INT` indicate types `ubyte`, `ushort`, and `uint`, respectively. The error `INVALID_VALUE` is generated if *size* is specified with a value other than that indicated in the table.

The *index* parameter in the **VertexAttribPointer** and **VertexAttribIPointer** commands identify the generic vertex attribute array being described. The error `INVALID_VALUE` is generated if *index* is greater than or equal to `MAX_VERTEX_ATTRIBS`. Generic attribute arrays with integer *type* arguments can be handled in one of three ways: converted to float by normalizing to $[0, 1]$ or $[-1, 1]$ as specified in table 2.10, converted directly to float, or left as integers. Data for an array specified by **VertexAttribPointer** will be converted to floating-point by normalizing if *normalized* is `TRUE`, and converted directly to floating-point otherwise. Data for an array specified by **VertexAttribIPointer** will always be left as integer values; such data are referred to as *pure* integers.

Command	Sizes	Integer Handling	Types
VertexPointer	2,3,4	cast	short, int, float, half, double
NormalPointer	3	normalize	byte, short, int, float, half, double
ColorPointer	3,4	normalize	byte, ubyte, short, ushort, int, uint, float, half, double
SecondaryColorPointer	3	normalize	byte, ubyte, short, ushort, int, uint, float, half, double
IndexPointer	1	cast	ubyte, short, int, float, double
FogCoordPointer	1	n/a	float, half, double
TexCoordPointer	1,2,3,4	cast	short, int, float, half, double
EdgeFlagPointer	1	integer	boolean
VertexAttribPointer	1,2,3,4	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: “cast” means that they converted to floating-point directly, “normalize” means that they are converted to floating-point by normalizing to $[0, 1]$ (for unsigned types) or $[-1, 1]$ (for signed types), “integer” means that they remain as integer values, and “flag” means that either “cast” or “normalized” applies, depending on the setting of the *normalized* flag in **VertexAttribPointer**.

The one, two, three, or four values in an array that correspond to a single vertex comprise an array *element*. The values within each array element are stored sequentially in memory. If *stride* is specified as zero, then array elements are stored sequentially as well. The error `INVALID_VALUE` is generated if *stride* is negative. Otherwise pointers to the *i*th and (*i* + 1)st elements of an array differ by *stride* basic machine units (typically unsigned bytes), the pointer to the (*i* + 1)st element being greater. For each command, *pointer* specifies the location in memory of the first value of the first element of the array being specified.

An individual array is enabled or disabled by calling one of

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

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

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

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

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

The command

```
void ClientActiveTexture( enum texture );
```

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

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

The command

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

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

```

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



```

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

```

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

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

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

The command

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

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

```
DrawArrays (mode, first, count) ;
```

is the same as the effect of the command sequence

```

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

```

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

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

The command

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

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

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

The command

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

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

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

is the same as the effect of the command sequence

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

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

The command

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

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

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

The command

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

is a restricted form of **DrawElements**. *mode*, *count*, *type*, and *indices* match the corresponding arguments to **DrawElements**, with the additional constraint that all values in the array *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 command

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

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

The effect of

```
InterleavedArrays ( format, stride, pointer ) ;
```

is the same as the effect of the command sequence

```
if ( format or stride is invalid )
    generate appropriate error
else {
    int str;
    set  $e_t, e_c, e_n, s_t, s_c, s_v, t_c, p_c, p_n, p_v$ , and  $s$  as a function
        of table 2.5 and the value of format.
    str = stride;
    if (str is zero)
        str =  $s$ ;
    DisableClientState (EDGE_FLAG_ARRAY);
    DisableClientState (INDEX_ARRAY);
    DisableClientState (SECONDARY_COLOR_ARRAY);
    DisableClientState (FOG_COORD_ARRAY);
    if (  $e_t$  ) {
        EnableClientState (TEXTURE_COORD_ARRAY);
        TexCoordPointer (  $s_t$ , FLOAT, str, pointer );
    } else
        DisableClientState (TEXTURE_COORD_ARRAY);
    if (  $e_c$  ) {
        EnableClientState (COLOR_ARRAY);
        ColorPointer (  $s_c, t_c$ , str, pointer +  $p_c$  );
    } else
        DisableClientState (COLOR_ARRAY);
    if (  $e_n$  ) {
        EnableClientState (NORMAL_ARRAY);
        NormalPointer (FLOAT, str, pointer +  $p_n$  );
    } else
```

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

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

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

```

    DisableClientState (NORMAL_ARRAY);
    EnableClientState (VERTEX_ARRAY);
    VertexPointer ( $s_v$ , FLOAT,  $\text{str}$ ,  $\text{pointer} + p_v$ );
}

```

If the number of supported texture units (the value of `MAX_TEXTURE_COORDS`) is m and the number of supported generic vertex attributes (the value of `MAX_VERTEX_ATTRIBS`) is n , then the client state required to implement vertex arrays consists of an integer for the client active texture unit selector, $7 + m + n$ boolean values, $7 + m + n$ memory pointers, $7 + m + n$ integer stride values, $7 + m + n$ symbolic constants representing array types, $3 + m + n$ integers representing values per element, n boolean values indicating normalization, and n boolean values indicating whether the attribute values are pure integers.

In the initial state, the client active texture unit selector is `TEXTURE0`, the boolean values are each false, the memory pointers are each `NULL`, the strides are each zero, the array types are each `FLOAT`, the integers representing values per element are each four, and the normalized and pure integer flags are each false.

2.9 Buffer Objects

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

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

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

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

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

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_MAPPED	boolean	FALSE	TRUE, FALSE
BUFFER_MAP_POINTER	void*	NULL	address

Table 2.6: Buffer object parameters and their values.

While a buffer object is bound, GL operations on the target to which it is bound affect the bound buffer object, and queries of the target to which a buffer object is bound return state from the bound object.

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

Buffer objects are deleted by calling

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

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

The command

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

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

While a buffer object is bound, any GL operations on that object affect any other bindings of that object. If a buffer object is deleted while it is bound, all bindings to that object in the current context (i.e. in the thread that called **DeleteBuffers**) are reset to zero. Bindings to that buffer in other contexts and other threads are not affected, but attempting to use a deleted buffer in another thread

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

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

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

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

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

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

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

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

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

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

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

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

Name	Value
BUFFER_SIZE	<i>size</i>
BUFFER_USAGE	<i>usage</i>
BUFFER_ACCESS	READ_WRITE
BUFFER_MAPPED	FALSE
BUFFER_MAP_POINTER	NULL

Table 2.7: Buffer object initial state.

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

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

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

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

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

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

Name	Value
BUFFER_ACCESS	<i>access</i>
BUFFER_MAPPED	TRUE
BUFFER_MAP_POINTER	pointer to the data store

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

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

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

Non-NULL pointers returned by **MapBuffer** may be used by the client to modify and query buffer object data, consistent with the access rules of the mapping, while the mapping remains valid. No GL error is generated if the pointer is used to attempt to modify a `READ_ONLY` data store, or to attempt to read from a `WRITE_ONLY` data store, but operation may be slow and system errors (possibly including program termination) may result. Pointer values returned by **MapBuffer** may not be passed as parameter values to GL commands. For example, they may not be used to specify array pointers, or to specify or query pixel or texture image data; such actions produce undefined results, although implementations may not check for such behavior for performance reasons.

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

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

After the client has specified the contents of a mapped data store, and before

the data in that store are dereferenced by any GL commands, the mapping must be relinquished by calling

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

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

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

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

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

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

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

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

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

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

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

- `MAP_INVALIDATE_RANGE_BIT` indicates that the previous contents of the specified range may be discarded. Data within this range are undefined with the exception of subsequently written data. No GL error is generated if subsequent GL operations access unwritten data, but the result is undefined and system errors (possibly including program termination) may occur. This flag may not be used in combination with `MAP_READ_BIT`.
- `MAP_INVALIDATE_BUFFER_BIT` indicates that the previous contents of the entire buffer may be discarded. Data within the entire buffer are undefined with the exception of subsequently written data. No GL error is generated if subsequent GL operations access unwritten data, but the result is undefined and system errors (possibly including program termination) may occur. This flag may not be used in combination with `MAP_READ_BIT`.
- `MAP_FLUSH_EXPLICIT_BIT` indicates that one or more discrete subranges of the mapping may be modified. When this flag is set, modifications to each subrange must be explicitly flushed by calling **FlushMappedBufferRange**. No GL error is set if a subrange of the mapping is modified and not flushed, but data within the corresponding subrange of the buffer are undefined. This flag may only be used in conjunction with `MAP_WRITE_BIT`. When this option is selected, flushing is strictly limited to regions that are explicitly indicated with calls to **FlushMappedBufferRange** prior to `unmap`; if this option is not selected **UnmapBuffer** will automatically flush the entire mapped range when called.
- `MAP_UNSYNCHRONIZED_BIT` indicates that the GL should not attempt to synchronize pending operations on the buffer prior to returning from **MapBufferRange**. No GL error is generated if pending operations which source or modify the buffer overlap the mapped region, but the result of such previous and any subsequent operations is undefined.

Errors

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

An `INVALID_VALUE` error is generated if *offset* or *length* is negative, if *offset* + *length* 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.

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

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

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

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

Errors

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

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

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

2.9.1 Vertex Arrays in Buffer Objects

Blocks of vertex array data may be stored in buffer objects with the same format and layout options supported for client-side vertex arrays. However, it is expected

that GL implementations will (at minimum) be optimized for data with all components represented as floats, as well as for color data with components represented as either floats or unsigned bytes.

A buffer object binding point is added to the client state associated with each vertex array type. The commands that specify the locations and organizations of vertex arrays copy the buffer object name that is bound to `ARRAY_BUFFER` to the binding point corresponding to the vertex array of the type being specified. For example, the **NormalPointer** command copies the value of `ARRAY_BUFFER_BINDING` (the queriable name of the buffer binding corresponding to the target `ARRAY_BUFFER`) to the client state variable `NORMAL_ARRAY_BUFFER_BINDING`.

Rendering commands **ArrayElement**, **DrawArrays**, **DrawElements**, **DrawRangeElements**, **MultiDrawArrays**, and **MultiDrawElements** operate as previously defined, except that data for enabled vertex and attrib arrays are sourced from buffers if the array's buffer binding is non-zero. When an array is sourced from a buffer object, the pointer value of that array is used to compute an offset, in basic machine units, into the data store of the buffer object. This offset is computed by subtracting a null pointer from the pointer value, where both pointers are treated as pointers to basic machine units.

It is acceptable for vertex or attrib arrays to be sourced from any combination of client memory and various buffer objects during a single rendering operation.

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

2.9.2 Array Indices in Buffer Objects

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

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

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

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

Buffer objects created by binding an unused name to `ARRAY_BUFFER` and to `ELEMENT_ARRAY_BUFFER` are formally equivalent, but the GL may make different choices about storage implementation based on the initial binding. In some cases performance will be optimized by storing indices and array data in separate buffer objects, and by creating those buffer objects with the corresponding binding points.

2.9.3 Buffer Object State

The state required to support buffer objects consists of binding names for the array buffer, element buffer, pixel unpack buffer, and pixel pack buffer. Additionally, each vertex array has an associated binding so there is a buffer object binding for each of the vertex array, normal array, color array, index array, multiple texture coordinate arrays, edge flag array, secondary color array, fog coordinate array, and vertex attribute arrays. The initial values for all buffer object bindings is zero.

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

2.10 Vertex Array Objects

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

The command

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

returns *n* previous unused vertex array object names in *arrays*. These names are marked as used, for the purposes of **GenVertexArrays** only, and are initialized with the state listed in tables 6.6 through 6.9.

Vertex array objects are deleted by calling

```
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 6.6 through 6.9.

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 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 **VertexAttribPointer** or **VertexAttribIPointer** is called while a non-zero vertex array object is bound and zero is bound to the `ARRAY_BUFFER` buffer object binding point ².

2.11 Rectangles

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

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

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

```
Rect (  $x_1, y_1, x_2, y_2$  ) ;
```

is exactly the same as the following sequence of commands:

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


```

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

```

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

2.12 Coordinate Transformations

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

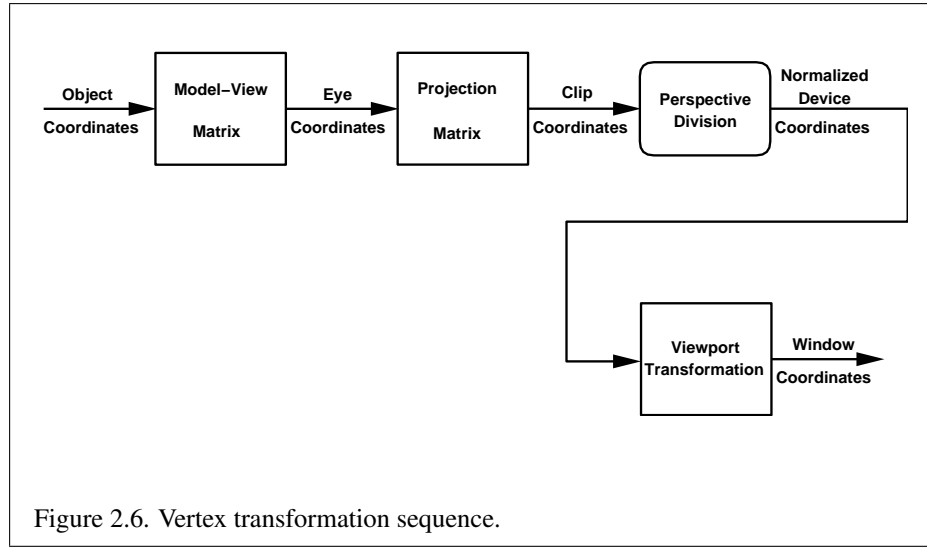
Vertices, normals, and texture coordinates are transformed before their coordinates are used to produce an image in the framebuffer. We begin with a description of how vertex coordinates are transformed and how this transformation is controlled.

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

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

If a vertex in object coordinates is given by $\begin{pmatrix} x_o \\ y_o \\ z_o \\ w_o \end{pmatrix}$ and the model-view matrix is M , then the vertex's eye coordinates are found as

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



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

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

The vertex's normalized device coordinates are then

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

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  $n$ , clampdf  $f$  );
```

z_w is represented as either fixed- or floating-point depending on whether the frame-buffer'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 $k \in \{0, 1, \dots, 2^m - 1\}$, 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 $o_x = x + w/2$ and $o_y = y + h/2$; $p_x = w$, $p_y = h$.

Viewport width and height are clamped to implementation-dependent maximums when specified. The maximum width and height may be found by issuing an appropriate **Get** command (see chapter 6). The maximum viewport dimensions must be greater than or equal to the visible dimensions of the display being rendered to. `INVALID_VALUE` is generated if either w or h is negative.

The state required to implement the viewport transformation is four integers and two clamped floating-point values. In the initial state, w and h are set to the width and height, respectively, of the window into which the GL is to do its rendering. o_x and o_y are set to $w/2$ and $h/2$, respectively. n and f are set to 0.0 and 1.0, respectively.

2.12.2 Matrices

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

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

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

The two basic commands for affecting the current matrix are

```
void LoadMatrix{fd}( T m[16] );
void MultMatrix{fd}( T m[16] );
```

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

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

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

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

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

The commands

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

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

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

The effect of

LoadTransposeMatrix[fd] (m) ;

is the same as the effect of

LoadMatrix[fd] (m^T) ;

The effect of

MultTransposeMatrix[fd] (m) ;

is the same as the effect of

MultMatrix[fd] (m^T) ;

The command

`void LoadIdentity(void);`

effectively calls **LoadMatrix** with the identity matrix:

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

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

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

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

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

Let $\mathbf{u} = \mathbf{v}/\|\mathbf{v}\| = (x' \ y' \ z')^T$. If

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

then

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

The arguments to

```
void Translate{fd}(T x, T y, T z);
```

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

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

```
void Scale{fd}(T x, T y, T z);
```

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

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

For

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

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

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

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

describes a matrix that produces parallel projection. $(l \ b \ -n)^T$ and $(r \ t \ -n)^T$ specify the points on the near clipping plane that are mapped to the lower left and upper right corners of the window, respectively. f gives the distance from the eye

to the far clipping plane. If l is equal to r , b is equal to t , or n is equal to f , the error `INVALID_VALUE` results. The corresponding matrix is

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

For each texture coordinate set, a 4×4 matrix is applied to the corresponding texture coordinates. This matrix is applied as

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

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

The command

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

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

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

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

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

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

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

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

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

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

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

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

The state required to implement transformations consists of an integer for the active texture unit selector, a four-valued integer indicating the current matrix

mode, one stack of at least two 4×4 matrices for each of `COLOR`, `PROJECTION`, and each texture coordinate set, `TEXTURE`; and a stack of at least 32 4×4 matrices for `MODELVIEW`. Each matrix stack has an associated stack pointer. Initially, there is only one matrix on each stack, and all matrices are set to the identity. The initial active texture unit selector is `TEXTURE0`, and the initial matrix mode is `MODELVIEW`.

2.12.3 Normal Transformation

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

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

and

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

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

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

$$(n_x' \ n_y' \ n_z' \ q') = (n_x \ n_y \ n_z \ q) \cdot M^{-1}$$

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

$$q = \begin{cases} 0, & w = 0, \\ -\frac{(n_x \ n_y \ n_z) \begin{pmatrix} x \\ y \\ z \end{pmatrix}}{w}, & w \neq 0 \end{cases} \quad (2.1)$$

Implementations may choose instead to transform $(n_x \ n_y \ n_z)$ to eye coordinates using

$$(n_x' \ n_y' \ n_z') = (n_x \ n_y \ n_z) \cdot M_u^{-1}$$

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

Rescale multiplies the transformed normals by a scale factor

$$(n_x'' \ n_y'' \ n_z'') = f (n_x' \ n_y' \ n_z')$$

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

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

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

Alternatively, an implementation may choose f as

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

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

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

$$n_f = m (n_x'' \ n_y'' \ n_z'')$$

If normalization is disabled, then $m = 1$. Otherwise

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

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

2.12.4 Generating Texture Coordinates

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

```
void TexGen{ifd}( enum coord, enum pname, T param );
void TexGen{ifd}v( enum coord, enum pname, T params );
```

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

If TEXTURE_GEN_MODE indicates OBJECT_LINEAR, then the generation function for the coordinate indicated by *coord* is

$$g = p_1x_o + p_2y_o + p_3z_o + p_4w_o.$$

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

If TEXTURE_GEN_MODE indicates EYE_LINEAR, then the function is

$$g = p'_1x_e + p'_2y_e + p'_3z_e + p'_4w_e$$

where

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

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

When used with a suitably constructed texture image, calling **TexGen** with TEXTURE_GEN_MODE indicating SPHERE_MAP can simulate the reflected image of

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

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

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

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

If `TEXTURE_GEN_MODE` indicates `NORMAL_MAP`, compute the normal vector n_f as described in section 2.12.3. Then the value assigned to an s coordinate is $s = n_{f_x}$; the value assigned to a t coordinate is $t = n_{f_y}$; and the value assigned to an r coordinate is $r = n_{f_z}$ (the values n_{f_x} , n_{f_y} , and n_{f_z} are the components of n_f .) Calling **TexGen** with a *coord* of `Q` when *pname* indicates `NORMAL_MAP` generates the error `INVALID_ENUM`.

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

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

2.13 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) returns information on the number of vertices and primitives processed by the GL and written to one or more buffer objects. Occlusion queries (see section 4.1.7) count the number of fragments or samples that pass the depth test.

The results of asynchronous queries are not returned by the GL immediately after the completion of the last command in the set; subsequent commands can be processed while the query results are not complete. When available, the query results are stored in an associated query object. The commands described in section 6.1.12 provide mechanisms to determine when query results are available and return the actual results of the query. The name space for query objects is the unsigned integers, with zero reserved by the GL.

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

A query object is created by calling

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

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

BeginQuery sets the active query object name for the query type given by *target* to *id*. If **BeginQuery** is called with an *id* of zero, if the active query object name for *target* is non-zero, if *id* is the active query object name for any query type, or if *id* is the active query object for conditional rendering (see section 2.14), the error `INVALID_OPERATION` is generated.

The command

```
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, **Begin**, **End**, all vertex array commands performing an implicit **Begin** and **End**, **DrawPixels** (see section 3.7.4), **Bitmap** (see section 3.8), **Clear** (see section 4.2.3), **Accum** (see section 4.2.4), **CopyPixels** (see section 4.3.3), and **EvalMesh1** and

EvalMesh2 (see section 5.1) have no effect. The effect of commands setting current vertex state, such as **Color** or **VertexAttrib**, are undefined. If the result of the occlusion query is non-zero, such commands are not discarded.

mode specifies how **BeginConditionalRender** interprets the results of the occlusion query given by *id*. If *mode* is `QUERY_WAIT`, the GL waits for the results of the query to be available and then uses the results to determine if subsequent rendering commands are discarded. If *mode* is `QUERY_NO_WAIT`, the GL may choose to unconditionally execute the subsequent rendering commands without waiting for the query to complete.

If *mode* is `QUERY_BY_REGION_WAIT`, the GL will also wait for occlusion query results and discard rendering commands if the result of the occlusion query is zero. If the query result is non-zero, subsequent rendering commands are executed, but the GL may discard the results of the commands for any region of the framebuffer that did not contribute to the sample count in the specified occlusion query. Any such discarding is done in an implementation-dependent manner, but the rendering command results may not be discarded for any samples that contributed to the occlusion query sample count. If *mode* is `QUERY_BY_REGION_NO_WAIT`, the GL operates as in `QUERY_BY_REGION_WAIT`, but may choose to unconditionally execute the subsequent rendering commands without waiting for the query to complete.

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

2.15 Transform Feedback

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

Transform feedback is started and finished by calling

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

and

Transform Feedback <i>primitiveMode</i>	Allowed render primitive (Begin) <i>modes</i>
POINTS	POINTS
LINES	LINES, LINE_LOOP, LINE_STRIP
TRIANGLES	TRIANGLES, TRIANGLE_STRIP, TRIANGLE_FAN QUADS, QUAD_STRIP, POLYGON

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

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

respectively. Transform feedback is said to be active after a call to **BeginTransformFeedback** and inactive after a call to **EndTransformFeedback**. *primitiveMode* is one of TRIANGLES, LINES, or POINTS, and specifies the output type of primitives that will be recorded into the buffer objects bound for transform feedback (see below). *primitiveMode* restricts the primitive types that may be rendered while transform feedback is active, as shown in table 2.9.

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

Transform feedback mode captures the values of varying variables written by an active vertex shader. The error INVALID_OPERATION is generated by **BeginTransformFeedback** if no vertex shader is active.

When transform feedback is active, all geometric primitives generated must be compatible with the value of *primitiveMode* passed to **BeginTransformFeedback**. The error INVALID_OPERATION is generated by **Begin** or any operation that implicitly calls **Begin** (such as **DrawElements**) if *mode* is not one of the allowed modes in table 2.9.

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

```
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 word-aligned. **BindBufferBase** is equivalent to calling **BindBufferRange** with *offset* zero and *size* equal to the size of *buffer*, rounded down so that it is word-aligned.

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

When quads and polygons are provided to transform feedback with a primitive mode of `TRIANGLES`, they will be tessellated and recorded as triangles (the order of tessellation within a primitive is undefined). Individual lines or triangles of a strip or fan primitive will be extracted and recorded separately. Incomplete primitives are not recorded.

Transform feedback can operate in either `INTERLEAVED_ATTRIBS` or `SEPARATE_ATTRIBS` mode. In `INTERLEAVED_ATTRIBS` mode, the values of one or more varyings are written, interleaved, into the buffer object bound to the first transform feedback binding point (*index* = 0). If more than one varying variable is written, they will be recorded in the order specified by **TransformFeedbackVaryings** (see section 2.20.3). In `SEPARATE_ATTRIBS` mode, the first varying variable specified by **TransformFeedbackVaryings** is written to the first transform feedback binding point; subsequent varying variables are written to the subsequent transform feedback binding points. The

total number of variables that may be captured in separate mode is given by `MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`.

If recording the vertices of a primitive to the buffer objects being used for transform feedback purposes would result in either exceeding the limits of any buffer object's size, or in exceeding the end position $offset + size - 1$, as set by **BindBufferRange**, then no vertices of that primitive are recorded in any buffer object, and the counter corresponding to the asynchronous query target `TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN` (see section 2.16) 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 **DrawPixels** and **ReadPixels** to a pixel buffer object binding point and client access to a buffer mapped with **MapBuffer**.

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

2.16 Primitive Queries

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

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

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

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

2.17 Clipping

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

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

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

A client-defined clip plane is specified with

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

The value of the first argument, p , is a symbolic constant, `CLIP_PLANE i` , where i is an integer between 0 and $n - 1$, indicating one of n client-defined clip planes. eqn

is an array of four double-precision floating-point values. These are the coefficients of a plane equation in object coordinates: p_1 , p_2 , p_3 , and p_4 (in that order). The inverse of the current model-view matrix is applied to these coefficients, at the time they are specified, yielding

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

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

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

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

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

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

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

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

Client-defined clip planes are enabled with the generic **Enable** command and disabled with the **Disable** command. The value of the argument to either command is `CLIP_PLANEi` 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_PLANEi = CLIP_PLANE0 + 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 in color, secondary color, texture coordinate, and fog coordinate clipping (section 2.19.8).

If the primitive is a polygon, then it is passed if every one of its edges lies entirely inside the clip volume and either clipped or discarded otherwise. Polygon clipping may cause polygon edges to be clipped, but because polygon connectivity must be maintained, these clipped edges are connected by new edges that lie along the clip volume's boundary. Thus, clipping may require the introduction of new vertices into a polygon. Edge flags are associated with these vertices so that edges introduced by clipping are flagged as boundary (edge flag `TRUE`), and so that original edges of the polygon that become cut off at these vertices retain their original flags.

If it happens that a polygon intersects an edge of the clip volume's boundary, then the clipped polygon must include a point on this boundary edge. This point must lie in the intersection of the boundary edge and the convex hull of the vertices of the original polygon. We impose this requirement because the polygon may not be exactly planar.

Primitives rendered with clip planes must satisfy a complementarity criterion. Suppose a single clip plane with coefficients $(p'_1 \ p'_2 \ p'_3 \ p'_4)$ (or a number of similarly specified clip planes) is enabled and a series of primitives are drawn. Next, suppose that the original clip plane is respecified with coefficients $(-p'_1 \ -p'_2 \ -p'_3 \ -p'_4)$ (and correspondingly for any other clip planes) and the primitives are drawn again (and the GL is otherwise in the same state). In this case, primitives must not be missing any pixels, nor may any pixels be drawn twice in regions where those primitives are cut by the clip planes.

The state required for clipping is at least 6 sets of plane equations (each consisting of four double-precision floating-point coefficients) and at least 6 corresponding bits indicating which of these client-defined plane equations are enabled. In the

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

2.18 Current Raster Position

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

The current raster position is set using one of the commands

```
void RasterPos{234}{sifd}( T coords );
void RasterPos{234}{sifd}v( T coords );
```

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

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

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

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

The transformed coordinates are passed to clipping as if they represented a point. If the "point" is not culled, then the projection to window coordinates is

computed (section 2.12) and saved as the current raster position, and the valid bit is set. If the “point” is culled, the current raster position and its associated data become indeterminate and the valid bit is cleared. Figure 2.7 summarizes the behavior of the current raster position.

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

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

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

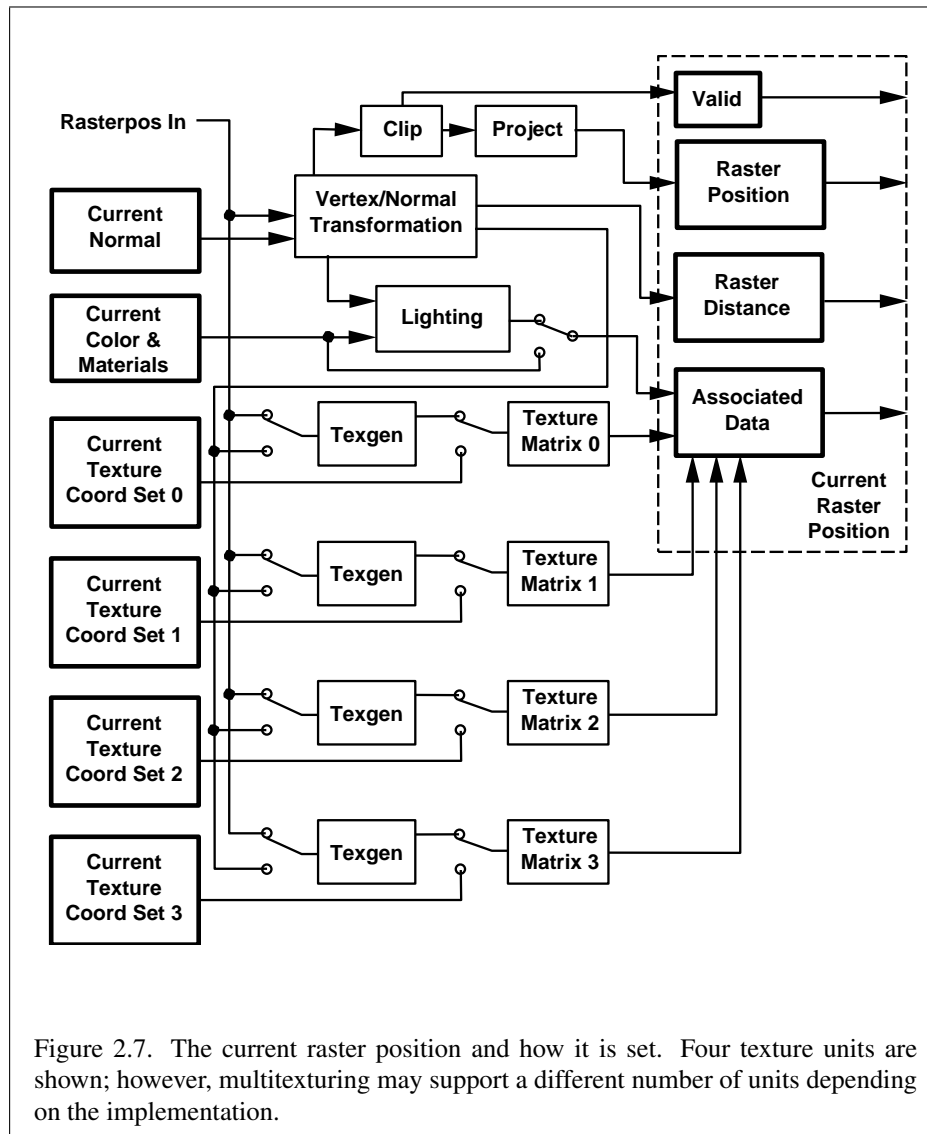
$$\begin{aligned} x_w &= x \\ y_w &= y \\ z_w &= \begin{cases} n, & z \leq 0 \\ f, & z \geq 1 \\ n + z(f - n), & \text{otherwise} \end{cases} \\ w_c &= 1 \end{aligned}$$

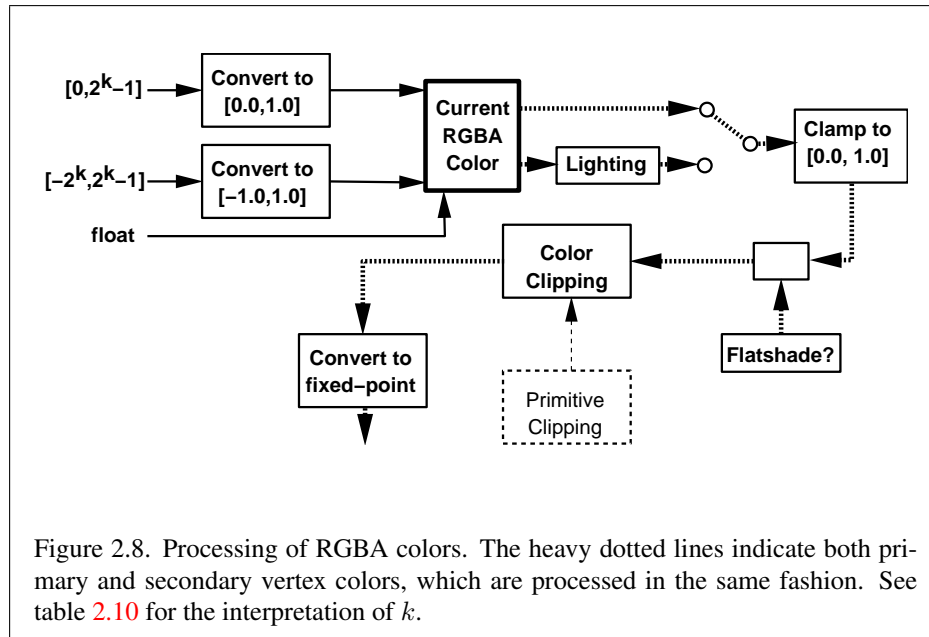
where n and f are the values passed to **DepthRange** (see section 2.12.1).

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

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

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



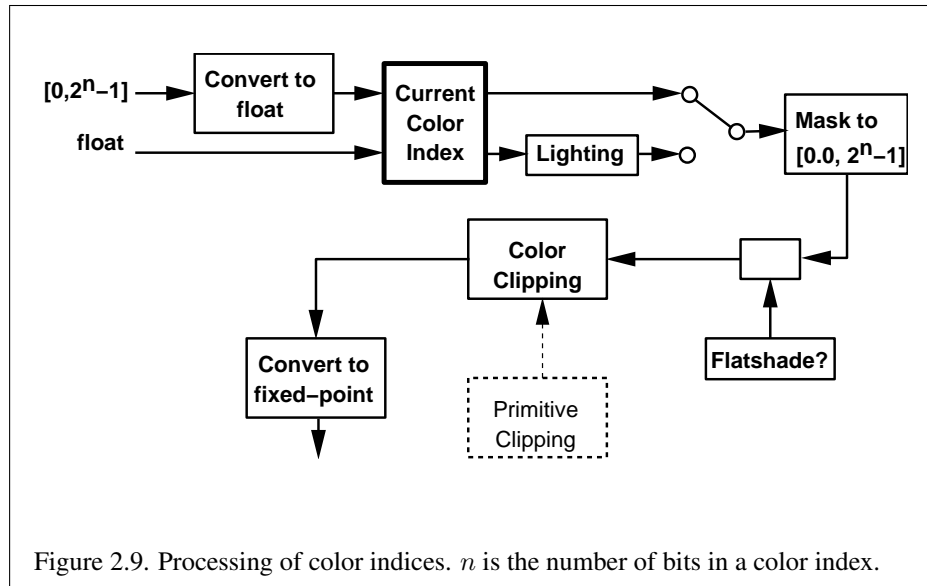


coordinate distance is 0, the fog coordinate is 0, the valid bit is set, the associated RGBA color is (1, 1, 1, 1), the associated RGBA secondary color is (0, 0, 0, 1), and the associated color index color is 1. In RGBA mode, the associated color index always has its initial value; in color index mode, the RGBA color and secondary color always maintain their initial values.

2.19 Colors and Coloring

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

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



GL Type of c	Conversion to floating-point
ubyte	$\frac{c}{2^8-1}$
byte	$\frac{2c+1}{2^8-1}$
ushort	$\frac{c}{2^{16}-1}$
short	$\frac{2c+1}{2^{16}-1}$
uint	$\frac{c}{2^{32}-1}$
int	$\frac{2c+1}{2^{32}-1}$
half	c
float	c
double	c

Table 2.10: Component conversions. Color, normal, and depth component values (c) of different types are converted to an internal floating-point representation using the equations in this table. All arithmetic is done in the internal floating point format. These conversions apply to components specified as parameters to GL commands and to components in pixel data. The equations remain the same even if the implemented ranges of the GL data types are greater than the minimum required ranges. (Refer to table 2.2)

tion is masked (see section 2.19.6). After clamping or masking, a primitive may be *flatshaded*, indicating that all vertices of the primitive are to have the same colors. Finally, if a primitive is clipped, then colors (and texture coordinates) must be computed at the vertices introduced or modified by clipping.

2.19.1 Lighting

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

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

Lighting Operation

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

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

Before specifying the way that lighting computes colors, we introduce operators and notation that simplify the expressions involved. If \mathbf{c}_1 and \mathbf{c}_2 are colors without alpha where $\mathbf{c}_1 = (r_1, g_1, b_1)$ and $\mathbf{c}_2 = (r_2, g_2, b_2)$, then define $\mathbf{c}_1 * \mathbf{c}_2 = (r_1 r_2, g_1 g_2, b_1 b_2)$. Addition of colors is accomplished by addition of

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

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

the components. Multiplication of colors by a scalar means multiplying each component by that scalar. If \mathbf{d}_1 and \mathbf{d}_2 are directions, then define

$$\mathbf{d}_1 \odot \mathbf{d}_2 = \max\{\mathbf{d}_1 \cdot \mathbf{d}_2, 0\}.$$

(Directions are taken to have three coordinates.) If \mathbf{P}_1 and \mathbf{P}_2 are (homogeneous, with four coordinates) points then let $\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$ be the unit vector that points from \mathbf{P}_1 to \mathbf{P}_2 . Note that if \mathbf{P}_2 has a zero w coordinate and \mathbf{P}_1 has non-zero w coordinate, then $\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$ is the unit vector corresponding to the direction specified by the x , y , and z coordinates of \mathbf{P}_2 ; if \mathbf{P}_1 has a zero w coordinate and \mathbf{P}_2 has a non-zero w coordinate then $\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$ is the unit vector that is the negative of that corresponding to the direction specified by \mathbf{P}_1 . If both \mathbf{P}_1 and \mathbf{P}_2 have zero w coordinates, then $\overrightarrow{\mathbf{P}_1\mathbf{P}_2}$ is the unit vector obtained by normalizing the direction corresponding to $\mathbf{P}_2 - \mathbf{P}_1$.

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

Lighting produces two colors at a vertex: a primary color \mathbf{c}_{pri} and a secondary color \mathbf{c}_{sec} . The values of \mathbf{c}_{pri} and \mathbf{c}_{sec} depend on the light model color control, c_{es} . If $c_{es} = \text{SINGLE_COLOR}$, then the equations to compute \mathbf{c}_{pri} and \mathbf{c}_{sec} are

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

If $c_{es} = \text{SEPARATE_SPECULAR_COLOR}$, then

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

where

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

$$\mathbf{h}_i = \begin{cases} \overrightarrow{\mathbf{VP}}_{pli} + \overrightarrow{\mathbf{VP}}_e, & v_{bs} = \text{TRUE}, \\ \overrightarrow{\mathbf{VP}}_{pli} + \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^T, & v_{bs} = \text{FALSE}, \end{cases} \quad (2.3)$$

$$att_i = \begin{cases} \frac{1}{k_{0i} + k_{1i}\|\overrightarrow{\mathbf{VP}}_{pli}\| + k_{2i}\|\overrightarrow{\mathbf{VP}}_{pli}\|^2}, & \text{if } \mathbf{P}_{pli}\text{'s } w \neq 0, \\ 1.0, & \text{otherwise.} \end{cases} \quad (2.4)$$

$$spot_i = \begin{cases} (\overrightarrow{\mathbf{P}}_{pli} \mathbf{V} \odot \hat{\mathbf{s}}_{dli})^{s_{rli}}, & c_{rli} \neq 180.0, \overrightarrow{\mathbf{P}}_{pli} \mathbf{V} \odot \hat{\mathbf{s}}_{dli} \geq \cos(c_{rli}), \\ 0.0, & c_{rli} \neq 180.0, \overrightarrow{\mathbf{P}}_{pli} \mathbf{V} \odot \hat{\mathbf{s}}_{dli} < \cos(c_{rli}), \\ 1.0, & c_{rli} = 180.0. \end{cases} \quad (2.5)$$

All computations are carried out in eye coordinates.

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

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

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

Additionally, vertex shaders can operate in two-sided color mode. When a vertex shader is active, front and back colors can be computed by the vertex shader and written to the `gl_FrontColor`, `gl_BackColor`, `gl_FrontSecondaryColor` and `gl_BackSecondaryColor` outputs. If `VERTEX_PROGRAM_TWO_SIDE` is enabled, the GL chooses between front and back colors, as described below. Otherwise, the front color output is always selected. Two-sided color mode is

enabled and disabled by calling **Enable** or **Disable** with the symbolic value `VERTEX_PROGRAM_TWO_SIDE`.

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

$$a = \frac{1}{2} \sum_{i=0}^{n-1} x_w^i y_w^{i \oplus 1} - x_w^{i \oplus 1} y_w^i \quad (2.6)$$

where x_w^i and y_w^i are the x and y window coordinates of the i th vertex of the n -vertex polygon (vertices are numbered starting at zero for purposes of this computation) and $i \oplus 1$ is $(i + 1) \bmod n$. 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) indicates that if $a \leq 0$, then the color of each vertex of the polygon becomes the back color computed for that vertex while if $a > 0$, then the front color is selected. If *dir* is `CW`, then a is replaced by $-a$ in the above inequalities. This requires one bit of state; initially, it indicates `CCW`.

2.19.2 Lighting Parameter Specification

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

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

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

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

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

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

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

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

$$\begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix}$$

is transformed to

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

An individual light is enabled or disabled by calling **Enable** or **Disable** with the symbolic value `LIGHTi` (*i* is in the range 0 to $n - 1$, where n is the implementation-dependent number of lights). If light *i* is disabled, the *i*th term in the lighting equation is effectively removed from the summation.

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

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

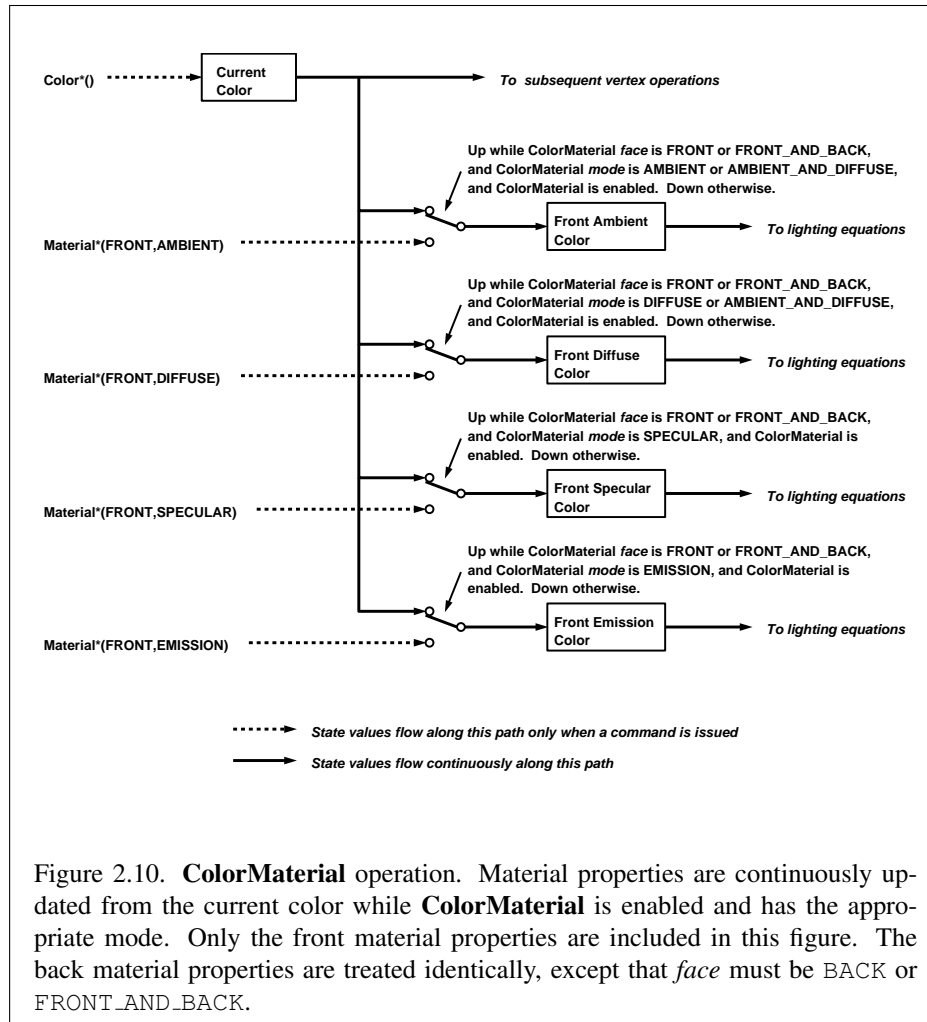


Figure 2.10. **ColorMaterial** operation. Material properties are continuously updated from the current color while **ColorMaterial** is enabled and has the appropriate mode. Only the front material properties are included in this figure. The back material properties are treated identically, except that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

2.19.3 ColorMaterial

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

The command that controls which of these modes is selected is

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

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

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

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

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

2.19.4 Lighting State

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

2.19.5 Color Index Lighting

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

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

and

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

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

Next, let

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

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

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

Then color index lighting produces a value c , given by

$$c = a_m + d(1 - s')(d_m - a_m) + s'(s_m - a_m).$$

The final color index is

$$c' = \min\{c, s_m\}.$$

The values a_m , d_m and s_m are material properties described in tables 2.11 and 2.12. Any ambient light intensities are incorporated into a_m . As with RGBA lighting, disabled lights cause the corresponding terms from the summations to be omitted. The interpretation of t_{bs} and the calculation of front and back colors is carried out as has already been described for RGBA lighting.

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

2.19.6 Clamping or Masking

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

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

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

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

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

2.19.7 Flatshading

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

Flatshading is controlled by

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

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

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

Primitive type of polygon i	Vertex
single polygon ($i \equiv 1$)	1
triangle strip	$i + 2$
triangle fan	$i + 2$
independent triangle	$3i$
quad strip	$2i + 2$
independent quad	$4i$

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

2.19.8 Color and Associated Data Clipping

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

Let the colors assigned to the two vertices \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 color associated with \mathbf{P} as

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

(For a color index color, multiplying a color by a scalar means multiplying the index by the scalar. For an RGBA color, it means multiplying each of R, G, B, and A by the scalar. Both primary and secondary colors are treated in the same fashion.) Polygon clipping may create a clipped vertex along an edge of the clip volume's boundary. This situation is handled by noting that polygon clipping proceeds by clipping against one plane of the clip volume's boundary at a time. Color clipping is done in the same way, so that clipped points always occur at the intersection of polygon edges (possibly already clipped) with the clip volume's boundary.

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

For vertex shader varying variables specified to be interpolated without per-

spective correction (using the `noperspective` qualifier), the value of t used to obtain the varying value associated with **P** will be adjusted to produce results that vary linearly in screen space.

2.19.9 Final Color Processing

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

In RGBA mode with vertex color clamping enabled, each color component (already clamped to $[0, 1]$) may be converted (by rounding to nearest) to a fixed-point value with m bits. We assume that the fixed-point representation used represents each value $k/(2^m - 1)$, where $k \in \{0, 1, \dots, 2^m - 1\}$, as k (e.g. 1.0 is represented in binary as a string of all ones). m must be at least as large as the number of bits in the corresponding component of the framebuffer. m must be at least 2 for A if the framebuffer does not contain an A component, or if there is only 1 bit of A in the framebuffer. GL implementations are not required to convert clamped color components to fixed-point.

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

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

2.20 Vertex Shaders

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

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

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

In addition to vertex shaders, *fragment shaders* can be created, compiled, and linked into program objects. Fragment shaders affect the processing of fragments during rasterization, and are described in section 3.12. A single program object can contain both vertex and fragment shaders.

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

2.20.1 Shader Objects

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

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

To create a shader object, use the command

```
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 `chars` in each string (the string length). If an element in *length* is negative, its accompanying string is null-terminated. If *length* is `NULL`, all strings in the *string* argument are considered null-terminated. The **ShaderSource** command sets the source code for the *shader* to the text strings in the *string* array. If *shader* previously had source code loaded into it, the existing source code is completely replaced. Any length passed in excludes the null terminator in its count.

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

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

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

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

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

Shader objects can be deleted with the command

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

If *shader* is not attached to any program object, it is deleted immediately. Otherwise, *shader* is flagged for deletion and will be deleted when it is no longer attached to any program object. If an object is flagged for deletion, its boolean status bit `DELETE_STATUS` is set to `true`. The value of `DELETE_STATUS` can be queried with **GetShaderiv** (see section 6.1.15). **DeleteShader** will silently ignore the value zero.

2.20.2 Program Objects

The shader objects that are to be used by the programmable stages of the GL are collected together to form a *program object*. The programs that are executed by

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 6.1.15). This status will be set to `TRUE` if a valid executable is created, and `FALSE` otherwise. Linking can fail for a variety of reasons as specified in the OpenGL Shading Language Specification. Linking will also fail if one or more of the shader objects, attached to *program* are not compiled successfully, or if more active uniform or active sampler variables are used in *program* than allowed (see section 2.20.3). If **LinkProgram** failed, any information about a previous link of that program object is lost. Thus, a failed link does not restore the old state of *program*.

Each program object has an information log that is overwritten as a result of a link operation. This information log can be queried with **GetProgramInfoLog** to obtain more information about the link operation or the validation information (see section 6.1.15).

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

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

This command will install the executable code as part of current rendering state if the program object *program* contains valid executable code, i.e. has been linked successfully. If **UseProgram** is called with *program* set to 0, it is as if the GL had no programmable stages and the fixed-function paths will be used instead. If *program* has not been successfully linked, the error `INVALID_OPERATION` is generated and the current rendering state is not modified.

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

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

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

Program objects can be deleted with the command

```
void DeleteProgram( uint program );
```

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

2.20.3 Shader Variables

A vertex shader can reference a number of variables as it executes. *Vertex attributes* are the per-vertex values specified in section 2.7. *Uniforms* are per-program vari-

ables that are constant during program execution. *Samplers* are a special form of uniform used for texturing (section 3.9). *Varying variables* hold the results of vertex shader execution that are used later in the pipeline. The following sections describe each of these variable types.

Vertex Attributes

Vertex shaders can access built-in vertex attribute variables corresponding to the per-vertex state set by commands such as **Vertex**, **Normal**, **Color**. Vertex shaders can also define named attribute variables, which are bound to the generic vertex attributes that are set by **VertexAttrib***. This binding can be specified by the application before the program is linked, or automatically assigned by the GL when the program is linked.

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

An attribute variable (either conventional or generic) is considered *active* if it is determined by the compiler and linker that the attribute may be accessed when the shader is executed. Attribute variables that are declared in a vertex shader but never used will not count against the limit. In cases where the compiler and linker cannot make a conclusive determination, an attribute will be considered active. A program object will fail to link if the sum of the active generic and active conventional attributes exceeds `MAX_VERTEX_ATTRIBS`.

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

```
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 6.1.15). If *index* is greater than or equal to `ACTIVE_ATTRIBUTES`, the error `INVALID_VALUE` is generated. Note that *index* simply identifies a member in a list of active attributes, and has no relation to the generic attribute that the corresponding variable is bound to.

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

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

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

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

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

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

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

returns the generic attribute index that the attribute variable named *name* was bound

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

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

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

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

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

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

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

The values of generic attributes sent to generic attribute index *i* are part of current state, just like the conventional attributes. If a new program object has

been made active, then these values will be tracked by the GL in such a way that the same values will be observed by attributes in the new program object that are also bound to index i .

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

Uniform Variables

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

The amount of storage available for uniform variables accessed by a vertex shader is specified by the implementation dependent constant `MAX_VERTEX_UNIFORM_COMPONENTS`. This value represents the number of individual floating-point, integer, or boolean values that can be held in uniform variable storage for a vertex shader. A uniform matrix will consume no more than $4 \times \min(r, c)$ such values, where r and c are the number of rows and columns in the matrix. A link error will be generated if an attempt is made to utilize more than the space available for vertex shader uniform variables.

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

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

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

This command will return the location of uniform variable *name*. *name* must be a null terminated string, without white space. The value -1 will be returned if *name* does not correspond to an active uniform variable name in *program* or if *name* starts with the reserved prefix "gl_". If *program* has not been successfully linked, the error `INVALID_OPERATION` is generated. After a program is linked, the location of a uniform variable will not change, unless the program is re-linked.

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

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

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

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

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

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

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

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

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`, which can be queried with **GetProgramiv** (see section 6.1.15).

Each uniform variable, declared in a shader, is broken down into one or more strings using the `"."` (dot) and `"["` operators, if necessary, to the point that it is legal to pass each string back into **GetUniformLocation**. Each of these strings constitutes one active uniform, and each string is assigned an index.

For the selected uniform, the type of the uniform is returned into *type*. The size of the uniform 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`, `INT`, `INT_VEC2`, `INT_VEC3`, `INT_VEC4`, `BOOL`, `BOOL_VEC2`, `BOOL_VEC3`, `BOOL_VEC4`, `FLOAT_MAT2`, `FLOAT_MAT3`, `FLOAT_MAT4`, `FLOAT_MAT2x3`, `FLOAT_MAT2x4`, `FLOAT_MAT3x2`, `FLOAT_MAT3x4`, `FLOAT_MAT4x2`, `FLOAT_MAT4x3`, `SAMPLER_1D`, `SAMPLER_2D`, `SAMPLER_3D`, `SAMPLER_CUBE`, `SAMPLER_1D_SHADOW`, `SAMPLER_2D_SHADOW`, `SAMPLER_1D_ARRAY`, `SAMPLER_2D_ARRAY`, `SAMPLER_1D_ARRAY_SHADOW`, `SAMPLER_2D_ARRAY_SHADOW`, `SAMPLER_CUBE_SHADOW`, `INT_SAMPLER_1D`, `INT_SAMPLER_2D`, `INT_SAMPLER_3D`, `INT_SAMPLER_CUBE`, `INT_SAMPLER_1D_ARRAY`, `INT_SAMPLER_2D_ARRAY`, `UNSIGNED_INT`, `UNSIGNED_INT_VEC2`, `UNSIGNED_INT_VEC3`, `UNSIGNED_INT_VEC4`, `UNSIGNED_INT_SAMPLER_1D`, `UNSIGNED_INT_SAMPLER_2D`, `UNSIGNED_INT_SAMPLER_3D`, `UNSIGNED_INT_SAMPLER_CUBE`, `UNSIGNED_INT_SAMPLER_1D_ARRAY`, or `UNSIGNED_INT_SAMPLER_2D_ARRAY`.

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.

To load values into the uniform variables 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 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 **Uniformli{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.

Samplers

Samplers are special uniforms used in the OpenGL Shading Language to identify the texture object used for each texture lookup. The value of a sampler indicates the texture image unit being accessed. Setting a sampler's value to i selects texture

image unit number i . The values of i range from zero to the implementation-dependent maximum supported number of texture image units.

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

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

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

Active samplers are samplers actually being used in a program object. The **LinkProgram** command determines if a sampler is active or not. The **LinkProgram** command will attempt to determine if the active samplers in the shader(s) contained in the program object exceed the maximum allowable limits. If it determines that the count of active samplers exceeds the allowable limits, then the link fails (these limits can be different for different types of shaders). Each active sampler variable counts against the limit, even if multiple samplers refer to the same texture image unit. If this cannot be determined at link time, for example if the program object only contains a vertex shader, then it will be determined at the next rendering command issued, and an `INVALID_OPERATION` error will then be generated.

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 floating-point 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. The varying variables specified in *varyings* can be either built-in varying variables (beginning with "gl-") or user-defined ones. varying variables are written out in the order they appear in the array *varyings*. *bufferMode* is either `INTERLEAVED_ATTRIBS` or `SEPARATE_ATTRIBS`, and identifies the mode used to capture the varying variables when transform feedback is active. The error `INVALID_VALUE` is generated if *program* is not the name of a program object, or if *bufferMode* is `SEPARATE_ATTRIBS` and *count* is greater than the value of the implementation-dependent limit `MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS`.

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

- the *count* specified by **TransformFeedbackVaryings** is non-zero, but the program object has no vertex shader;
- any variable name specified in the *varyings* array is not declared as an output in the vertex shader.
- any two entries in the *varyings* array specify the same varying variable;
- the total number of components to capture in any varying variable in *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 6.1.15). 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 a new set of varying variables is specified by **TransformFeedbackVaryings** after a program object has been linked, the information returned by **GetTransformFeedbackVarying** will not reflect those variables until the program is re-linked.

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

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 FLOAT, FLOAT_VEC2, FLOAT_VEC3, FLOAT_VEC4, INT, INT_VEC2, INT_VEC3, INT_VEC4, UNSIGNED_INT, UNSIGNED_INT_VEC2, UNSIGNED_INT_VEC3, UNSIGNED_INT_VEC4, FLOAT_MAT2, FLOAT_MAT3, or FLOAT_MAT4. If an error occurred, the return parameters *length*, *size*, *type* and

name will be unmodified. This command will return as much information about the varying variables as possible. If no information is available, *length* will be set to zero and *name* will be an empty string. This situation could arise if **GetTransformFeedbackVarying** is called after a failed link.

2.20.4 Shader Execution

If a successfully linked program object that contains a vertex shader is made current by calling **UseProgram**, the executable version of the vertex shader is used to process incoming vertex values rather than the fixed-function vertex processing described in sections 2.12 through 2.19. In particular,

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

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

- Color clamping or masking (section 2.19.6).
- Perspective division on clip coordinates (section 2.12).
- Viewport mapping, including depth range scaling (section 2.12.1).
- Clipping, including client-defined clip planes (section 2.17).

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

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

Shader Only Texturing

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

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

Texel Fetches

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

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

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

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

- the computed LOD is less than the texture's base level ($level_{base}$) or greater than the maximum level ($level_{max}$)
- the computed LOD is not the texture's base level and the texture's minification filter is 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 3.15

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

Texture Size Query

The OpenGL Shading Language texture size functions provide the ability to query the size of a texture image. The LOD value lod passed in as an argument to the texture size functions is added to the $level_{base}$ of the texture to determine a texture image level. The dimensions of that image level, excluding a possible border, are then returned. If the computed texture image level is outside the range $[level_{base}, level_{max}]$, the results are undefined. When querying the size of an array texture, both the dimensions and the layer index are returned.

Texture Access

Vertex shaders have the ability to do a lookup into a texture map, if supported by the GL implementation. The maximum number of texture image units available to a vertex shader is `MAX_VERTEX_TEXTURE_IMAGE_UNITS`; a maximum number of zero indicates that the GL implementation does not support texture accesses in vertex shaders. The maximum number of texture image units available to the fragment stage of the GL is `MAX_TEXTURE_IMAGE_UNITS`. Both the vertex shader and fragment processing combined cannot use more than `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 3.9.7 and 3.9.8, and converted it to a texture source color C_s according to table 3.23 (section 3.9.13). A four-component vector (R_s, G_s, B_s, A_s) is returned to the vertex shader.

In a vertex shader, it is not possible to perform automatic level-of-detail calculations using partial derivatives of the texture coordinates with respect to window coordinates as described in section 3.9.7. Hence, there is no automatic selection of an image array level. Minification or magnification of a texture map is controlled by a level-of-detail value optionally passed as an argument in the texture lookup functions. If the texture lookup function supplies an explicit level-of-detail value l , then the pre-bias level-of-detail value $\lambda_{base}(x, y) = l$ (replacing equation 3.16). 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 3.20) are ignored.

Texture lookups involving textures with depth component data can either return the depth data directly or return the results of a comparison with a reference depth value specified in the coordinates passed to the texture lookup function, as described in section 3.9.14. The comparison operation is requested in the shader by using any of the shadow sampler types and in the texture using the `TEXTURE_COMPARE_MODE` parameter. These requests must be consistent; the results of a texture lookup are undefined if:

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

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

If a vertex shader uses a sampler where the associated texture object is not complete, as defined in section 3.9.10, the texture image unit will return $(R, G, B, A) = (0, 0, 0, 1)$.

Shader Inputs

Besides having access to vertex attributes and uniform variables, vertex shaders can access the read-only built-in variable `gl_VertexID`. `gl_VertexID` holds the integer index i explicitly passed to **ArrayElement** to specify the vertex, or implicitly

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

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

Also see section 7.1 of the OpenGL Shading Language Specification.

Shader Outputs

A vertex shader can write to built-in as well as user-defined varying variables. These values are expected to be interpolated across the primitive it outputs, unless they are specified to be flat shaded. Refer to section 2.19.7 and the OpenGL Shading Language specification sections 4.3.6, 7.1 and 7.6 for more detail.

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

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

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

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

The built-in special variable `gl_ClipVertex` holds the vertex coordinate used in the clipping stage, as described in section 2.17.

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

Position Invariance

If a vertex shader uses the built-in function `frtransform` to generate a vertex position, then this generally guarantees that the transformed position will be the same whether using this vertex shader or the fixed-function pipeline. This allows for correct multi-pass rendering algorithms, where some passes use fixed-function vertex

transformation and other passes use a vertex shader. If a vertex shader does not use `glTransform` to generate a position, transformed positions are not guaranteed to match, even if the sequence of instructions used to compute the position match the sequence of transformations described in section 2.12.

Validation

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

This error is generated by **Begin**, **RasterPos**, or any command that performs an implicit **Begin** if:

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

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

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

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

to validate the program object *program* against the current GL state. Each program object has a boolean status, `VALIDATE_STATUS`, that is modified as a result of validation. This status can be queried with **GetProgramiv** (see section 6.1.15). 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.

2.20.5 Required State

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

The state required per shader object consists of:

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

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

The state required per program object consists of:

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

Additional state required to support vertex shaders consists of:

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

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

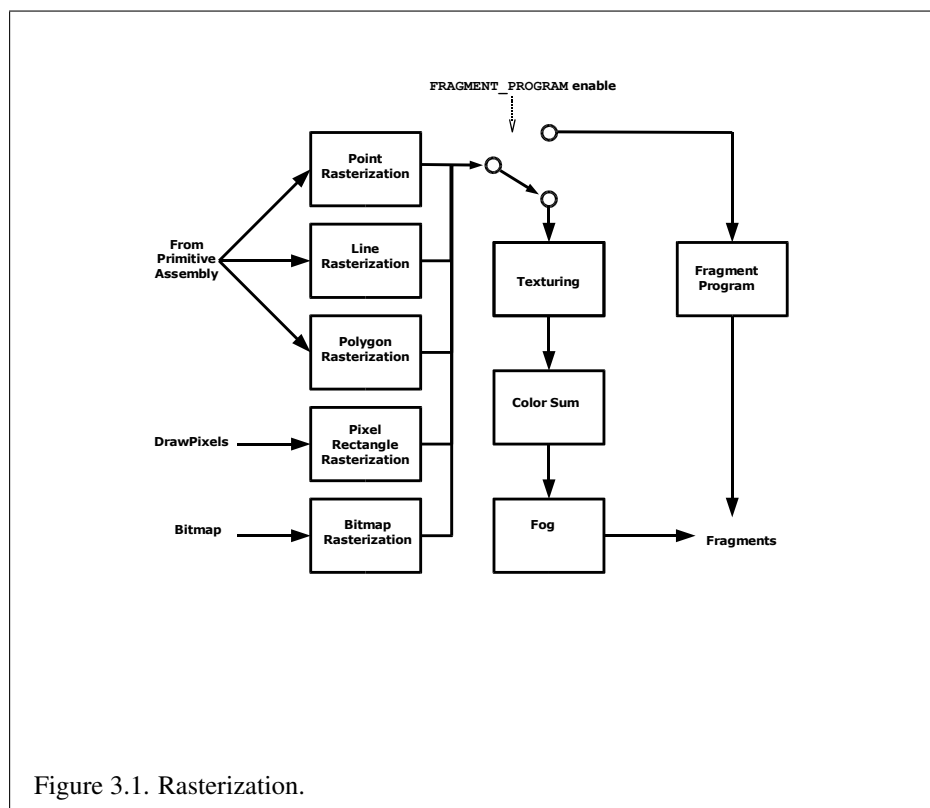
Chapter 3

Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional image. Each point of this image contains such information as color and depth. Thus, rasterizing a primitive consists of two parts. The first is to determine which squares of an integer grid in window coordinates are occupied by the primitive. The second is assigning a depth value and one or more color values to each such square. The results of this process are passed on to the next stage of the GL (per-fragment operations), which uses the information to update the appropriate locations in the framebuffer. Figure 3.1 diagrams the rasterization process. The color values assigned to a fragment are initially determined by the rasterization operations (sections 3.4 through 3.8) and modified by either the execution of the texturing, color sum, and fog operations defined in sections 3.9, 3.10, and 3.11, or by a fragment shader as defined in section 3.12. The final depth value is initially determined by the rasterization operations and may be modified or replaced by a fragment shader. The results from rasterizing a point, line, polygon, pixel rectangle or bitmap can be routed through a fragment shader.

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

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



Several factors affect rasterization. Primitives may be discarded before rasterization. Lines and polygons may be stippled. Points may be given differing diameters and line segments differing widths. A point, line segment, or polygon may be antialiased.

3.1 Discarding Primitives Before Rasterization

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

3.2 Invariance

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

3.3 Antialiasing

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

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

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

The details of how antialiased fragment coverage values are computed are difficult to specify in general. The reason is that high-quality antialiasing may take

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 5.6), allowing a user to make an image quality versus speed tradeoff.

3.3.1 Multisampling

Multisampling is a mechanism to antialias all GL primitives: points, lines, polygons, bitmaps, and images. The technique is to sample all primitives multiple times

at each pixel. The color sample values are resolved to a single, displayable color each time a pixel is updated, so the antialiasing appears to be automatic at the application level. Because each sample includes color, depth, and stencil information, the color (including texture operation), depth, and stencil functions perform equivalently to the single-sample mode.

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

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

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

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

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

Multisample rasterization is enabled or disabled by calling **Enable** or **Disable**

with the symbolic constant `MULTISAMPLE`.

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

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

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

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

3.4 Points

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

```
void PointSize( float size );
```

size specifies the requested size of a point. The default value is 1.0. A value less than or equal to zero results in the error `INVALID_VALUE`.

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

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

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

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

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

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

$$width = \begin{cases} derived_size & derived_size \geq threshold \\ threshold & otherwise \end{cases} \quad (3.1)$$

and the fade factor is computed as follows:

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

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

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

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

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

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

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

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

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

3.4.1 Basic Point Rasterization

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

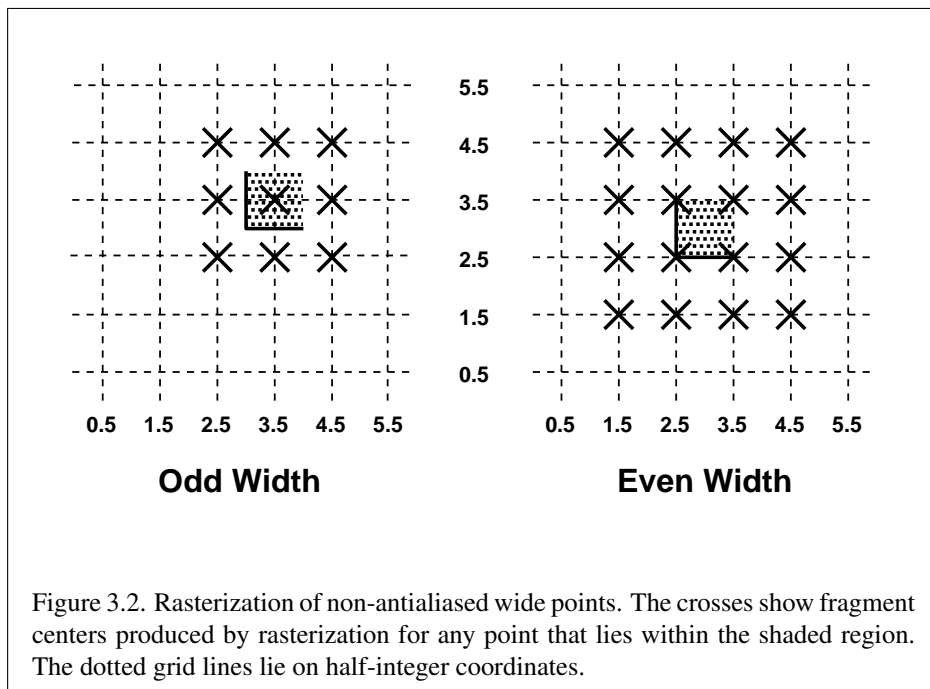
The effect of a point width other than 1.0 depends on the state of point antialiasing and point sprites. If antialiasing and point sprites are disabled, the actual width is determined by rounding the supplied width to the nearest integer, then clamping it to the implementation-dependent maximum non-antialiased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

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

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

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

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



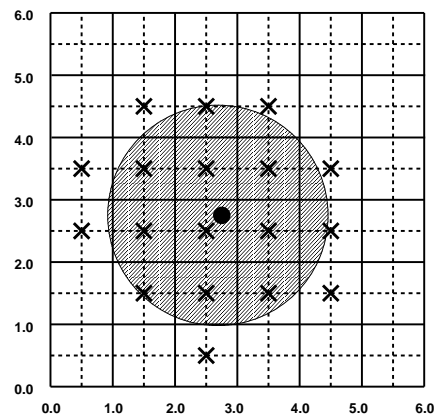


Figure 3.3. Rasterization of antialiased wide points. The black dot indicates the point to be rasterized. The shaded region has the specified width. The X marks indicate those fragment centers produced by rasterization. A fragment's computed coverage value is based on the portion of the shaded region that covers the corresponding fragment square. Solid lines lie on integer coordinates.

All fragments produced in rasterizing a non-antialiased point are assigned the same associated data, which are those of the vertex corresponding to the point.

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

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

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

All fragments produced in rasterizing a point sprite are assigned the same associated data, which are those of the vertex corresponding to the point. However, for each texture coordinate set where `COORD_REPLACE` is `TRUE`, these texture coordinates are replaced with point sprite texture coordinates. The s coordinate varies from 0 to 1 across the point horizontally left-to-right. If `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 coordinates:

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

$$t = \begin{cases} \frac{1}{2} + \frac{(y_f + \frac{1}{2} - y_w)}{size}, & \text{POINT_SPRITE_COORD_ORIGIN} = \text{LOWER_LEFT} \\ \frac{1}{2} - \frac{(y_f + \frac{1}{2} - y_w)}{size}, & \text{POINT_SPRITE_COORD_ORIGIN} = \text{UPPER_LEFT} \end{cases} \quad (3.4)$$

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

vertex for the point.

The widths supported for point sprites must be a superset of those supported for antialiased points. There is no requirement that these widths must be equally spaced. If an unsupported width is requested, the nearest supported width is used instead.

3.4.2 Point Rasterization State

The state required to control point rasterization consists of the floating-point point width, three floating-point values specifying the minimum and maximum point size and the point fade threshold size, three floating-point values specifying the distance attenuation coefficients, a bit indicating whether or not antialiasing is enabled, a bit for the point sprite texture coordinate replacement mode for each texture coordinate set, and a bit for the point sprite texture coordinate origin.

3.4.3 Point Multisample Rasterization

If `MULTISAMPLE` is enabled, and the value of `SAMPLE_BUFFERS` is one, then points are rasterized using the following algorithm, regardless of whether point antialiasing (`POINT_SMOOTH`) is enabled or disabled. Point rasterization produces a fragment for each framebuffer pixel with one or more sample points that intersect a region centered at the point's (x_w, y_w) . This region is a circle having diameter equal to the current point width if `POINT_SPRITE` is disabled, or a square with side equal to the current point width if `POINT_SPRITE` is enabled. Coverage bits that correspond to sample points that intersect the region are 1, other coverage bits are 0. All data associated with each sample for the fragment are the data associated with the point being rasterized, with the exception of texture coordinates when `POINT_SPRITE` is enabled; these texture coordinates are computed as described in section 3.4.

Point size range and number of gradations are equivalent to those supported for antialiased points when `POINT_SPRITE` is disabled. The set of point sizes supported is equivalent to those for point sprites without multisample when `POINT_SPRITE` is enabled.

3.5 Line Segments

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

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

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

3.5.1 Basic Line Segment Rasterization

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

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

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

Essentially, a line segment starting at \mathbf{p}_a and ending at \mathbf{p}_b produces those fragments f for which the segment intersects R_f , except if \mathbf{p}_b is contained in R_f . See figure 3.4.

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

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

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

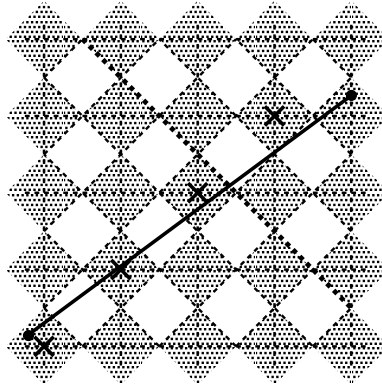


Figure 3.4. Visualization of Bresenham's algorithm. A portion of a line segment is shown. A diamond shaped region of height 1 is placed around each fragment center; those regions that the line segment exits cause rasterization to produce corresponding fragments.

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

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

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

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

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

$$z = (1-t)z_a + tz_b \quad (3.7)$$

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

3.5.2 Other Line Segment Features

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

Line Stipple

The command

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

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

Line stippling masks certain fragments that are produced by rasterization so that they are not sent to the per-fragment stage of the GL. The masking is achieved

using three parameters: the 16-bit line stipple p , the line repeat count r , and an integer stipple counter s . Let

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

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

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

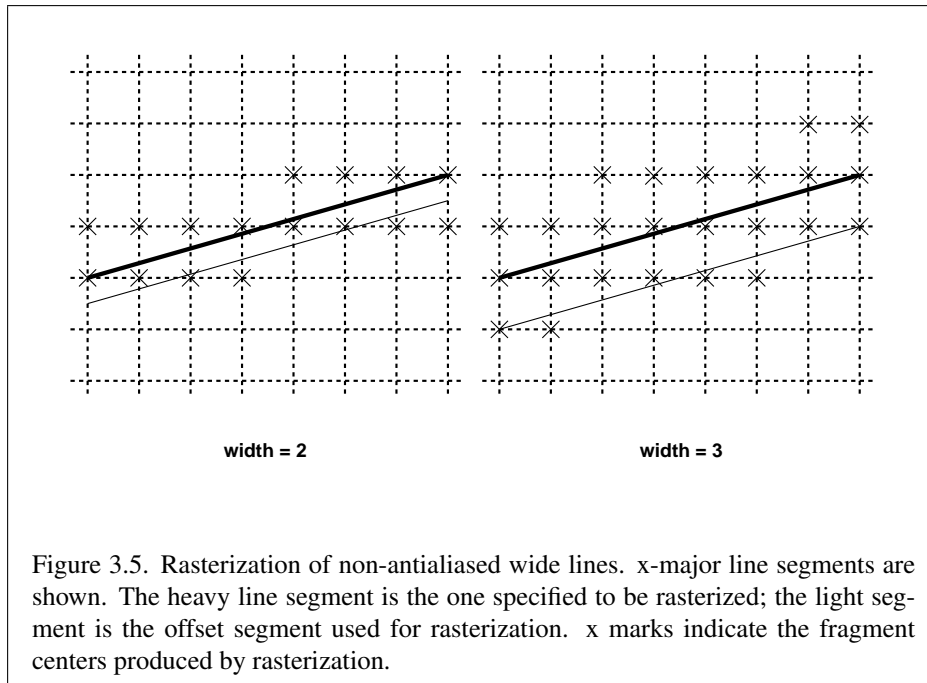
Wide Lines

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

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

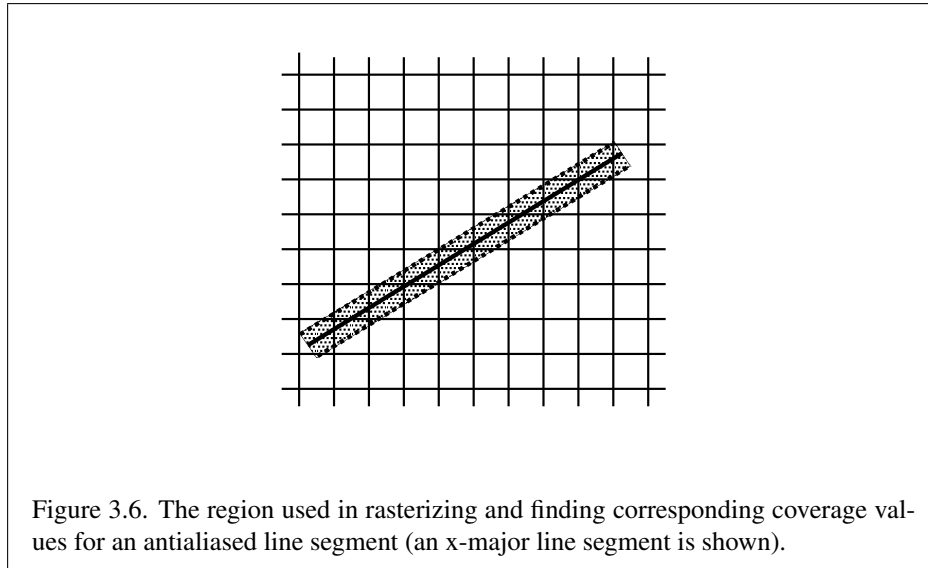
Antialiasing

Rasterized antialiased line segments produce fragments whose fragment squares intersect a rectangle centered on the line segment. Two of the edges are parallel to



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

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



is rasterized as if it were an antialiased polygon, described below (but culling, non-default settings of **PolygonMode**, and polygon stippling are not applied).

3.5.3 Line Rasterization State

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

3.5.4 Line Multisample Rasterization

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

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

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

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

3.6 Polygons

A polygon results from a polygon **Begin/End** object, a triangle resulting from a triangle strip, triangle fan, or series of separate triangles, or a quadrilateral arising from a quadrilateral strip, series of separate quadrilaterals, or a **Rect** command. Like points and line segments, polygon rasterization is controlled by several variables. Polygon antialiasing is controlled with **Enable** and **Disable** with the symbolic constant `POLYGON_SMOOTH`. The analog to line segment stippling for polygons is polygon stippling, described below.

3.6.1 Basic Polygon Rasterization

The first step of polygon rasterization is to determine if the polygon is *back facing* or *front facing*. This determination is made by examining the sign of the area computed by equation 2.6 of section 2.19.1 (including the possible reversal of this sign as indicated by the last call to **FrontFace**). If this sign is positive, the polygon is frontfacing; otherwise, it is back facing. This determination is used in conjunction with the **CullFace** enable bit and mode value to decide whether or not a particular polygon is rasterized. The **CullFace** mode is set by calling

```
void CullFace( enum mode );
```

mode is a symbolic constant: one of `FRONT`, `BACK` or `FRONT_AND_BACK`. Culling is enabled or disabled with **Enable** or **Disable** using the symbolic constant `CULL_FACE`. Front facing polygons are rasterized if either culling is disabled or

the **CullFace** mode is `BACK` while back facing polygons are rasterized only if either culling is disabled or the **CullFace** mode is `FRONT`. The initial setting of the **CullFace** mode is `BACK`. Initially, culling is disabled.

The rule for determining which fragments are produced by polygon rasterization is called *point sampling*. The two-dimensional projection obtained by taking the x and y window coordinates of the polygon's vertices is formed. Fragment centers that lie inside of this polygon are produced by rasterization. Special treatment is given to a fragment whose center lies on a polygon boundary edge. In such a case we require that if two polygons lie on either side of a common edge (with identical endpoints) on which a fragment center lies, then exactly one of the polygons results in the production of the fragment during rasterization.

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

$$p = ap_a + bp_b + cp_c,$$

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

$$a = \frac{A(pp_bp_c)}{A(p_ap_bp_c)}, \quad b = \frac{A(pp_ap_c)}{A(p_ap_bp_c)}, \quad c = \frac{A(pp_ap_b)}{A(p_ap_bp_c)},$$

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

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

$$f = \frac{af_a/w_a + bf_b/w_b + cf_c/w_c}{a/w_a + b/w_b + c/w_c} \quad (3.8)$$

where w_a , w_b and w_c are the clip w coordinates of p_a , p_b , and p_c , respectively. a , b , and c are the barycentric coordinates of the fragment for which the data are produced. a , b , and c must correspond precisely to the exact coordinates of the center of the fragment. Another way of saying this is that the data associated with a fragment must be sampled at the fragment's center. However, depth values for polygons must be interpolated by

$$z = az_a + bz_b + cz_c,$$

where z_a , z_b , and z_c are the depth values of p_a , p_b , and p_c , respectively.

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

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

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

One algorithm that achieves the required behavior is to triangulate a polygon (without adding any vertices) and then treat each triangle individually as already discussed. A scan-line rasterizer that linearly interpolates data along each edge and then linearly interpolates data across each horizontal span from edge to edge also satisfies the restrictions (in this case, the numerator and denominator of equation 3.8 should be iterated independently and a division performed for each fragment).

3.6.2 Stippling

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

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

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

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

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

3.6.3 Antialiasing

Polygon antialiasing rasterizes a polygon by producing a fragment wherever the interior of the polygon intersects that fragment's square. A coverage value is computed at each such fragment, and this value is saved to be applied as described in section 3.13. An associated datum is assigned to a fragment by integrating the datum's value over the region of the intersection of the fragment square with the polygon's interior and dividing this integrated value by the area of the intersection. For a fragment square lying entirely within the polygon, the value of a datum at the fragment's center may be used instead of integrating the value across the fragment.

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

3.6.4 Options Controlling Polygon Rasterization

The interpretation of polygons for rasterization is controlled using

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

face is one of FRONT, BACK, or FRONT_AND_BACK, indicating that the rasterizing method described by *mode* replaces the rasterizing method for front facing polygons, back facing polygons, or both front and back facing polygons, respectively. *mode* is one of the symbolic constants POINT, LINE, or FILL. Calling **PolygonMode** with POINT causes certain vertices of a polygon to be treated, for rasterization purposes, just as if they were enclosed within a **Begin**(POINT) and **End** pair. The vertices selected for this treatment are those that have been tagged as having a polygon boundary edge beginning on them (see section 2.6.2). LINE causes edges that are tagged as boundary to be rasterized as line segments. (The line stipple counter is reset at the beginning of the first rasterized edge of the polygon, but not for subsequent edges.) FILL is the default mode of polygon rasterization, corresponding to the description in sections 3.6.1, 3.6.2, and 3.6.3. Note that these modes affect only the final rasterization of polygons: in particular, a polygon's vertices are lit, and the polygon is clipped and possibly culled before these modes are applied.

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

3.6.5 Depth Offset

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

```
void PolygonOffset( float factor, float units );
```

factor scales the maximum depth slope of the polygon, and *units* scales an implementation dependent constant that relates to the usable resolution of the depth buffer. The resulting values are summed to produce the polygon offset value. Both *factor* and *units* may be either positive or negative.

The maximum depth slope m of a triangle is

$$m = \sqrt{\left(\frac{\partial z_w}{\partial x_w}\right)^2 + \left(\frac{\partial z_w}{\partial y_w}\right)^2} \quad (3.9)$$

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

$$m = \max \left\{ \left| \frac{\partial z_w}{\partial x_w} \right|, \left| \frac{\partial z_w}{\partial y_w} \right| \right\}. \quad (3.10)$$

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

The minimum resolvable difference r is an implementation-dependent parameter that depends on the depth buffer representation. It is the smallest difference in window coordinate z values that is guaranteed to remain distinct throughout polygon rasterization and in the depth buffer. All pairs of fragments generated by the rasterization of two polygons with otherwise identical vertices, but z_w values that differ by r , will have distinct depth values.

For fixed-point depth buffer representations, r is constant throughout the range of the entire depth buffer. For floating-point depth buffers, there is no single minimum resolvable difference. In this case, the minimum resolvable difference for a given polygon is dependent on the maximum exponent, e , in the range of z values spanned by the primitive. If n is the number of bits in the floating-point mantissa, the minimum resolvable difference, r , for the given primitive is defined as

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

The offset value o for a polygon is

$$o = m \times factor + r \times units. \quad (3.11)$$

m is computed as described above. If the depth buffer uses a fixed-point representation, m is a function of depth values in the range $[0, 1]$, and o is applied to depth values in the same range.

Boolean state values `POLYGON_OFFSET_POINT`, `POLYGON_OFFSET_LINE`, and `POLYGON_OFFSET_FILL` determine whether o is applied during the rasterization of polygons in `POINT`, `LINE`, and `FILL` modes. These boolean state values are enabled and disabled as argument values to the commands **Enable** and **Disable**. If `POLYGON_OFFSET_POINT` is enabled, o is added to the depth value of each fragment produced by the rasterization of a polygon in `POINT` mode. Likewise, if `POLYGON_OFFSET_LINE` or `POLYGON_OFFSET_FILL` is enabled, o is added to the depth value of each fragment produced by the rasterization of a polygon in `LINE` or `FILL` modes, respectively.

For fixed-point depth buffers, fragment depth values are always limited to the range $[0, 1]$, either by clamping after offset addition is performed (preferred), or by clamping the vertex values used in the rasterization of the polygon. Fragment depth values are clamped even when the depth buffer uses a floating-point representation.

3.6.6 Polygon Multisample Rasterization

If `MULTISAMPLE` is enabled and the value of `SAMPLE_BUFFERS` is one, then polygons are rasterized using the following algorithm, regardless of whether polygon antialiasing (`POLYGON_SMOOTH`) is enabled or disabled. Polygon rasterization produces a fragment for each framebuffer pixel with one or more sample points that satisfy the point sampling criteria described in section 3.6.1, including the special treatment for sample points that lie on a polygon boundary edge. If a polygon is culled, based on its orientation and the **CullFace** mode, then no fragments are produced during rasterization. Fragments are culled by the polygon stipple just as they are for aliased and antialiased polygons.

Coverage bits that correspond to sample points that satisfy the point sampling criteria are 1, other coverage bits are 0. Each color, depth, and set of texture coordinates is produced by substituting the corresponding sample location into the barycentric equations described in section 3.6.1, using the approximation to equation 3.8 that omits w components. An implementation may choose to assign the same color value and the same set of texture coordinates to more than one sample by barycentric evaluation using any location with the pixel including the fragment

center or one of the sample locations. The color value and the set of texture coordinates need not be evaluated at the same location.

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

3.6.7 Polygon Rasterization State

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

3.7 Pixel Rectangles

Rectangles of color, depth, and certain other values may be converted to fragments using the **DrawPixels** command (described in section 3.7.4). Some of the parameters and operations governing the operation of **DrawPixels** are shared by **ReadPixels** (used to obtain pixel values from the framebuffer) and **CopyPixels** (used to copy pixels from one framebuffer location to another); the discussion of **ReadPixels** and **CopyPixels**, however, is deferred until chapter 4 after the framebuffer has been discussed in detail. Nevertheless, we note in this section when parameters and state pertaining to **DrawPixels** also pertain to **ReadPixels** or **CopyPixels**.

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

3.7.1 Pixel Storage Modes and Pixel Buffer Objects

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

Parameter Name	Type	Initial Value	Valid Range
UNPACK_SWAP_BYTES	boolean	FALSE	TRUE/FALSE
UNPACK_LSB_FIRST	boolean	FALSE	TRUE/FALSE
UNPACK_ROW_LENGTH	integer	0	$[0, \infty)$
UNPACK_SKIP_ROWS	integer	0	$[0, \infty)$
UNPACK_SKIP_PIXELS	integer	0	$[0, \infty)$
UNPACK_ALIGNMENT	integer	4	1,2,4,8
UNPACK_IMAGE_HEIGHT	integer	0	$[0, \infty)$
UNPACK_SKIP_IMAGES	integer	0	$[0, \infty)$

Table 3.1: **PixelStore** parameters pertaining to one or more of **DrawPixels**, **ColorTable**, **ColorSubTable**, **ConvolutionFilter1D**, **ConvolutionFilter2D**, **SeparableFilter2D**, **PolygonStipple**, **TexImage1D**, **TexImage2D**, **TexImage3D**, **TexSubImage1D**, **TexSubImage2D**, and **TexSubImage3D**.

```
void PixelStore{if}( enum pname, T param );
```

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

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

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

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

3.7.2 The Imaging Subset

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

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

1. Color tables, including all commands and enumerants described in subsections **Color Table Specification**, **Alternate Color Table Specification Commands**, **Color Table State and Proxy State**, **Color Table Lookup**, **Post Convolution Color Table Lookup**, and **Post Color Matrix Color Table Lookup**, as well as the query commands described in section 6.1.7.
2. Convolution, including all commands and enumerants described in subsections **Convolution Filter Specification**, **Alternate Convolution Filter Specification Commands**, and **Convolution**, as well as the query commands described in section 6.1.8.
3. Color matrix, including all commands and enumerants described in subsections **Color Matrix Specification** and **Color Matrix Transformation**, as well as the simple query commands described in section 6.1.6.
4. Histogram and minmax, including all commands and enumerants described in subsections **Histogram Table Specification**, **Histogram State and Proxy State**, **Histogram**, **Minmax Table Specification**, and **Minmax**, as well as the query commands described in section 6.1.9 and section 6.1.10.

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

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

3.7.3 Pixel Transfer Modes

Pixel transfer modes affect the operation of **DrawPixels** (section 3.7.4), **ReadPixels** (section 4.3.2), and **CopyPixels** (section 4.3.3) at the time when one of these

Parameter Name	Type	Initial Value	Valid Range
MAP_COLOR	boolean	FALSE	TRUE/FALSE
MAP_STENCIL	boolean	FALSE	TRUE/FALSE
INDEX_SHIFT	integer	0	$(-\infty, \infty)$
INDEX_OFFSET	integer	0	$(-\infty, \infty)$
x_SCALE	float	1.0	$(-\infty, \infty)$
DEPTH_SCALE	float	1.0	$(-\infty, \infty)$
x_BIAS	float	0.0	$(-\infty, \infty)$
DEPTH_BIAS	float	0.0	$(-\infty, \infty)$
POST_CONVOLUTION_ x_SCALE	float	1.0	$(-\infty, \infty)$
POST_CONVOLUTION_ x_BIAS	float	0.0	$(-\infty, \infty)$
POST_COLOR_MATRIX_ x_SCALE	float	1.0	$(-\infty, \infty)$
POST_COLOR_MATRIX_ x_BIAS	float	0.0	$(-\infty, \infty)$

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

commands is executed (which may differ from the time the command is issued). Some pixel transfer modes are set with

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

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

The pixel map lookup tables are set with

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

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

The entries of a table may be specified using one of three types: single-precision floating-point, unsigned short integer, or unsigned integer, depending on which of the three versions of **PixelMap** is called. A table entry is converted to the appropriate type when it is specified. An entry giving a color component value is converted according to table 2.10 and then clamped to the range $[0, 1]$. An entry giving a color index value is converted from an unsigned short integer or unsigned

Map Name	Address	Value	Init. Size	Init. Value
PIXEL_MAP_I_TO_I	color idx	color idx	1	0.0
PIXEL_MAP_S_TO_S	stencil idx	stencil idx	1	0
PIXEL_MAP_I_TO_R	color idx	R	1	0.0
PIXEL_MAP_I_TO_G	color idx	G	1	0.0
PIXEL_MAP_I_TO_B	color idx	B	1	0.0
PIXEL_MAP_I_TO_A	color idx	A	1	0.0
PIXEL_MAP_R_TO_R	R	R	1	0.0
PIXEL_MAP_G_TO_G	G	G	1	0.0
PIXEL_MAP_B_TO_B	B	B	1	0.0
PIXEL_MAP_A_TO_A	A	A	1	0.0

Table 3.3: **PixelMap** parameters.

integer to floating-point. An entry giving a stencil index is converted from single-precision floating-point to an integer by rounding to nearest. The various tables and their initial sizes and entries are summarized in table 3.3. A table that takes an index as an address must have $size = 2^n$ or the error `INVALID_VALUE` results. The maximum allowable $size$ of each table is specified by the implementation dependent value `MAX_PIXEL_MAP_TABLE`, but must be at least 32 (a single maximum applies to all tables). The error `INVALID_VALUE` is generated if a $size$ larger than the implemented maximum, or less than one, is given to **PixelMap**.

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

Color Table Specification

Color lookup tables are specified with

Table Name	Type
COLOR_TABLE POST_CONVOLUTION_COLOR_TABLE POST_COLOR_MATRIX_COLOR_TABLE	regular
PROXY_COLOR_TABLE PROXY_POST_CONVOLUTION_COLOR_TABLE PROXY_POST_COLOR_MATRIX_COLOR_TABLE	proxy

Table 3.4: Color table names. Regular tables have associated image data. Proxy tables have no image data, and are used only to determine if an image can be loaded into the corresponding regular table.

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

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

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

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

The color lookup table is redefined to have *width* entries, each with the specified internal format. The table is formed with indices 0 through *width* - 1. Table

location i is specified by the i th image pixel, counting from zero.

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

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

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

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

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

Alternate Color Table Specification Commands

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

The command

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

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

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

Two additional commands,

```

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

```

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

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

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

Calling **CopyColorTable** or **CopyColorSubTable** will result in an `INVALID_FRAMEBUFFER_OPERATION` error if the object bound to `READ_FRAMEBUFFER_BINDING` is not framebuffer complete (see section 4.4.4).

Color Table State and Proxy State

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

In addition to the color lookup tables, partially instantiated proxy color lookup tables are maintained. Each proxy table includes width and internal format state values, as well as state for the red, green, blue, alpha, luminance, and intensity component resolutions. Proxy tables do not include image data, nor do they in-

clude scale and bias parameters. When **ColorTable** is executed with *target* specified as one of the proxy color table names listed in table 3.4, the proxy state values of the table are recomputed and updated. If the table is too large, no error is generated, but the proxy format, width and component resolutions are set to zero. If the color table would be accommodated by **ColorTable** called with *target* set to the corresponding regular table name (COLOR_TABLE is the regular name corresponding to PROXY_COLOR_TABLE, for example), the proxy state values are set exactly as though the regular table were being specified. Calling **ColorTable** with a proxy *target* has no effect on the image or state of any actual color table.

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

Convolution Filter Specification

A two-dimensional convolution filter image is specified by calling

```
void ConvolutionFilter2D( enum target, enum internalformat,
                          sizei width, sizei height, enum format, enum type,
                          void *data );
```

target must be CONVOLUTION_2D. *width*, *height*, *format*, *type*, and *data* specify an image in memory with the same meaning and allowed values as the corresponding parameters to **DrawPixels**. The *formats* COLOR_INDEX, DEPTH_COMPONENT, DEPTH_STENCIL, and STENCIL_INDEX and the *type* BITMAP are not allowed.

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

Components are then selected from the resulting R, G, B, and A values to obtain a table with the *base internal format* specified by (or derived from) *internalformat*, in the same manner as for textures (section 3.9.1). *internalformat* accepts the same values as the corresponding argument of **ColorTable**.

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

Image location i, j is specified by the N th pixel, counting from zero, where

$$N = i + j * width$$

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

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

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

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

A one-dimensional convolution filter is defined using

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

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

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

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

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

Special facilities are provided for the definition of two-dimensional *separable* filters – filters whose image can be represented as the product of two one-dimensional images, rather than as full two-dimensional images. A two-dimensional separable convolution filter is specified with


```
void SeparableFilter2D( enum target, enum internalformat,
                        sizei width, sizei height, enum format, enum type,
                        void *row, void *column );
```

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

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

Alternate Convolution Filter Specification Commands

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

The command

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

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

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

The command

```
void CopyConvolutionFilter1D( enum target,
                               enum internalformat, int x, int y, sizei width );
```

defines a one-dimensional filter in exactly the manner of **ConvolutionFilter1D**, except that image data are taken from the framebuffer, rather than from client memory. *target* must be `CONVOLUTION_1D`. *x*, *y*, and *width* correspond precisely to the corresponding arguments of **CopyPixels** (refer to section 4.3.3); they specify the image's *width* and the lower left (*x*, *y*) coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to **CopyPixels** with argument *type* set to `COLOR` and *height* set to 1, stopping after the final expansion to `RGBA`.

Subsequent processing is identical to that described for **ConvolutionFilter1D**, beginning with scaling by `CONVOLUTION_FILTER_SCALE`. Parameters *target*, *internalformat*, and *width* are specified using the same values, with the same meanings, as the equivalent arguments of **ConvolutionFilter2D**. *format* is taken to be `RGBA`.

Calling **CopyConvolutionFilter1D** or **CopyConvolutionFilter2D** will result in an `INVALID_FRAMEBUFFER_OPERATION` error if the object bound to `READ_FRAMEBUFFER_BINDING` is not framebuffer complete (see section 4.4.4).

Convolution Filter State

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

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

Color Matrix Specification

Setting the matrix mode to `COLOR` causes the matrix operations described in section 2.12.2 to apply to the top matrix on the color matrix stack. All matrix operations have the same effect on the color matrix as they do on the other matrices.

Histogram Table Specification

The histogram table is specified with

```
void Histogram( enum target, sizei width,
                enum internalformat, boolean sink );
```

target must be `HISTOGRAM` if a histogram table is to be specified. *target* value `PROXY_HISTOGRAM` is a special case discussed later in this section. *width* specifies the number of entries in the histogram table, and *internalformat* specifies the format of each table entry. The maximum allowable *width* of the histogram table is implementation-dependent, but must be at least 32. *sink* specifies whether pixel groups will be consumed by the histogram operation (`TRUE`) or passed on to the minmax operation (`FALSE`).

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

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

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

Histogram State and Proxy State

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

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

PROXY_HISTOGRAM has no effect on the actual histogram table.

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

Minmax Table Specification

The minmax table is specified with

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

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

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

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

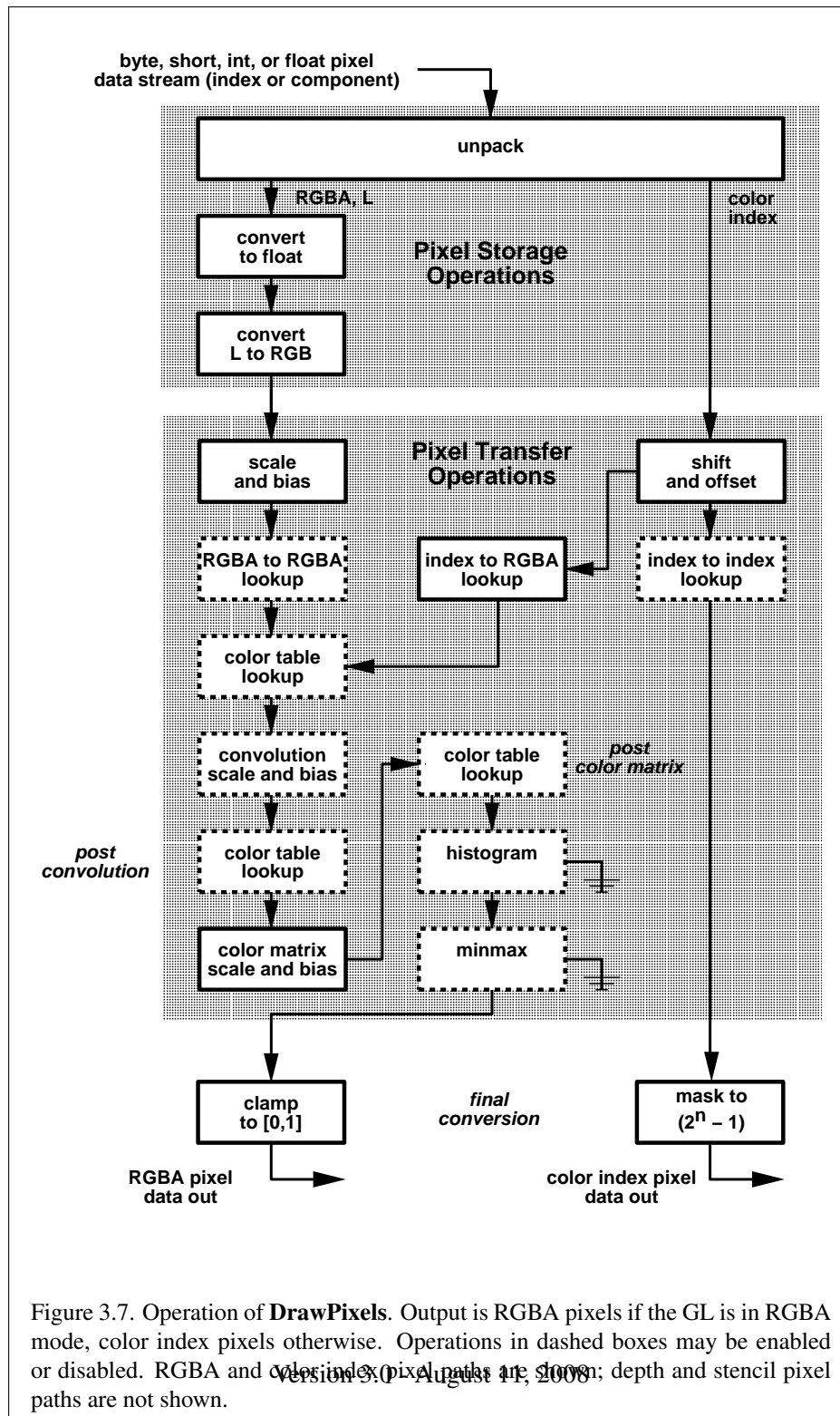
3.7.4 Rasterization of Pixel Rectangles

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

Pixels are drawn using

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

format is a symbolic constant indicating what the values in memory represent. *width* and *height* are the width and height, respectively, of the pixel rectangle to be drawn. *data* refers to the data to be drawn. The correspondence between the *type* token values and the GL data types they indicate is



given in table 3.5. If the GL is in color index mode and *format* is not one of `COLOR_INDEX`, `STENCIL_INDEX`, `DEPTH_COMPONENT`, or `DEPTH_STENCIL`, then the error `INVALID_OPERATION` occurs. Results of rasterization are undefined if any of the selected draw buffers of the draw framebuffer have an integer format and no fragment shader is active. If *format* contains integer components, as shown in table 3.6, an `INVALID_OPERATION` error is generated. If *type* is `BITMAP` and *format* is not `COLOR_INDEX` or `STENCIL_INDEX` then the error `INVALID_ENUM` occurs. If *format* is `DEPTH_STENCIL` and *type* is not `UNSIGNED_INT_24_8` or `FLOAT_32_UNSIGNED_INT_24_8_REV`, then the error `INVALID_ENUM` occurs. If *format* is one of the integer component formats as defined in table 3.6 and *type* is `FLOAT`, the error `INVALID_ENUM` occurs. Some additional constraints on the combinations of *format* and *type* values that are accepted are discussed below.

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

Unpacking

Data are taken from the currently bound pixel unpack buffer or client memory as a sequence of signed or unsigned bytes (GL data types `byte` and `ubyte`), signed or unsigned short integers (GL data types `short` and `ushort`), signed or unsigned integers (GL data types `int` and `uint`), or floating point values (GL data types `half` and `float`). These elements are grouped into sets of one, two, three, or four values, depending on the *format*, to form a group. Table 3.6 summarizes the format of groups obtained from memory; it also indicates those formats that yield indices and those that yield floating-point or integer components.

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

By default the values of each GL data type are interpreted as they would be specified in the language of the client's GL binding. If `UNPACK_SWAP_BYTES` is enabled, however, then the values are interpreted with the bit orderings modified as per table 3.7. The modified bit orderings are defined only if the GL data type

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

Table 3.5: **DrawPixels** and **ReadPixels** *type* parameter values and the corresponding GL data types. Refer to table 2.2 for definitions of GL data types. Special interpretations are described near the end of section 3.7.4.

Format Name	Element Meaning and Order	Target Buffer
COLOR_INDEX	Color Index	Color
STENCIL_INDEX	Stencil Index	Stencil
DEPTH_COMPONENT	Depth	Depth
DEPTH_STENCIL	Depth and Stencil Index	Depth and Stencil
RED	R	Color
GREEN	G	Color
BLUE	B	Color
ALPHA	A	Color
RG	R, G	Color
RGB	R, G, B	Color
RGBA	R, G, B, A	Color
BGR	B, G, R	Color
BGRA	B, G, R, A	Color
LUMINANCE	Luminance	Color
LUMINANCE_ALPHA	Luminance, A	Color
RED_INTEGER	iR	Color
GREEN_INTEGER	iG	Color
BLUE_INTEGER	iB	Color
ALPHA_INTEGER	iA	Color
RG_INTEGER	iR, iG	Color
RGB_INTEGER	iR, iG, iB	Color
RGBA_INTEGER	iR, iG, iB, iA	Color
BGR_INTEGER	iB, iG, iR	Color
BGRA_INTEGER	iB, iG, iR, iA	Color

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

Element Size	Default Bit Ordering	Modified Bit Ordering
8 bit	[7..0]	[7..0]
16 bit	[15..0]	[7..0][15..8]
32 bit	[31..0]	[7..0][15..8][23..16][31..24]

Table 3.7: Bit ordering modification of elements when `UNPACK_SWAP_BYTES` is enabled. These reorderings are defined only when GL data type `ubyte` has 8 bits, and then only for GL data types with 8, 16, or 32 bits. Bit 0 is the least significant.

`ubyte` has eight bits, and then for each specific GL data type only if that type is represented with 8, 16, or 32 bits.

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

$$p + Nk \quad (3.12)$$

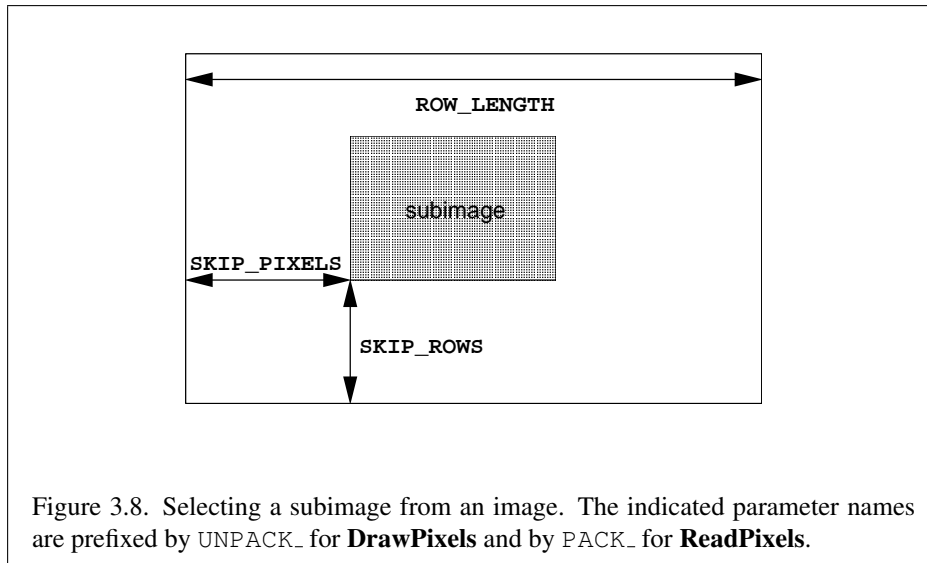
where N is the row number (counting from zero) and k is defined as

$$k = \begin{cases} nl & s \geq a, \\ a/s \lceil snl/a \rceil & s < a \end{cases} \quad (3.13)$$

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

There is a mechanism for selecting a sub-rectangle of groups from a larger containing rectangle. This mechanism relies on three integer parameters: `UNPACK_ROW_LENGTH`, `UNPACK_SKIP_ROWS`, and `UNPACK_SKIP_PIXELS`. Before obtaining the first group from memory, the pointer supplied to **DrawPixels** is effectively advanced by $(\text{UNPACK_SKIP_PIXELS})n + (\text{UNPACK_SKIP_ROWS})k$ elements. Then *width* groups are obtained from contiguous elements in memory (without advancing the pointer), after which the pointer is advanced by k elements. *height* sets of *width* groups of values are obtained this way. See figure 3.8.

Calling **DrawPixels** with a *type* matching one of the types in table 3.8 is a special case in which all the components of each group are packed into a sin-



gle 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 two 32-bit words; the first word contains the float component, and the second word contains packed 24-bit and 8-bit components. The number of components per packed pixel is fixed by the type, and must match the number of components per group indicated by the *format* parameter, as listed in table 3.8. The error `INVALID_OPERATION` is generated if a mismatch occurs. This constraint also holds for all other functions that accept or return pixel data using *type* and *format* parameters to define the type and format of that data.

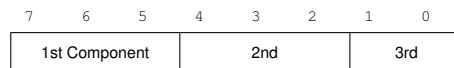
Bitfield locations of the first, second, third, and fourth components of each packed pixel type are illustrated in tables 3.9, 3.10, and 3.11. Each bitfield is interpreted as an unsigned integer value. If the base GL type is supported with more than the minimum precision (e.g. a 9-bit byte) the packed components are right-justified in the pixel.

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

<i>type</i> Parameter Token Name	GL Data Type	Number of Components	Matching Pixel Formats
UNSIGNED_BYTE_3_3_2	ubyte	3	RGB
UNSIGNED_BYTE_2_3_3_REV	ubyte	3	RGB
UNSIGNED_SHORT_5_6_5	ushort	3	RGB
UNSIGNED_SHORT_5_6_5_REV	ushort	3	RGB
UNSIGNED_SHORT_4_4_4_4	ushort	4	RGBA,BGRA
UNSIGNED_SHORT_4_4_4_4_REV	ushort	4	RGBA,BGRA
UNSIGNED_SHORT_5_5_5_1	ushort	4	RGBA,BGRA
UNSIGNED_SHORT_1_5_5_5_REV	ushort	4	RGBA,BGRA
UNSIGNED_INT_8_8_8_8	uint	4	RGBA,BGRA
UNSIGNED_INT_8_8_8_8_REV	uint	4	RGBA,BGRA
UNSIGNED_INT_10_10_10_2	uint	4	RGBA,BGRA
UNSIGNED_INT_2_10_10_10_REV	uint	4	RGBA,BGRA
UNSIGNED_INT_24_8	uint	2	DEPTH_STENCIL
UNSIGNED_INT_10F_11F_11F_REV	uint	3	RGB
UNSIGNED_INT_5_9_9_9_REV	uint	4	RGB
FLOAT_32_UNSIGNED_INT_24_8_REV	n/a	2	DEPTH_STENCIL

Table 3.8: Packed pixel formats.

UNSIGNED_BYTE_3_3_2:



UNSIGNED_BYTE_2_3_3_REV:

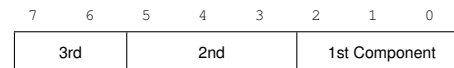
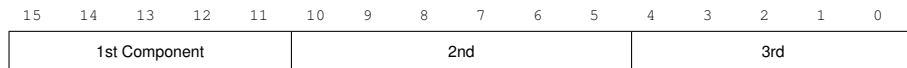
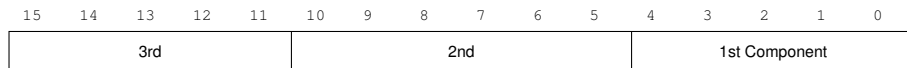


Table 3.9: UNSIGNED_BYTE formats. Bit numbers are indicated for each component.

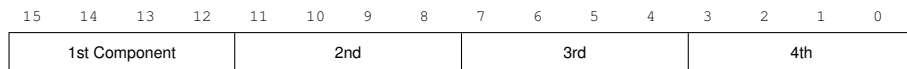
UNSIGNED_SHORT_5_6_5:



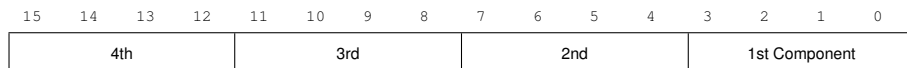
UNSIGNED_SHORT_5_6_5_REV:



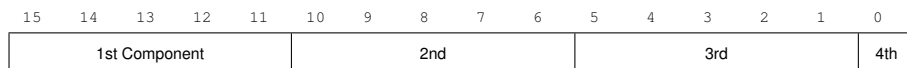
UNSIGNED_SHORT_4_4_4_4:



UNSIGNED_SHORT_4_4_4_4_REV:



UNSIGNED_SHORT_5_5_5_1:



UNSIGNED_SHORT_1_5_5_5_REV:

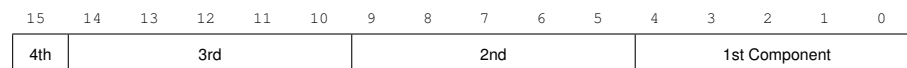


Table 3.10: UNSIGNED_SHORT formats

UNSIGNED_INT_8_8_8_8:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1st Component								2nd								3rd								4th							

UNSIGNED_INT_8_8_8_8_REV:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
4th								3rd								2nd								1st Component							

UNSIGNED_INT_10_10_10_2:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1st Component										2nd										3rd										4th	

UNSIGNED_INT_2_10_10_10_REV:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
4th		3rd										2nd										1st Component									

UNSIGNED_INT_24_8:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1st Component																								2nd							

UNSIGNED_INT_10F_11F_11F_REV:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
3rd										2nd										1st Component											

UNSIGNED_INT_5_9_9_9_REV:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
4th					3rd										2nd										1st Component							

Table 3.11: 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 3.12: Packed pixel field assignments.

The assignment of component to fields in the packed pixel is as described in table 3.12.

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

Calling **DrawPixels** with a type of `UNSIGNED_INT_10F_11F_11F_REV` and format of `RGB` is a special case in which the data are a series of `GLuint` values. Each `uint` value specifies 3 packed components as shown in table 3.11. The 1st, 2nd, and 3rd components are called f_{red} (11 bits), f_{green} (11 bits), and f_{blue} (10 bits) respectively.

f_{red} and f_{green} are treated as unsigned 11-bit floating-point values and converted to floating-point red and green components respectively as described in section 2.1.3. f_{blue} is treated as an unsigned 10-bit floating-point value and converted to a floating-point blue component as described in section 2.1.4.

Calling **DrawPixels** with a type of `UNSIGNED_INT_5_9_9_9_REV` and format of `RGB` is a special case in which the data are a series of `GLuint` values. Each `uint` value specifies 4 packed components as shown in table 3.11. The 1st, 2nd, 3rd, and 4th components are called p_{red} , p_{green} , p_{blue} , and p_{exp} respectively and are treated as unsigned integers. These are then used to compute floating-point RGB components (ignoring the “Conversion to floating-point” section below in this case) as follows:

$$\begin{aligned}
 red &= p_{red} 2^{p_{exp}-B-N} \\
 green &= p_{green} 2^{p_{exp}-B-N} \\
 blue &= p_{blue} 2^{p_{exp}-B-N}
 \end{aligned}$$

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

Calling **DrawPixels** with a *type* of `BITMAP` is a special case in which the data are a series of GL `ubyte` values. Each `ubyte` value specifies 8 1-bit elements with its 8 least-significant bits. The 8 single-bit elements are ordered from most significant to least significant if the value of `UNPACK_LSB_FIRST` is `FALSE`; otherwise, the ordering is from least significant to most significant. The values of bits other than the 8 least significant in each `ubyte` are not significant.

The first element of the first row is the first bit (as defined above) of the `ubyte` pointed to by the pointer passed to **DrawPixels**. The first element of the second row is the first bit (again as defined above) of the `ubyte` at location $p + k$, where k is computed as

$$k = a \left\lceil \frac{l}{8a} \right\rceil \quad (3.14)$$

There is a mechanism for selecting a sub-rectangle of elements from a `BITMAP` image as well. Before obtaining the first element from memory, the pointer supplied to **DrawPixels** is effectively advanced by `UNPACK_SKIP_ROWS * k` `ubbytes`. Then `UNPACK_SKIP_PIXELS` 1-bit elements are ignored, and the subsequent *width* 1-bit elements are obtained, without advancing the `ubyte` pointer, after which the pointer is advanced by k `ubbytes`. *height* sets of *width* elements are obtained this way.

Conversion to floating-point

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

Each element in a group is converted to a floating-point value according to the appropriate formula in table 2.10 (section 2.19). For packed pixel types, each element in the group is converted by computing $c / (2^N - 1)$, where c is the unsigned integer value of the bitfield containing the element and N is the number of bits in the bitfield.

Conversion to RGB

This step is applied only if the *format* is `LUMINANCE` or `LUMINANCE_ALPHA`. If the *format* is `LUMINANCE`, then each group of one element is converted to a group of R, G, and B (three) elements by copying the original single element into each of the three new elements. If the *format* is `LUMINANCE_ALPHA`, then each group of two elements is converted to a group of R, G, B, and A (four) elements by copying

the first original element into each of the first three new elements and copying the second original element to the A (fourth) new element.

Final Expansion to RGBA

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

Pixel Transfer Operations

This step is actually a sequence of steps. Because the pixel transfer operations are performed equivalently during the drawing, copying, and reading of pixels, and during the specification of texture images (either from memory or from the framebuffer), they are described separately in section 3.7.5. After the processing described in that section is completed, groups are processed as described in the following sections.

Final Conversion

For a color index, final conversion consists of masking the bits of the index to the left of the binary point by $2^n - 1$, where n is the number of bits in an index buffer.

For integer RGBA components, no conversion is performed. For floating-point RGBA components, if fragment color clamping is enabled, each element is clamped to $[0, 1]$, and may be converted to fixed-point according to the rules given in section 2.19.9. If fragment color clamping is disabled, RGBA components are unmodified. Fragment color clamping is controlled using **ClampColor**, as described in section 2.19.6, with a *target* of `CLAMP_FRAGMENT_COLOR`.

For a depth component, an element is processed according to the depth buffer's representation. For fixed-point depth buffers, the element is first clamped to the range $[0, 1]$ and then converted to fixed-point as if it were a window z value (see section 2.12.1). Conversion is not necessary when the depth buffer uses a floating-point representation, but clamping is.

Stencil indices are masked by $2^n - 1$, where n is the number of bits in the stencil buffer.

The state required for fragment color clamping is a three-valued integer. The initial value of fragment color clamping is `FIXED_ONLY`.

Conversion to Fragments

The conversion of a group to fragments is controlled with

```
void PixelZoom( float  $z_x$ , float  $z_y$  );
```

Let (x_{rp}, y_{rp}) be the current raster position (section 2.18). (If the current raster position is invalid, then **DrawPixels** is ignored; pixel transfer operations do not update the histogram or minmax tables, and no fragments are generated. However, the histogram and minmax tables are updated even if the corresponding fragments are later rejected by the pixel ownership (section 4.1.1) or scissor (section 4.1.2) tests.) If a particular group (index or components) is the n th in a row and belongs to the m th row, consider the region in window coordinates bounded by the rectangle with corners

$$(x_{rp} + z_x n, y_{rp} + z_y m) \quad \text{and} \quad (x_{rp} + z_x (n + 1), y_{rp} + z_y (m + 1))$$

(either z_x or z_y may be negative). A fragment representing group (n, m) is produced for each framebuffer pixel inside, or on the bottom or left boundary, of this rectangle

A fragment arising from a group consisting of color data takes on the color index or color components of the group and the current raster position's associated depth value, while a fragment arising from a depth component takes that component's depth value and the current raster position's associated color index or color components. In both cases, the fog coordinate is taken from the current raster position's associated raster distance, the secondary color is taken from the current raster position's associated secondary color, and texture coordinates are taken from the current raster position's associated texture coordinates. Groups arising from **DrawPixels** with a *format* of DEPTH_STENCIL or STENCIL_INDEX are treated specially and are described in section 4.3.1.

3.7.5 Pixel Transfer Operations

The GL defines six kinds of pixel groups:

1. *Floating-point RGBA component*: Each group comprises four color components in floating-point format: red, green, blue, and alpha.
2. *Integer RGBA component*: Each group comprises four color components in integer format: red, green, blue, and alpha.
3. *Depth component*: Each group comprises a single depth component.

4. *Color index*: Each group comprises a single color index.
5. *Stencil index*: Each group comprises a single stencil index.
6. *Depth/stencil*: Each group comprises a single depth component and a single stencil index.

Each operation described in this section is applied sequentially to each pixel group in an image. Many operations are applied only to pixel groups of certain kinds; if an operation is not applicable to a given group, it is skipped. None of the operations defined in this section affect integer RGBA component pixel groups.

Arithmetic on Components

This step applies only to RGBA component and depth component groups, and to the depth components in depth/stencil groups. Each component is multiplied by an appropriate signed scale factor: `RED_SCALE` for an R component, `GREEN_SCALE` for a G component, `BLUE_SCALE` for a B component, and `ALPHA_SCALE` for an A component, or `DEPTH_SCALE` for a depth component. Then the result is added to the appropriate signed bias: `RED_BIAS`, `GREEN_BIAS`, `BLUE_BIAS`, `ALPHA_BIAS`, or `DEPTH_BIAS`.

Arithmetic on Indices

This step applies only to color index and stencil index groups, and to the stencil indices in depth/stencil groups. If the index is a floating-point value, it is converted to fixed-point, with an unspecified number of bits to the right of the binary point and at least $\lceil \log_2(\text{MAX_PIXEL_MAP_TABLE}) \rceil$ bits to the left of the binary point. Indices that are already integers remain so; any fraction bits in the resulting fixed-point value are zero.

The fixed-point index is then shifted by $|\text{INDEX_SHIFT}|$ bits, left if `INDEX_SHIFT` > 0 and right otherwise. In either case the shift is zero-filled. Then, the signed integer offset `INDEX_OFFSET` is added to the index.

RGBA to RGBA Lookup

This step applies only to RGBA component groups, and is skipped if `MAP_COLOR` is `FALSE`. First, each component is clamped to the range $[0, 1]$. There is a table associated with each of the R, G, B, and A component elements: `PIXEL_MAP_R_TO_R` for R, `PIXEL_MAP_G_TO_G` for G, `PIXEL_MAP_B_TO_B` for B, and `PIXEL_MAP_A_TO_A` for A. Each element is multiplied by an integer one less than the size of the corresponding table, and, for each element, an address is found by rounding this value

to the nearest integer. For each element, the addressed value in the corresponding table replaces the element.

Color Index Lookup

This step applies only to color index groups. If the GL command that invokes the pixel transfer operation requires that RGBA component pixel groups be generated, then a conversion is performed at this step. RGBA component pixel groups are required if

1. The groups will be rasterized, and the GL is in RGBA mode, or
2. The groups will be loaded as an image into texture memory, or
3. The groups will be returned to client memory with a format other than `COLOR_INDEX`.

If RGBA component groups are required, then the integer part of the index is used to reference 4 tables of color components: `PIXEL_MAP_I_TO_R`, `PIXEL_MAP_I_TO_G`, `PIXEL_MAP_I_TO_B`, and `PIXEL_MAP_I_TO_A`. Each of these tables must have 2^n entries for some integer value of n (n may be different for each table). For each table, the index is first rounded to the nearest integer; the result is ANDed with $2^n - 1$, and the resulting value used as an address into the table. The indexed value becomes an R, G, B, or A value, as appropriate. The group of four elements so obtained replaces the index, changing the group's type to RGBA component.

If RGBA component groups are not required, and if `MAP_COLOR` is enabled, then the index is looked up in the `PIXEL_MAP_I_TO_I` table (otherwise, the index is not looked up). Again, the table must have 2^n entries for some integer n . The index is first rounded to the nearest integer; the result is ANDed with $2^n - 1$, and the resulting value used as an address into the table. The value in the table replaces the index. The floating-point table value is first rounded to a fixed-point value with unspecified precision. The group's type remains color index.

Stencil Index Lookup

This step applies only to stencil index groups, and to the stencil indices in depth/stencil groups. If `MAP_STENCIL` is enabled, then the index is looked up in the `PIXEL_MAP_S_TO_S` table (otherwise, the index is not looked up). The table must have 2^n entries for some integer n . The integer index is ANDed with $2^n - 1$, and the resulting value used as an address into the table. The integer value in the table replaces the index.

Base Internal Format	R	G	B	A
ALPHA				A_t
LUMINANCE	L_t	L_t	L_t	
LUMINANCE_ALPHA	L_t	L_t	L_t	A_t
INTENSITY	I_t	I_t	I_t	I_t
RGB	R_t	G_t	B_t	
RGBA	R_t	G_t	B_t	A_t

Table 3.13: Color table lookup. R_t , G_t , B_t , A_t , L_t , and I_t are color table values that are assigned to pixel components R , G , B , and A depending on the table format. When there is no assignment, the component value is left unchanged by lookup.

Color Table Lookup

This step applies only to RGBA component groups. Color table lookup is only done if `COLOR_TABLE` is enabled. If a zero-width table is enabled, no lookup is performed.

The internal format of the table determines which components of the group will be replaced (see table 3.13). The components to be replaced are converted to indices by clamping to $[0, 1]$, multiplying by an integer one less than the width of the table, and rounding to the nearest integer. Components are replaced by the table entry at the index.

The required state is one bit indicating whether color table lookup is enabled or disabled. In the initial state, lookup is disabled.

Convolution

This step applies only to RGBA component groups. If `CONVOLUTION_1D` is enabled, the one-dimensional convolution filter is applied only to the one-dimensional texture images passed to **TexImage1D**, **TexSubImage1D**, **CopyTexImage1D**, and **CopyTexSubImage1D**. If `CONVOLUTION_2D` is enabled, the two-dimensional convolution filter is applied only to the two-dimensional images passed to **DrawPixels**, **CopyPixels**, **ReadPixels**, **TexImage2D**, **TexSubImage2D**, **CopyTexImage2D**, **CopyTexSubImage2D**, and **CopyTexSubImage3D**. If `SEPARABLE_2D` is enabled, and `CONVOLUTION_2D` is disabled, the separable two-dimensional convolution filter is instead applied these images.

The convolution operation is a sum of products of source image pixels and convolution filter pixels. Source image pixels always have four components: red,

Base Filter Format	R	G	B	A
ALPHA	R_s	G_s	B_s	$A_s * A_f$
LUMINANCE	$R_s * L_f$	$G_s * L_f$	$B_s * L_f$	A_s
LUMINANCE_ALPHA	$R_s * L_f$	$G_s * L_f$	$B_s * L_f$	$A_s * A_f$
INTENSITY	$R_s * I_f$	$G_s * I_f$	$B_s * I_f$	$A_s * I_f$
RGB	$R_s * R_f$	$G_s * G_f$	$B_s * B_f$	A_s
RGBA	$R_s * R_f$	$G_s * G_f$	$B_s * B_f$	$A_s * A_f$

Table 3.14: Computation of filtered color components depending on filter image format. $C * F$ indicates the convolution of image component C with filter F .

green, blue, and alpha, denoted in the equations below as R_s , G_s , B_s , and A_s . Filter pixels may be stored in one of five formats, with 1, 2, 3, or 4 components. These components are denoted as R_f , G_f , B_f , A_f , L_f , and I_f in the equations below. The result of the convolution operation is the 4-tuple R,G,B,A. Depending on the internal format of the filter, individual color components of each source image pixel are convolved with one filter component, or are passed unmodified. The rules for this are defined in table 3.14.

The convolution operation is defined differently for each of the three convolution filters. The variables W_f and H_f refer to the dimensions of the convolution filter. The variables W_s and H_s refer to the dimensions of the source pixel image.

The convolution equations are defined as follows, where C refers to the filtered result, C_f refers to the one- or two-dimensional convolution filter, and C_{row} and C_{column} refer to the two one-dimensional filters comprising the two-dimensional separable filter. C'_s depends on the source image color C_s and the convolution border mode as described below. C_r , the filtered output image, depends on all of these variables and is described separately for each border mode. The pixel indexing nomenclature is described in the **Convolution Filter Specification** subsection of section 3.7.3.

One-dimensional filter:

$$C[i'] = \sum_{n=0}^{W_f-1} C'_s[i' + n] * C_f[n]$$

Two-dimensional filter:

$$C[i', j'] = \sum_{n=0}^{W_f-1} \sum_{m=0}^{H_f-1} C'_s[i' + n, j' + m] * C_f[n, m]$$

Two-dimensional separable filter:

$$C[i', j'] = \sum_{n=0}^{W_f-1} \sum_{m=0}^{H_f-1} C'_s[i' + n, j' + m] * C_{row}[n] * C_{column}[m]$$

If W_f of a one-dimensional filter is zero, then $C[i]$ is always set to zero. Likewise, if either W_f or H_f of a two-dimensional filter is zero, then $C[i, j]$ is always set to zero.

The convolution border mode for a specific convolution filter is specified by calling

```
void ConvolutionParameter{if}( enum target, enum pname,
                               T param );
```

where *target* is the name of the filter, *pname* is `CONVOLUTION_BORDER_MODE`, and *param* is one of `REDUCE`, `CONSTANT_BORDER` or `REPLICATE_BORDER`.

Border Mode `REDUCE`

The width and height of source images convolved with border mode `REDUCE` are reduced by $W_f - 1$ and $H_f - 1$, respectively. If this reduction would generate a resulting image with zero or negative width and/or height, the output is simply null, with no error generated. The coordinates of the image that results from a convolution with border mode `REDUCE` are zero through $W_s - W_f$ in width, and zero through $H_s - H_f$ in height. In cases where errors can result from the specification of invalid image dimensions, it is these resulting dimensions that are tested, not the dimensions of the source image. (A specific example is **TexImage1D** and **TexImage2D**, which specify constraints for image dimensions. Even if **TexImage1D** or **TexImage2D** is called with a null pixel pointer, the dimensions of the resulting texture image are those that would result from the convolution of the specified image).

When the border mode is `REDUCE`, C'_s equals the source image color C_s and C_r equals the filtered result C .

For the remaining border modes, define $C_w = \lfloor W_f/2 \rfloor$ and $C_h = \lfloor H_f/2 \rfloor$. The coordinates (C_w, C_h) define the center of the convolution filter.

Border Mode `CONSTANT_BORDER`

If the convolution border mode is `CONSTANT_BORDER`, the output image has the same dimensions as the source image. The result of the convolution is the same as if the source image were surrounded by pixels with the same color as the

current convolution border color. Whenever the convolution filter extends beyond one of the edges of the source image, the constant-color border pixels are used as input to the filter. The current convolution border color is set by calling **ConvolutionParameterfv** or **ConvolutionParameteriv** with *pname* set to `CONVOLUTION_BORDER_COLOR` and *params* containing four values that comprise the RGBA color to be used as the image border. Integer color components are interpreted linearly such that the largest positive integer maps to 1.0, and the smallest negative integer maps to -1.0. Floating point color components are not clamped when they are specified.

For a one-dimensional filter, the result color is defined by

$$C_r[i] = C[i - C_w]$$

where $C[i']$ is computed using the following equation for $C'_s[i']$:

$$C'_s[i'] = \begin{cases} C_s[i'], & 0 \leq i' < W_s \\ C_c, & \text{otherwise} \end{cases}$$

and C_c is the convolution border color.

For a two-dimensional or two-dimensional separable filter, the result color is defined by

$$C_r[i, j] = C[i - C_w, j - C_h]$$

where $C[i', j']$ is computed using the following equation for $C'_s[i', j']$:

$$C'_s[i', j'] = \begin{cases} C_s[i', j'], & 0 \leq i' < W_s, 0 \leq j' < H_s \\ C_c, & \text{otherwise} \end{cases}$$

Border Mode `REPLICATE_BORDER`

The convolution border mode `REPLICATE_BORDER` also produces an output image with the same dimensions as the source image. The behavior of this mode is identical to that of the `CONSTANT_BORDER` mode except for the treatment of pixel locations where the convolution filter extends beyond the edge of the source image. For these locations, it is as if the outermost one-pixel border of the source image was replicated. Conceptually, each pixel in the leftmost one-pixel column of the source image is replicated C_w times to provide additional image data along the left edge, each pixel in the rightmost one-pixel column is replicated C_w times to provide additional image data along the right edge, and each pixel value in the top and bottom one-pixel rows is replicated to create C_h rows of image data along

the top and bottom edges. The pixel value at each corner is also replicated in order to provide data for the convolution operation at each corner of the source image.

For a one-dimensional filter, the result color is defined by

$$C_r[i] = C[i - C_w]$$

where $C[i']$ is computed using the following equation for $C'_s[i']$:

$$C'_s[i'] = C_s[\text{clamp}(i', W_s)]$$

and the clamping function $\text{clamp}(val, max)$ is defined as

$$\text{clamp}(val, max) = \begin{cases} 0, & val < 0 \\ val, & 0 \leq val < max \\ max - 1, & val \geq max \end{cases}$$

For a two-dimensional or two-dimensional separable filter, the result color is defined by

$$C_r[i, j] = C[i - C_w, j - C_h]$$

where $C[i', j']$ is computed using the following equation for $C'_s[i', j']$:

$$C'_s[i', j'] = C_s[\text{clamp}(i', W_s), \text{clamp}(j', H_s)]$$

If a convolution operation is performed, each component of the resulting image is scaled by the corresponding **PixelTransfer** parameters: `POST_CONVOLUTION_RED_SCALE` for an R component, `POST_CONVOLUTION_GREEN_SCALE` for a G component, `POST_CONVOLUTION_BLUE_SCALE` for a B component, and `POST_CONVOLUTION_ALPHA_SCALE` for an A component. The result is added to the corresponding bias: `POST_CONVOLUTION_RED_BIAS`, `POST_CONVOLUTION_GREEN_BIAS`, `POST_CONVOLUTION_BLUE_BIAS`, or `POST_CONVOLUTION_ALPHA_BIAS`.

The required state is three bits indicating whether each of one-dimensional, two-dimensional, or separable two-dimensional convolution is enabled or disabled, an integer describing the current convolution border mode, and four floating-point values specifying the convolution border color. In the initial state, all convolution operations are disabled, the border mode is `REDUCE`, and the border color is (0, 0, 0, 0).

Post Convolution Color Table Lookup

This step applies only to RGBA component groups. Post convolution color table lookup is enabled or disabled by calling **Enable** or **Disable** with the symbolic constant `POST_CONVOLUTION_COLOR_TABLE`. The post convolution table is defined by calling **ColorTable** with a *target* argument of `POST_CONVOLUTION_COLOR_TABLE`. In all other respects, operation is identical to color table lookup, as defined earlier in section 3.7.5.

The required state is one bit indicating whether post convolution table lookup is enabled or disabled. In the initial state, lookup is disabled.

Color Matrix Transformation

This step applies only to RGBA component groups. The components are transformed by the color matrix. Each transformed component is multiplied by an appropriate signed scale factor: `POST_COLOR_MATRIX_RED_SCALE` for an R component, `POST_COLOR_MATRIX_GREEN_SCALE` for a G component, `POST_COLOR_MATRIX_BLUE_SCALE` for a B component, and `POST_COLOR_MATRIX_ALPHA_SCALE` for an A component. The result is added to a signed bias: `POST_COLOR_MATRIX_RED_BIAS`, `POST_COLOR_MATRIX_GREEN_BIAS`, `POST_COLOR_MATRIX_BLUE_BIAS`, or `POST_COLOR_MATRIX_ALPHA_BIAS`. The resulting components replace each component of the original group.

That is, if M_c is the color matrix, a subscript of s represents the scale term for a component, and a subscript of b represents the bias term, then the components

$$\begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix}$$

are transformed to

$$\begin{pmatrix} R' \\ G' \\ B' \\ A' \end{pmatrix} = \begin{pmatrix} R_s & 0 & 0 & 0 \\ 0 & G_s & 0 & 0 \\ 0 & 0 & B_s & 0 \\ 0 & 0 & 0 & A_s \end{pmatrix} M_c \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix} + \begin{pmatrix} R_b \\ G_b \\ B_b \\ A_b \end{pmatrix}.$$

Post Color Matrix Color Table Lookup

This step applies only to RGBA component groups. Post color matrix color table lookup is enabled or disabled by calling **Enable** or **Disable**

with the symbolic constant `POST_COLOR_MATRIX_COLOR_TABLE`. The post color matrix table is defined by calling **ColorTable** with a *target* argument of `POST_COLOR_MATRIX_COLOR_TABLE`. In all other respects, operation is identical to color table lookup, as defined in section 3.7.5.

The required state is one bit indicating whether post color matrix lookup is enabled or disabled. In the initial state, lookup is disabled.

Histogram

This step applies only to RGBA component groups. Histogram operation is enabled or disabled by calling **Enable** or **Disable** with the symbolic constant `HISTOGRAM`.

If the width of the table is non-zero, then indices R_i , G_i , B_i , and A_i are derived from the red, green, blue, and alpha components of each pixel group (without modifying these components) by clamping each component to $[0, 1]$, multiplying by one less than the width of the histogram table, and rounding to the nearest integer. If the format of the `HISTOGRAM` table includes red or luminance, the red or luminance component of histogram entry R_i is incremented by one. If the format of the `HISTOGRAM` table includes green, the green component of histogram entry G_i is incremented by one. The blue and alpha components of histogram entries B_i and A_i are incremented in the same way. If a histogram entry component is incremented beyond its maximum value, its value becomes undefined; this is not an error.

If the **Histogram** *sink* parameter is `FALSE`, histogram operation has no effect on the stream of pixel groups being processed. Otherwise, all RGBA pixel groups are discarded immediately after the histogram operation is completed. Because histogram precedes minmax, no minmax operation is performed. No pixel fragments are generated, no change is made to texture memory contents, and no pixel values are returned. However, texture object state is modified whether or not pixel groups are discarded.

Minmax

This step applies only to RGBA component groups. Minmax operation is enabled or disabled by calling **Enable** or **Disable** with the symbolic constant `MINMAX`.

If the format of the minmax table includes red or luminance, the red component value replaces the red or luminance value in the minimum table element if and only if it is less than that component. Likewise, if the format includes red or luminance and the red component of the group is greater than the red or luminance value in the maximum element, the red group component replaces the red or lumi-

nance maximum component. If the format of the table includes green, the green group component conditionally replaces the green minimum and/or maximum if it is smaller or larger, respectively. The blue and alpha group components are similarly tested and replaced, if the table format includes blue and/or alpha. The internal type of the minimum and maximum component values is floating point, with at least the same representable range as a floating point number used to represent colors (section 2.1.1). There are no semantics defined for the treatment of group component values that are outside the representable range.

If the **Minmax sink** parameter is `FALSE`, minmax operation has no effect on the stream of pixel groups being processed. Otherwise, all RGBA pixel groups are discarded immediately after the minmax operation is completed. No pixel fragments are generated, no change is made to texture memory contents, and no pixel values are returned. However, texture object state is modified whether or not pixel groups are discarded.

3.7.6 Pixel Rectangle Multisample Rasterization

If `MULTISAMPLE` is enabled, and the value of `SAMPLE_BUFFERS` is one, then pixel rectangles are rasterized using the following algorithm. Let (X_{rp}, Y_{rp}) be the current raster position. (If the current raster position is invalid, then **DrawPixels** is ignored.) If a particular group (index or components) is the n th in a row and belongs to the m th row, consider the region in window coordinates bounded by the rectangle with corners

$$(X_{rp} + Z_x * n, Y_{rp} + Z_y * m)$$

and

$$(X_{rp} + Z_x * (n + 1), Y_{rp} + Z_y * (m + 1))$$

where Z_x and Z_y are the pixel zoom factors specified by **PixelZoom**, and may each be either positive or negative. A fragment representing group (n, m) is produced for each framebuffer pixel with one or more sample points that lie inside, or on the bottom or left boundary, of this rectangle. Each fragment so produced takes its associated data from the group and from the current raster position, in a manner consistent with the discussion in the **Conversion to Fragments** subsection of section 3.7.4. All depth and color sample values are assigned the same value, taken either from their group (for depth and color component groups) or from the current raster position (if they are not). All sample values are assigned the same fog coordinate and the same set of texture coordinates, taken from the current raster position.

A single pixel rectangle will generate multiple, perhaps very many fragments for the same framebuffer pixel, depending on the pixel zoom factors.

3.8 Bitmaps

Bitmaps are rectangles of zeros and ones specifying a particular pattern of fragments to be produced. Each of these fragments has the same associated data. These data are those associated with the *current raster position*.

Bitmaps are sent using

```
void Bitmap(sizei w, sizei h, float xbo, float ybo,
             float xbi, float ybi, ubyte *data );
```

w and *h* comprise the integer width and height of the rectangular bitmap, respectively. (*x_{bo}*, *y_{bo}*) gives the floating-point *x* and *y* values of the bitmap's origin. (*x_{bi}*, *y_{bi}*) gives the floating-point *x* and *y* increments that are added to the raster position after the bitmap is rasterized. *data* is a pointer to a bitmap.

Like a polygon pattern, a bitmap is unpacked from memory according to the procedure given in section 3.7.4 for **DrawPixels**; it is as if the *width* and *height* passed to that command were equal to *w* and *h*, respectively, the *type* were `BITMAP`, and the *format* were `COLOR_INDEX`. The unpacked values (before any conversion or arithmetic would have been performed) form a stipple pattern of zeros and ones. See figure 3.9.

A bitmap sent using **Bitmap** is rasterized as follows. First, if the current raster position is invalid (the valid bit is reset), the bitmap is ignored. Otherwise, a rectangular array of fragments is constructed, with lower left corner at

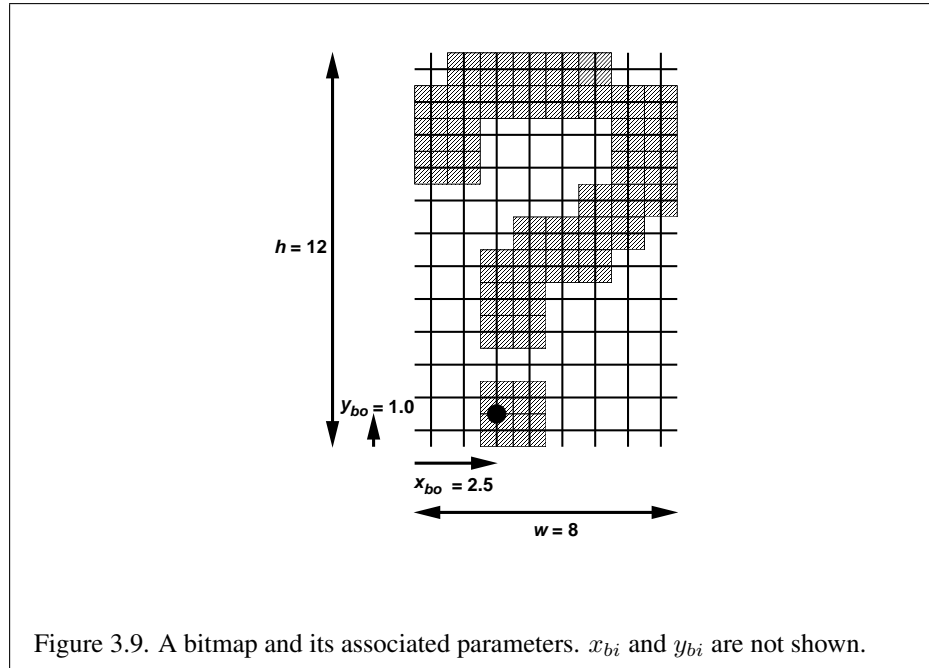
$$(x_{ll}, y_{ll}) = (\lfloor x_{rp} - x_{bo} \rfloor, \lfloor y_{rp} - y_{bo} \rfloor)$$

and upper right corner at $(x_{ll} + w, y_{ll} + h)$ where *w* and *h* are the width and height of the bitmap, respectively. Fragments in the array are produced if the corresponding bit in the bitmap is 1 and not produced otherwise. The associated data for each fragment are those associated with the current raster position. Once the fragments have been produced, the current raster position is updated:

$$(x_{rp}, y_{rp}) \leftarrow (x_{rp} + x_{bi}, y_{rp} + y_{bi}).$$

The *z* and *w* values of the current raster position remain unchanged.

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



Bitmap Multisample Rasterization

If `MULTISAMPLE` is enabled, and the value of `SAMPLE_BUFFERS` is one, then bitmaps are rasterized using the following algorithm. If the current raster position is invalid, the bitmap is ignored. Otherwise, a screen-aligned array of pixel-size rectangles is constructed, with its lower left corner at (X_{rp}, Y_{rp}) , and its upper right corner at $(X_{rp} + w, Y_{rp} + h)$, where w and h are the width and height of the bitmap. Rectangles in this array are eliminated if the corresponding bit in the bitmap is 0, and are retained otherwise. Bitmap rasterization produces a fragment for each framebuffer pixel with one or more sample points either inside or on the bottom or left edge of a retained rectangle.

Coverage bits that correspond to sample points either inside or on the bottom or left edge of a retained rectangle are 1, other coverage bits are 0. The associated data for each sample are those associated with the current raster position. Once the fragments have been produced, the current raster position is updated exactly as it is in the single-sample rasterization case.

3.9 Texturing

Texturing maps a portion of one or more specified images onto each primitive for which texturing is enabled. This mapping is accomplished by using the color of an image at the location indicated by a texture coordinate set's (s, t, r, q) coordinates.

The internal data type of a texture may be fixed-point, floating-point, signed integer or unsigned integer, depending on the internal format of the texture. The correspondence between the internal format and the internal data type is given in tables 3.16-3.18. Fixed-point and floating-point textures return a floating-point value and integer textures return signed or unsigned integer values. When a fragment shader is active, the shader is responsible for interpreting the result of a texture lookup as the correct data type, otherwise the result is undefined. When not using a fragment shader, floating-point texture values are assumed, and the results of using integer textures in this case are undefined.

Six types of texture are supported; each is a collection of images built from one-, two-, or three-dimensional array of image elements referred to as texels. One-, two-, and three-dimensional textures consist respectively of one-, two-, or three-dimensional texel arrays. One- and two-dimensional array textures are arrays of one- or two-dimensional images, consisting of one or more layers. Finally, a cube map is a special two-dimensional array texture with six layers that represent the faces of a cube. When accessing a cube map, the texture coordinates are projected onto one of the six faces of the cube.

Implementations must support texturing using at least two images at a time. Each fragment or vertex carries multiple sets of texture coordinates (s, t, r, q) which are used to index separate images to produce color values which are collectively used to modify the resulting transformed vertex or fragment color. Texturing is specified only for RGBA mode; its use in color index mode is undefined. The following subsections (up to and including section 3.9.7) specify the GL operation with a single texture and section 3.9.17 specifies the details of how multiple texture units interact.

The GL provides two ways to specify the details of how texturing of a primitive is effected. The first is referred to as *fixed-function fragment shading*, or simply *fixed-function*, and is described in this section. The second is referred to as a fragment shader, and is described in section 3.12. The specification of the image to be texture mapped and the means by which the image is filtered when applied to the primitive are common to both methods and are discussed in this section. The fixed-function method for determining what RGBA value is produced is also described in this section. If a fragment shader is active, the method for determining the RGBA value is specified by an application-supplied fragment shader as described in the OpenGL Shading Language Specification.

When no fragment shader is active, the coordinates used for texturing are $(s/q, t/q, r/q)$, derived from the original texture coordinates (s, t, r, q) . If the q texture coordinate is less than or equal to zero, the coordinates used for texturing are undefined. When a fragment shader is active, the (s, t, r, q) coordinates are available to the fragment shader. The coordinates used for texturing in a fragment shader are defined by the OpenGL Shading Language Specification.

3.9.1 Texture Image Specification

The command

```
void TexImage3D( enum target, int level, int internalformat,  
                 sizei width, sizei height, sizei depth, int border,  
                 enum format, enum type, void *data );
```

is used to specify a three-dimensional texture image. *target* must be one of `TEXTURE_3D` for a three-dimensional texture or `TEXTURE_2D_ARRAY` for an two-dimensional array texture. Additionally, *target* may be either `PROXY_TEXTURE_3D` for a three-dimensional proxy texture, or `PROXY_TEXTURE_2D_ARRAY` for a two-dimensional proxy array texture, as discussed in section 3.9.11. *format*, *type*, and *data* match the corresponding arguments to **DrawPixels** (refer to section 3.7.4); they specify the format of the image data, the type of those data, and a reference to the image data in the currently bound pixel unpack buffer or client memory. The *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 in the same manner as **DrawPixels**. 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 in the same manner as **DrawPixels**.

The selected groups are processed exactly as for **DrawPixels**, stopping just before final conversion. If the *internalformat* of the texture is signed or unsigned

integer, the components are clamped to the representable range of the internal format. For signed formats, this is $[-2^{n-1}, 2^{n-1} - 1]$ where n is the number of bits per component; for unsigned formats, the range is $[0, 2^n - 1]$. For color component groups, if the *internalformat* of the texture is fixed-point, the R, G, B, and A values are clamped to $[0, 1]$. For depth component groups, the depth value is clamped to $[0, 1]$. Otherwise, values are not modified. Stencil index values are masked by $2^n - 1$, where n is the number of stencil bits in the internal format resolution (see below). If the base internal format is `DEPTH_STENCIL` and *format* is not `DEPTH_STENCIL`, then the values of the stencil index texture components are undefined.

Components are then selected from the resulting R, G, B, A, depth, or stencil values to obtain a texture with the *base internal format* specified by (or derived from) *internalformat*. Table 3.15 summarizes the mapping of R, G, B, A, depth, or stencil values to texture components, as a function of the base internal format of the texture image. *internalformat* may be specified as one of the internal format symbolic constants listed in table 3.15, as one of the *sized internal format* symbolic constants listed in tables 3.16- 3.18, as one of the generic compressed internal format symbolic constants listed in table 3.19, or as one of the specific compressed internal format symbolic constants (if listed in table 3.19). *internalformat* may (for backwards compatibility with the 1.0 version of the GL) also take on the integer values 1, 2, 3, and 4, which are equivalent to symbolic constants `LUMINANCE`, `LUMINANCE_ALPHA`, `RGB`, and `RGBA` respectively. Specifying a value for *internalformat* that is not one of the above values generates the error `INVALID_VALUE`.

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

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

Textures with integer internal formats tables 3.16- 3.17 require integer data. The error `INVALID_OPERATION` is generated if the internal format is integer and *format* is not one of the integer formats listed in table 3.6; if the internal format is not integer and *format* is an integer format; or if *format* is an integer format and

Base Internal Format	RGBA, Depth, and Stencil Values	Internal Components
ALPHA	A	<i>A</i>
DEPTH_COMPONENT	Depth	<i>D</i>
DEPTH_STENCIL	Depth, Stencil	<i>D, S</i>
LUMINANCE	R	<i>L</i>
LUMINANCE_ALPHA	R, A	<i>L, A</i>
INTENSITY	R	<i>I</i>
RED	R	<i>R</i>
RG	R, G	<i>R, G</i>
RGB	R, G, B	<i>R, G, B</i>
RGBA	R, G, B, A	<i>R, G, B, A</i>

Table 3.15: Conversion from RGBA, depth, and stencil pixel components to internal texture, table, or filter components. See section 3.9.13 for a description of the texture components *R*, *G*, *B*, *A*, *L*, *I*, *D*, and *S*.

type is `GLfloat`.

The GL provides no specific compressed internal formats but does provide a mechanism to obtain token values for such formats provided by extensions. The number of specific compressed internal formats supported by the renderer can be obtained by querying the value of `NUM_COMPRESSED_TEXTURE_FORMATS`. The set of specific compressed internal formats supported by the renderer can be obtained by querying the value of `COMPRESSED_TEXTURE_FORMATS`. The only values returned by this query are those corresponding to formats suitable for general-purpose usage. The renderer will not enumerate formats with restrictions that need to be specifically understood prior to use.

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

The *internal component resolution* is the number of bits allocated to each value in a texture image. If *internalformat* is specified as a base internal format, the GL stores the resulting texture with internal component resolutions of its own choosing. If a sized internal format is specified, the mapping of the R, G, B, A, depth, and stencil values to texture components is equivalent to the mapping of the corresponding base internal format's components, as specified in table 3.15; the type (unsigned int, float, etc.) is assigned the same type specified by *internalformat*; and the memory allocation per texture component is assigned by the GL to match the allocations listed in tables 3.16- 3.18 as closely as possible. (The definition of closely is left up to the implementation. However, a non-zero number of bits must be allocated for each component whose *desired* allocation in tables 3.16- 3.18 is non-zero, and zero bits must be allocated for all other components).

Required Texture Formats

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

In addition, implementations are required to support the following sized internal formats. Requesting one of these internal formats for any texture type will allocate exactly the internal component sizes and types shown for that format in tables 3.16- 3.17:

- Color formats:
 - RGBA32F, RGBA32I, RGBA32UI, RGBA16, RGBA16F, RGBA16I, RGBA16UI, RGBA8, RGBA8I, RGBA8UI, SRGB8_ALPHA8, and RGB10_A2.
 - R11F_G11F_B10F.
 - RG32F, RG32I, RG32UI, RG16, RG16F, RG16I, RG16UI, RG8, RG8I, and RG8UI.
 - R32F, R32I, R32UI, R16F, R16I, R16UI, R16 R8, R8I, and R8UI.
 - ALPHA8.
- Color formats (texture-only):
 - RGB32F, RGB32I, and RGB32UI.
 - RGB16F, RGB16I, RGB16UI, and RGB16.
 - RGB8, RGB8I, RGB8UI, and SRGB8.

- RGB9_E5.
- COMPRESSED_RG_RGTC2 and COMPRESSED_SIGNED_RG_RGTC2.
- COMPRESSED_RED_RGTC1 and COMPRESSED_SIGNED_RED_RGTC1.
- Depth formats: DEPTH_COMPONENT32F, DEPTH_COMPONENT24, and DEPTH_COMPONENT16,
- Combined depth+stencil formats: DEPTH32F_STENCIL8 and DEPTH24_STENCIL8.

Encoding of Special Internal Formats

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

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

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

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

where

$$sharedexp_{max} = \frac{(2^N - 1)}{2^N} 2^{E_{max} - B}.$$

N is the number of mantissa bits per component (9), B is the exponent bias (15), and E_{max} is the maximum allowed biased exponent value (31).

The largest clamped component, max_c , is determined:

$$max_c = \max(red_c, green_c, blue_c)$$

A preliminary shared exponent exp_p is computed:

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

A refined shared exponent exp_s is computed:

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

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

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

$$\begin{aligned} red_s &= \left\lfloor \frac{red_c}{2^{exp_s - B - N}} + 0.5 \right\rfloor \\ green_s &= \left\lfloor \frac{green_c}{2^{exp_s - B - N}} + 0.5 \right\rfloor \\ blue_s &= \left\lfloor \frac{blue_c}{2^{exp_s - B - N}} + 0.5 \right\rfloor \end{aligned}$$

The resulting red_s , $green_s$, $blue_s$, and exp_s are stored in the red, green, blue, and shared bits respectively of the texture image.

An implementation accepting pixel data of *type* UNSIGNED_INT_5_9_9_9_REV with *format* RGB is allowed to store the components “as is” if the implementation can determine the current pixel transfer state acts as an identity transform on the components.

Sized Internal Format	Base Internal Format	<i>R</i> bits	<i>G</i> bits	<i>B</i> bits	<i>A</i> bits	Shared bits
ALPHA4	ALPHA				4	
ALPHA8	ALPHA				8	
ALPHA12	ALPHA				12	
ALPHA16	ALPHA				16	
R8	RED	8				
R16	RED	16				
RG8	RG	8	8			
RG16	RG	16	16			
R3_G3_B2	RGB	3	3	2		
RGB4	RGB	4	4	4		
RGB5	RGB	5	5	5		
RGB8	RGB	8	8	8		
RGB10	RGB	10	10	10		
RGB12	RGB	12	12	12		
RGB16	RGB	16	16	16		
Sized internal color formats continued on next page						

Sized internal color formats continued from previous page						
Sized Internal Format	Base Internal Format	<i>R</i> bits	<i>G</i> bits	<i>B</i> bits	<i>A</i> bits	Shared bits
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	
RGB10_A2	RGBA	10	10	10	2	
RGBA12	RGBA	12	12	12	12	
RGBA16	RGBA	16	16	16	16	
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		
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		
Sized internal color formats continued on next page						

Sized internal color formats continued from previous page						
Sized Internal Format	Base Internal Format	<i>R</i> bits	<i>G</i> bits	<i>B</i> bits	<i>A</i> bits	Shared bits
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 3.16: Correspondence of sized internal color formats to base internal formats, internal data type, and *desired* component resolutions for each sized internal format. The component resolution prefix indicates the internal data type: *f* is floating point, *i* is signed integer, *ui* is unsigned integer, and no prefix is fixed-point.

Sized Internal Format	Base Internal Format	<i>A</i> bits	<i>L</i> bits	<i>I</i> bits
LUMINANCE4	LUMINANCE		4	
LUMINANCE8	LUMINANCE		8	
LUMINANCE12	LUMINANCE		12	
LUMINANCE16	LUMINANCE		16	
LUMINANCE4_ALPHA4	LUMINANCE_ALPHA	4	4	
LUMINANCE6_ALPHA2	LUMINANCE_ALPHA	2	6	
LUMINANCE8_ALPHA8	LUMINANCE_ALPHA	8	8	
LUMINANCE12_ALPHA4	LUMINANCE_ALPHA	4	12	
LUMINANCE12_ALPHA12	LUMINANCE_ALPHA	12	12	
LUMINANCE16_ALPHA16	LUMINANCE_ALPHA	16	16	
INTENSITY4	INTENSITY			4
INTENSITY8	INTENSITY			8
INTENSITY12	INTENSITY			12
INTENSITY16	INTENSITY			16
Sized internal luminance formats continued on next page				

Sized Internal Format	Base Internal Format	<i>D</i> bits	<i>S</i> 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 3.18: Correspondence of sized internal depth and stencil formats to base internal formats, internal data type, and *desired* component resolutions for each sized internal format. The component resolution prefix indicates the internal data type: *f* is floating point, *i* is signed integer, *ui* is unsigned integer, and no prefix is fixed-point.

Sized internal luminance formats continued from previous page				
Sized Internal Format	Base Internal Format	<i>A</i> bits	<i>L</i> bits	<i>I</i> bits
SLUMINANCE	LUMINANCE		8	
SLUMINANCE_ALPHA8	LUMINANCE_ALPHA	8	8	

Table 3.17: Correspondence of sized internal luminance and intensity formats to base internal formats, internal data type, and *desired* component resolutions for each sized internal format. The component resolution prefix indicates the internal data type: *f* is floating point, *i* is signed integer, *ui* is unsigned integer, and no prefix is fixed-point.

If a compressed internal format is specified, the mapping of the R, G, B, and A values to texture components is equivalent to the mapping of the corresponding base internal format's components, as specified in table 3.15. The specified image is compressed using a (possibly lossy) compression algorithm chosen by the GL.

A GL implementation may vary its allocation of internal component resolution or compressed internal format based on any **TexImage3D**, **TexImage2D** (see below), or **TexImage1D** (see below) parameter (except *target*), but the allocation and chosen compressed image format must not be a function of any other state and cannot be changed once they are established. In addition, the choice of a compressed

Compressed Internal Format	Base Internal Format	Type
COMPRESSED_ALPHA	ALPHA	Generic
COMPRESSED_LUMINANCE	LUMINANCE	Generic
COMPRESSED_LUMINANCE_ALPHA	LUMINANCE_ALPHA	Generic
COMPRESSED_INTENSITY	INTENSITY	Generic
COMPRESSED_RED	RED	Generic
COMPRESSED_RG	RG	Generic
COMPRESSED_RGB	RGB	Generic
COMPRESSED_RGBA	RGBA	Generic
COMPRESSED_SRGB	RGB	Generic
COMPRESSED_SRGB_ALPHA	RGBA	Generic
COMPRESSED_SLUMINANCE	LUMINANCE	Generic
COMPRESSED_SLUMINANCE_ALPHA	LUMINANCE_ALPHA	Generic
COMPRESSED_RED_RGTC1	RED	Specific
COMPRESSED_SIGNED_RED_RGTC1	RED	Specific
COMPRESSED_RG_RGTC2	RG	Specific
COMPRESSED_SIGNED_RG_RGTC2	RG	Specific

Table 3.19: Generic and specific compressed internal formats. The specific *RGTC* formats are described in appendix C.1.

image format may not be affected by the *data* parameter. Allocations must be invariant; the same allocation and compressed image format must be chosen each time a texture image is specified with the same parameter values. These allocation rules also apply to proxy textures, which are described in section 3.9.11.

The image itself (referred to by *data*) is a sequence of groups of values. The first group is the lower left back corner of the texture image. Subsequent groups fill out rows of width *width* from left to right; *height* rows are stacked from bottom to top forming a single two-dimensional image slice; and *depth* slices are stacked from back to front. When the final R, G, B, and A components have been computed for a group, they are assigned to components of a *texel* as described by table 3.15. Counting from zero, each resulting *N*th texel is assigned internal integer coordinates (i, j, k) , where

$$\begin{aligned} i &= (N \bmod \text{width}) - w_b \\ j &= (\lfloor \frac{N}{\text{width}} \rfloor \bmod \text{height}) - h_b \\ k &= (\lfloor \frac{N}{\text{width} \times \text{height}} \rfloor \bmod \text{depth}) - d_b \end{aligned}$$

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

Each color component is converted (by rounding to nearest) to a fixed-point value with n bits, where n is the number of bits of storage allocated to that component in the image array. We assume that the fixed-point representation used represents each value $k/(2^n - 1)$, where $k \in \{0, 1, \dots, 2^n - 1\}$, as k (e.g. 1.0 is represented in binary as a string of all ones).

The *level* argument to **TexImage3D** is an integer *level-of-detail* number. Levels of detail are discussed below, under **Mipmapping**. The main texture image has a level of detail number of 0. If a level-of-detail less than zero is specified, the error INVALID_VALUE is generated.

The *border* argument to **TexImage3D** is a border width. The significance of borders is described below. The border width affects the dimensions of the texture image: let

$$\begin{aligned} w_s &= w_t + 2w_b \\ h_s &= h_t + 2h_b \\ d_s &= d_t + 2d_b \end{aligned} \tag{3.15}$$

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

An image with zero width, height, or depth indicates the null texture. If the null texture is specified for the level-of-detail specified by texture parameter `TEXTURE_BASE_LEVEL` (see section 3.9.4), it is as if texturing were disabled.

Currently, the maximum border width b_t is 1. If *border* is less than zero, or greater than b_t , then the error `INVALID_VALUE` is generated.

The maximum allowable width, height, or depth of a texel array for a three-dimensional texture is an implementation dependent function of the level-of-detail and internal format of the resulting image array. It must be at least $2^{k-lod} + 2b_t$ for image arrays of level-of-detail 0 through k , where k is the log base 2 of `MAX_3D_TEXTURE_SIZE`, lod is the level-of-detail of the image array, and b_t is the maximum border width. It may be zero for image arrays of any level-of-detail greater than k . The error `INVALID_VALUE` is generated if the specified image is too large to be stored under any conditions.

If a pixel unpack buffer object is bound and storing texture data would access memory beyond the end of the pixel unpack buffer, an `INVALID_OPERATION` error results.

In a similar fashion, the maximum allowable width of a texel array for a one- or two-dimensional, 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.

An implementation may allow an image array of level 0 to be created only if that single image array can be supported. Additional constraints on the creation of image arrays of level 1 or greater are described in more detail in section 3.9.10.

The command

```
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, 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, or PROXY_TEXTURE_CUBE_MAP for a cube map proxy texture in the special case discussed in section 3.9.11. The other parameters match the corresponding parameters of **TexImage3D**.

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

- The border depth, d_b , is zero, and the *depth* of the image is always 1 regardless of the value of *border*.
- The border height, h_b , is zero if *target* is TEXTURE_1D_ARRAY, and *border* otherwise.
- Convolution will be performed on the image (possibly changing its *width* and *height*) if SEPARABLE_2D or CONVOLUTION_2D is enabled.
- UNPACK_SKIP_IMAGES is ignored.

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

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

Finally, the command

```
void TexImage1D( enum target, int level,
                 int internalformat, sizei width, int border,
                 enum format, enum type, void *data );
```

is used to specify a one-dimensional texture image. *target* must be either `TEXTURE_1D`, or `PROXY_TEXTURE_1D` in the special case discussed in section 3.9.11.)

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

- The border height and depth (h_b and d_b) are always zero, regardless of the value of *border*.
- Convolution will be performed on the image (possibly changing its *width*) only if `CONVOLUTION_1D` is enabled.

The image indicated to the GL by the image pointer is decoded and copied into the GL's internal memory. This copying effectively places the decoded image inside a border of the maximum allowable width b_t whether or not a border has been specified (see figure 3.10)¹. If no border or a border smaller than the maximum allowable width has been specified, then the image is still stored as if it were surrounded by a border of the maximum possible width. Any excess border (which surrounds the specified image, including any border) is assigned unspecified values. A two-dimensional texture has a border only at its left, right, top, and bottom ends, and a one-dimensional texture has a border only at its left and right ends.

We shall refer to the (possibly border augmented) decoded image as the *texel array*. A three-dimensional texel array has width, height, and depth w_s , h_s , and d_s as defined in equation 3.15. A two-dimensional texel array has depth $d_s = 1$, with height h_s and width w_s as above, and a one-dimensional texel array has depth $d_s = 1$, height $h_s = 1$, and width w_s as above.

An element (i, j, k) of the texel array is called a *texel* (for a two-dimensional texture or one-dimensional array texture, k is irrelevant; for a one-dimensional texture, j and k are both irrelevant). The *texture value* used in texturing a fragment is determined by that fragment's associated (s, t, r) coordinates, but may not correspond to any actual texel. See figure 3.10.

If the *data* argument of **TexImage1D**, **TexImage2D**, or **TexImage3D** is a null pointer (a zero-valued pointer in the C implementation), and the pixel unpack buffer object is zero, a one-, two-, or three-dimensional texel array is created with the specified *target*, *level*, *internalformat*, *border*, *width*, *height*, and *depth*, but with unspecified image contents. In this case no pixel values are accessed in client memory, and no pixel processing is performed. Errors are generated, however, exactly as though the *data* pointer were valid. Otherwise if the pixel unpack buffer object is non-zero, the *data* argument is treated normally to refer to the beginning of the pixel unpack buffer object's data.

¹ Figure 3.10 needs to show a three-dimensional texture image.

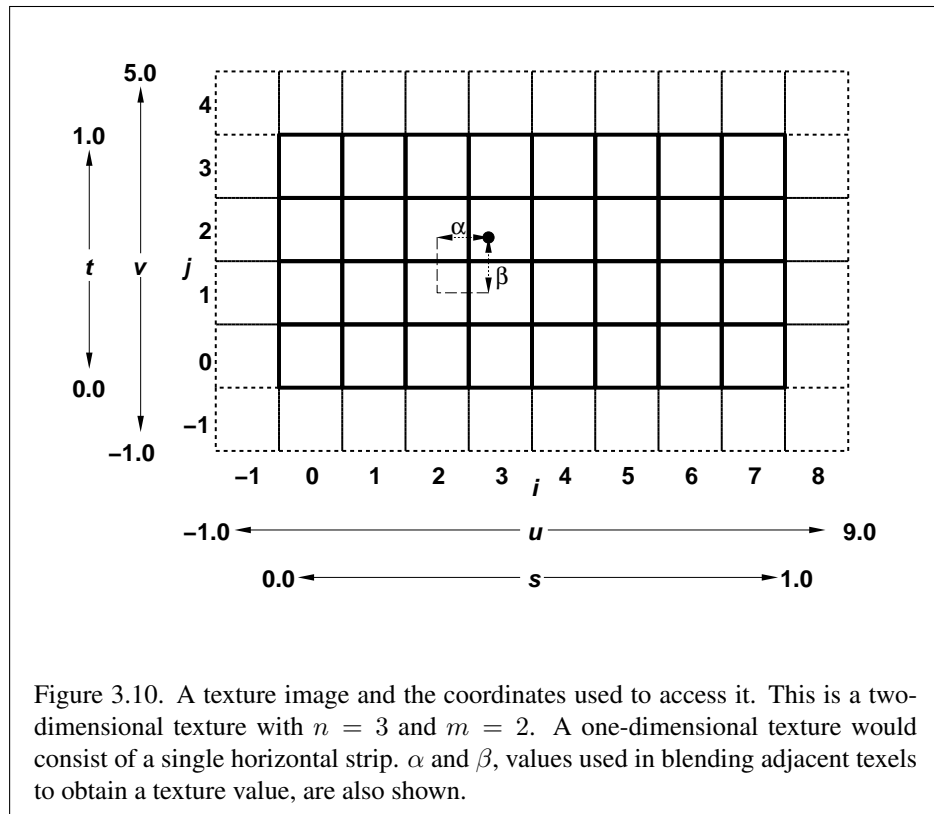


Figure 3.10. A texture image and the coordinates used to access it. This is a two-dimensional texture with $n = 3$ and $m = 2$. A one-dimensional texture would consist of a single horizontal strip. α and β , values used in blending adjacent texels to obtain a texture value, are also shown.

3.9.2 Alternate Texture Image Specification Commands

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

The command

```
void CopyTexImage2D(enum target, int level,
                    enum internalformat, int x, int y, sizei width,
                    sizei height, int border);
```

defines a two-dimensional texel array in exactly the manner of **TexImage2D**, except that the image data are taken from the framebuffer rather than from client memory. Currently, *target* must be one of `TEXTURE_2D`, `TEXTURE_1D_ARRAY`, `TEXTURE_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, or `TEXTURE_CUBE_MAP_NEGATIVE_Z`. *x*, *y*, *width*, and *height* correspond precisely to the corresponding arguments to **CopyPixels** (refer to section 4.3.3); they specify the image's *width* and *height*, and the lower left (*x*, *y*) coordinates of the framebuffer region to be copied. The image is taken from the framebuffer exactly as if these arguments were passed to **CopyPixels** with argument *type* set to `COLOR`, `DEPTH`, or `DEPTH_STENCIL`, depending on *internalformat*, stopping after pixel transfer processing is complete. RGBA data is taken from the current color buffer, while depth component and stencil index data are taken from the depth and stencil buffers, respectively. The error `INVALID_OPERATION` is generated if depth component data is required and no depth buffer is present; if stencil index data is required and no stencil buffer is present; if integer RGBA data is required and the format of the current color buffer is not integer; or if floating- or fixed-point RGBA data is required and the format of the current color buffer is integer.

Subsequent processing is identical to that described for **TexImage2D**, beginning with clamping of the R, G, B, A, or depth values, and masking of the stencil index values from the resulting pixel groups. Parameters *level*, *internalformat*, and *border* are specified using the same values, with the same meanings, as the equivalent arguments of **TexImage2D**, except that *internalformat* may not be specified as 1, 2, 3, or 4. An invalid value specified for *internalformat* generates the error `INVALID_ENUM`. The constraints on *width*, *height*, and *border* are exactly those for the equivalent arguments of **TexImage2D**.

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

The command

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

defines a one-dimensional texel array in exactly the manner of **TexImage1D**, except that the image data are taken from the framebuffer, rather than from client memory. Currently, *target* must be `TEXTURE_1D`. For the purposes of decoding the texture image, **CopyTexImage1D** is equivalent to calling **CopyTexImage2D** with corresponding arguments and *height* of 1, except that the *height* of the image is always 1, regardless of the value of *border*. *level*, *internalformat*, and *border* are specified using the same values, with the same meanings, as the equivalent arguments of **TexImage1D**, except that *internalformat* may not be specified as 1, 2, 3, or 4. The constraints on *width* and *border* are exactly those of the equivalent arguments of **TexImage1D**.

Six additional commands,

```
void TexSubImage3D( enum target, int level, int xoffset,
                   int yoffset, int zoffset, sizei width, sizei height,
                   sizei depth, enum format, enum type, void *data );
void TexSubImage2D( enum target, int level, int xoffset,
                   int yoffset, sizei width, sizei height, enum format,
                   enum type, void *data );
void TexSubImage1D( enum target, int level, int xoffset,
                   sizei width, enum format, enum type, void *data );
void CopyTexSubImage3D( enum target, int level,
                       int xoffset, int yoffset, int zoffset, int x, int y,
                       sizei width, sizei height );
void CopyTexSubImage2D( enum target, int level,
                       int xoffset, int yoffset, int x, int y, sizei width,
                       sizei height );
void CopyTexSubImage1D( enum target, int level,
                       int xoffset, int x, int y, sizei width );
```

respecify only a rectangular subregion of an existing texel array. No change is made to the *internalformat*, *width*, *height*, *depth*, or *border* parameters of the specified texel array, nor is any change made to texel values outside the specified subregion. Currently the *target* arguments of **TexSubImage1D** and **CopyTexSubImage1D** must be `TEXTURE_1D`, the *target* arguments of **TexSubImage2D**

and **CopyTexSubImage2D** must be one of `TEXTURE_2D`, `TEXTURE_1D_ARRAY`, `TEXTURE_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.

TexSubImage3D arguments *width*, *height*, *depth*, *format*, *type*, and *data* match the corresponding arguments to **TexImage3D**, meaning that they are specified using the same values, and have the same meanings. Likewise, **TexSubImage2D** arguments *width*, *height*, *format*, *type*, and *data* match the corresponding arguments to **TexImage2D**, and **TexSubImage1D** arguments *width*, *format*, *type*, and *data* match the corresponding arguments to **TexImage1D**.

CopyTexSubImage3D and **CopyTexSubImage2D** arguments *x*, *y*, *width*, and *height* match the corresponding arguments to **CopyTexImage2D**². **CopyTexSubImage1D** arguments *x*, *y*, and *width* match the corresponding arguments to **CopyTexImage1D**. Each of the **TexSubImage** commands interprets and processes pixel groups in exactly the manner of its **TexImage** counterpart, except that the assignment of R, G, B, A, depth, and stencil index pixel group values to the texture components is controlled by the *internalformat* of the texel array, not by an argument to the command. The same constraints and errors apply to the **TexSubImage** commands' argument *format* and the *internalformat* of the texel array being re-specified as apply to the *format* and *internalformat* arguments of its **TexImage** counterparts.

Arguments *xoffset*, *yoffset*, and *zoffset* of **TexSubImage3D** and **CopyTexSubImage3D** specify the lower left texel coordinates of a *width*-wide by *height*-high by *depth*-deep rectangular subregion of the texel array. The *depth* argument associated with **CopyTexSubImage3D** is always 1, because framebuffer memory is two-dimensional - only a portion of a single *s, t* slice of a three-dimensional texture is replaced by **CopyTexSubImage3D**.

Negative values of *xoffset*, *yoffset*, and *zoffset* correspond to the coordinates of border texels, addressed as in figure 3.10. Taking w_s , h_s , d_s , w_b , h_b , and d_b to be the specified width, height, depth, and border width, border height, and border depth of the texel array, and taking x , y , z , w , h , and d to be the *xoffset*, *yoffset*, *zoffset*, *width*, *height*, and *depth* argument values, any of the following relationships

² Because the framebuffer is inherently two-dimensional, there is no **CopyTexImage3D** command.

generates the error `INVALID_VALUE`:

$$\begin{aligned}
 x &< -w_b \\
 x + w &> w_s - w_b \\
 y &< -h_b \\
 y + h &> h_s - h_b \\
 z &< -d_b \\
 z + d &> d_s - d_b
 \end{aligned}$$

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

$$\begin{aligned}
 i &= x + (n \bmod w) \\
 j &= y + (\lfloor \frac{n}{w} \rfloor \bmod h) \\
 k &= z + (\lfloor \frac{n}{width * height} \rfloor \bmod d)
 \end{aligned}$$

Arguments *xoffset* and *yoffset* of **TexSubImage2D** and **CopyTexSubImage2D** specify the lower left texel coordinates of a *width*-wide by *height*-high rectangular subregion of the texel array. Negative values of *xoffset* and *yoffset* correspond to the coordinates of border texels, addressed as in figure 3.10. Taking w_s , h_s , and b_s to be the specified width, height, and border width of the texel array, and taking x , y , w , and h to be the *xoffset*, *yoffset*, *width*, and *height* argument values, any of the following relationships generates the error `INVALID_VALUE`:

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

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

$$\begin{aligned}
 i &= x + (n \bmod w) \\
 j &= y + (\lfloor \frac{n}{w} \rfloor \bmod h)
 \end{aligned}$$

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 n th 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 3.19, 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 or equal to `TEXTURE_WIDTH`, unless *xoffset* and *yoffset* are both zero.
- *height* is not a multiple of four or equal to `TEXTURE_HEIGHT`, unless *xoffset* and *yoffset* are both 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).

3.9.3 Compressed Texture Images

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

The commands

```
void CompressedTexImage1D( enum target, int level,
                           enum internalformat, sizei width, int border,
                           sizei imageSize, void *data );
void CompressedTexImage2D( enum target, int level,
                           enum internalformat, sizei width, sizei height,
                           int border, sizei imageSize, void *data );
void CompressedTexImage3D( enum target, int level,
                           enum internalformat, sizei width, sizei height,
                           sizei depth, int border, sizei imageSize, void *data );
```

define one-, two-, and three-dimensional texture images, respectively, with incoming data stored in a specific compressed image format. The *target*, *level*, *internalformat*, *width*, *height*, *depth*, and *border* parameters have the same meaning as in **TexImage1D**, **TexImage2D**, and **TexImage3D**. *data* refers to compressed image data stored in the specific compressed image format corresponding to *internalformat*. If a pixel unpack buffer is bound (as indicated by a non-zero value of `PIXEL_UNPACK_BUFFER_BINDING`), *data* is an offset into the pixel unpack buffer and the compressed data is read from the buffer relative to this offset; otherwise, *data* is a pointer to client memory and the compressed data is read from client memory relative to the pointer.

internalformat must be a supported specific compressed internal format. An `INVALID_ENUM` error will be generated if any other values, including any of the six generic compressed internal formats, is specified.

For all other compressed internal formats, the compressed image will be decoded according to the specification defining the *internalformat* token. Compressed texture images are treated as an array of *imageSize* ubytes relative to *data*. If a pixel unpack buffer object is bound and *data* + *imageSize* is greater than the size of the pixel buffer, an `INVALID_OPERATION` error results. All pixel storage and pixel transfer modes are ignored when decoding a compressed texture image. If the *imageSize* parameter is not consistent with the format, dimensions, and contents of the compressed image, an `INVALID_VALUE` error results. If the compressed image is not encoded according to the defined image format, the results of the call are undefined.

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

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

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

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

If *internalformat* is one of the specific RGTC or formats described in table 3.19, the compressed image data is stored using one of the RGTC compressed texture image encodings (see appendix C.1) The RGTC texture compression algorithm supports only two-dimensional images without borders. If *internalformat* is an RGTC

format, **CompressedTexImage1D** will generate an `INVALID_ENUM` error; **CompressedTexImage2D** will generate an `INVALID_OPERATION` error if *border* is non-zero; and **CompressedTexImage3D** will generate an `INVALID_OPERATION` error if *border* is non-zero or *target* is not `TEXTURE_2D_ARRAY`.

The commands

```
void CompressedTexSubImage1D( enum target, int level,
    int xoffset, sizei width, enum format, sizei imageSize,
    void *data );
void CompressedTexSubImage2D( enum target, int level,
    int xoffset, int yoffset, sizei width, sizei height,
    enum format, sizei imageSize, void *data );
void CompressedTexSubImage3D( enum target, int level,
    int xoffset, int yoffset, int zoffset, sizei width,
    sizei height, sizei depth, enum format,
    sizei imageSize, void *data );
```

respecify only a rectangular region of an existing texel array, with incoming data stored in a known compressed image format. The *target*, *level*, *xoffset*, *yoffset*, *zoffset*, *width*, *height*, and *depth* parameters have the same meaning as in **TexSubImage1D**, **TexSubImage2D**, and **TexSubImage3D**. *data* points to compressed image data stored in the compressed image format corresponding to *format*. Since the core GL provides no specific image formats, using any of these six generic compressed internal formats as *format* will result in an `INVALID_ENUM` error.

The image pointed to by *data* and the *imageSize* parameter are interpreted as though they were provided to **CompressedTexImage1D**, **CompressedTexImage2D**, and **CompressedTexImage3D**. These commands do not provide for image format conversion, so an `INVALID_OPERATION` error results if *format* does not match the internal format of the texture image being modified. If the *imageSize* parameter is not consistent with the format, dimensions, and contents of the compressed image (too little or too much data), an `INVALID_VALUE` error results.

As with **CompressedTexImage** calls, compressed internal formats may have additional restrictions on the use of the compressed image specification calls or parameters. Any such restrictions will be documented in the specification defining the compressed internal format; violating these restrictions will result in an `INVALID_OPERATION` error.

Any restrictions imposed by specific compressed internal formats will be invariant, meaning that if the GL accepts and stores a texture image in compressed form, providing the same image to **CompressedTexSubImage1D**, **CompressedTexSubImage2D**, **CompressedTexSubImage3D** will not result in an `INVALID_OPERATION` error if the following restrictions are satisfied:

- *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 or formats described in table 3.19, the texture is stored using one of the RGTC compressed texture image encodings (see appendix C.1). If *internalformat* is an RGTC format, **CompressedTexSubImage1D** will generate an `INVALID_ENUM` error; **CompressedTexSubImage2D** will generate an `INVALID_OPERATION` error if *border* is non-zero; and **CompressedTexSubImage3D** will generate an `INVALID_OPERATION` error if *border* is non-zero or *target* is not `TEXTURE_2D_ARRAY`. Since RGTC images are easily edited along 4×4 texel boundaries, the limitations on subimage location and size are relaxed for **CompressedTexSubImage2D** and **CompressedTexSubImage3D**. These commands will result in an `INVALID_OPERATION` error if one of the following conditions occurs:

- *width* is not a multiple of four or equal to `TEXTURE_WIDTH`.

- *height* is not a multiple of four or equal to `TEXTURE_HEIGHT`.
- *xoffset* or *yoffset* is not a multiple of four.

The contents of any 4×4 block of texels of an RGTC compressed texture image that does not intersect the area being modified are preserved during valid `TexSubImage*` and `CopyTexSubImage*` calls.

3.9.4 Texture Parameters

Various parameters control how the texel array is treated when specified or changed, and when applied to a fragment. Each parameter is set by calling

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

target is the target, either `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`, or `TEXTURE_CUBE_MAP`. *pname* is a symbolic constant indicating the parameter to be set; the possible constants and corresponding parameters are summarized in table 3.20. In the first form of the command, *param* is a value to which to set a single-valued parameter; in the remaining forms, *params* is an array of parameters whose type depends on the parameter being set.

If the value for `TEXTURE_PRIORITY` is specified as an integer, the conversion for signed integers from table 2.10 is applied to convert this value to floating-point, followed by clamping the value to lie in $[0, 1]$.

If the values for `TEXTURE_BORDER_COLOR` are specified with `TexParameterIiv` or `TexParameterIuiv`, the values are unmodified and stored with an internal data type of integer. If specified with `TexParameteriv`, the conversion for signed integers from table 2.10 is applied to convert these values to floating-point. Otherwise the values are unmodified and stored as floating-point.

In the remainder of section 3.9, denote by lod_{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.

Name	Type	Legal Values
TEXTURE_WRAP_S	enum	CLAMP, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER, MIRRORED_REPEAT
TEXTURE_WRAP_T	enum	CLAMP, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER, MIRRORED_REPEAT
TEXTURE_WRAP_R	enum	CLAMP, CLAMP_TO_EDGE, REPEAT, CLAMP_TO_BORDER, MIRRORED_REPEAT
TEXTURE_MIN_FILTER	enum	NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, LINEAR_MIPMAP_LINEAR,
TEXTURE_MAG_FILTER	enum	NEAREST, LINEAR
TEXTURE_BORDER_COLOR	4 floats, integers, or unsigned integers	any 4 values
TEXTURE_PRIORITY	float	any value in $[0, 1]$
TEXTURE_MIN_LOD	float	any value
TEXTURE_MAX_LOD	float	any value
TEXTURE_BASE_LEVEL	integer	any non-negative integer
TEXTURE_MAX_LEVEL	integer	any non-negative integer
TEXTURE_LOD_BIAS	float	any value
DEPTH_TEXTURE_MODE	enum	RED, LUMINANCE, INTENSITY, ALPHA
TEXTURE_COMPARE_MODE	enum	NONE, COMPARE_REF_TO_TEXTURE
TEXTURE_COMPARE_FUNC	enum	LEQUAL, GEQUAL, LESS, GREATER, EQUAL, NOTEQUAL, ALWAYS, NEVER
GENERATE_MIPMAP	boolean	TRUE or FALSE

Table 3.20: 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 3.21: Selection of cube map images based on major axis direction of texture coordinates.

If the value of texture parameter `GENERATE_MIPMAP` is `TRUE`, specifying or changing texel arrays may have side effects, which are discussed in the **Automatic Mipmap Generation** discussion of section 3.9.7.

3.9.5 Depth Component Textures

Depth textures and the depth components of depth/stencil textures can be treated as `RED`, `LUMINANCE`, `INTENSITY` or `ALPHA` textures during texture filtering and application (see section 3.9.14). The initial state for depth and depth/stencil textures treats them as `LUMINANCE` textures except in a forward-compatible context, where the initial state instead treats them as `RED` textures.

3.9.6 Cube Map Texture Selection

When cube map texturing is enabled, the $(s \ t \ r)$ texture coordinates are treated as a direction vector $(r_x \ r_y \ r_z)$ emanating from the center of a cube (the q coordinate can be ignored, since it merely scales the vector without affecting the direction.) At texture application time, the interpolated per-fragment direction vector selects one of the cube map face's two-dimensional images based on the largest magnitude coordinate direction (the major axis direction). If two or more coordinates have the identical magnitude, the implementation may define the rule to disambiguate this situation. The rule must be deterministic and depend only on $(r_x \ r_y \ r_z)$. The target column in table 3.21 explains how the major axis direction maps to the two-dimensional image of a particular cube map target.

Using the s_c , t_c , and m_a determined by the major axis direction as specified in table 3.21, an updated $(s \ t)$ is calculated as follows:

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

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

This new $(s \ t)$ is used to find a texture value in the determined face's two-dimensional texture image using the rules given in sections 3.9.7 through 3.9.8.

3.9.7 Texture Minification

Applying a texture to a primitive implies a mapping from texture image space to framebuffer image space. In general, this mapping involves a reconstruction of the sampled texture image, followed by a homogeneous warping implied by the mapping to framebuffer space, then a filtering, followed finally by a resampling of the filtered, warped, reconstructed image before applying it to a fragment. In the GL this mapping is approximated by one of two simple filtering schemes. One of these schemes is selected based on whether the mapping from texture space to framebuffer space is deemed to *magnify* or *minify* the texture image.

Scale Factor and Level of Detail

The choice is governed by a scale factor $\rho(x, y)$ and the *level-of-detail* parameter $\lambda(x, y)$, defined as

$$\lambda_{base}(x, y) = \log_2[\rho(x, y)] \quad (3.16)$$

$$\lambda'(x, y) = \lambda_{base}(x, y) + \text{clamp}(\text{bias}_{texobj} + \text{bias}_{texunit} + \text{bias}_{shader}) \quad (3.17)$$

$$\lambda = \begin{cases} lod_{max}, & \lambda' > lod_{max} \\ \lambda', & lod_{min} \leq \lambda' \leq lod_{max} \\ lod_{min}, & \lambda' < lod_{min} \\ undefined, & lod_{min} > lod_{max} \end{cases} \quad (3.18)$$

bias_{texobj} is the value of `TEXTURE_LOD_BIAS` for the bound texture object (as described in section 3.9.4). $\text{bias}_{texunit}$ is the value of `TEXTURE_LOD_BIAS` for the current texture unit (as described in section 3.9.13). bias_{shader} is the value of the optional bias parameter in the texture lookup functions available to fragment shaders. If the texture access is performed in a fragment shader without a provided

bias, or outside a fragment shader, then $bias_{shader}$ is zero. The sum of these values is clamped to the range $[-bias_{max}, bias_{max}]$ where $bias_{max}$ is the value of the implementation defined constant `MAX_TEXTURE_LOD_BIAS`.

If $\lambda(x, y)$ is less than or equal to the constant c (see section 3.9.8) the texture is said to be magnified; if it is greater, the texture is minified. Sampling of minified textures is described in the remainder of this section, while sampling of magnified textures is described in section 3.9.8.

The initial values of lod_{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 **Tex-Parameter[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

$$\begin{aligned} u(x, y) &= w_t \times s(x, y) + \delta_u \\ v(x, y) &= h_t \times t(x, y) + \delta_v \\ w(x, y) &= d_t \times r(x, y) + \delta_w \end{aligned} \tag{3.19}$$

where w_t , h_t , and d_t are as defined by equation 3.15 with w_s , h_s , and d_s equal to the width, height, and depth of the image array whose level is $level_{base}$. For a one-dimensional or one-dimensional array texture, define $v(x, y) \equiv 0$ and $w(x, y) \equiv 0$; for a two-dimensional, two-dimensional array, or cube map texture, define $w(x, y) \equiv 0$.

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

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

$$\rho = \max \left\{ \sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial w}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial y}\right)^2} \right\} \tag{3.20}$$

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

For a line, the formula is

$$\rho = \sqrt{\left(\frac{\partial u}{\partial x}\Delta x + \frac{\partial u}{\partial y}\Delta y\right)^2 + \left(\frac{\partial v}{\partial x}\Delta x + \frac{\partial v}{\partial y}\Delta y\right)^2 + \left(\frac{\partial w}{\partial x}\Delta x + \frac{\partial w}{\partial y}\Delta y\right)^2} / l, \quad (3.21)$$

where $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$ with (x_1, y_1) and (x_2, y_2) being the segment's window coordinate endpoints and $l = \sqrt{\Delta x^2 + \Delta y^2}$. For a point, pixel rectangle, or bitmap, $\rho \equiv 1$.

While it is generally agreed that equations 3.20 and 3.21 give the best results when texturing, they are often impractical to implement. Therefore, an implementation may approximate the ideal 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

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

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

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

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

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

$$\begin{aligned} u'(x, y) &= \begin{cases} \text{clamp}(u(x, y), 0, w_t), & \text{TEXTURE_WRAP_S is CLAMP} \\ u(x, y), & \text{otherwise} \end{cases} \\ v'(x, y) &= \begin{cases} \text{clamp}(v(x, y), 0, h_t), & \text{TEXTURE_WRAP_T is CLAMP} \\ v(x, y), & \text{otherwise} \end{cases} \\ w'(x, y) &= \begin{cases} \text{clamp}(w(x, y), 0, h_t), & \text{TEXTURE_WRAP_R is CLAMP} \\ w(x, y), & \text{otherwise} \end{cases} \end{aligned}$$

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

The value assigned to `TEXTURE_MIN_FILTER` is used to determine how the texture value for a fragment is selected.

When the value of `TEXTURE_MIN_FILTER` is `NEAREST`, the texel in the image array of level level_{base} that is nearest (in Manhattan distance) to (u', v', w') is obtained. Let (i, j, k) be integers such that

$$i = \text{wrap}(\lfloor u'(x, y) \rfloor)$$

$$j = \text{wrap}(\lfloor v'(x, y) \rfloor)$$

$$k = \text{wrap}(\lfloor w'(x, y) \rfloor)$$

and the value returned by $\text{wrap}()$ is defined in table 3.22. For a three-dimensional texture, the texel at location (i, j, k) becomes the texture value. For two-dimensional, two-dimensional array, or cube map textures, k is irrelevant, and the texel at location (i, j) becomes the texture value. For one-dimensional texture or one-dimensional array textures, j and k are irrelevant, and the texel at location i becomes the texture value.

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

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

Wrap mode	Result of $\text{wrap}(\text{coord})$
CLAMP	$\begin{cases} \text{clamp}(\text{coord}, 0, \text{size} - 1), & \text{for NEAREST filtering} \\ \text{clamp}(\text{coord}, -1, \text{size}), & \text{for LINEAR filtering} \end{cases}$
CLAMP_TO_EDGE	$\text{clamp}(\text{coord}, 0, \text{size} - 1)$
CLAMP_TO_BORDER	$\text{clamp}(\text{coord}, -1, \text{size})$
REPEAT	$fmod(\text{coord}, \text{size})$
MIRRORED_REPEAT	$(\text{size} - 1) - \text{mirror}(fmod(\text{coord}, 2 \times \text{size}) - \text{size})$

Table 3.22: Texel location wrap mode application. $fmod(a, b)$ returns $a - b \times \lfloor \frac{a}{b} \rfloor$. $\text{mirror}(a)$ returns a if $a \geq 0$, and $-(1 + a)$ otherwise. The values of *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

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

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 3.15. The internal data type of the border values must be consistent with the type returned by the texture as described in section 3.9, or the result is undefined. The border values for texture components stored as fixed-point values are clamped to $[0, 1]$ before they are used. If the texture contains depth components, the first component of `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

$$\begin{aligned} \tau &= (1 - \alpha)(1 - \beta)(1 - \gamma)\tau_{i_0j_0k_0} + \alpha(1 - \beta)(1 - \gamma)\tau_{i_1j_0k_0} \\ &\quad + (1 - \alpha)\beta(1 - \gamma)\tau_{i_0j_1k_0} + \alpha\beta(1 - \gamma)\tau_{i_1j_1k_0} \\ &\quad + (1 - \alpha)(1 - \beta)\gamma\tau_{i_0j_0k_1} + \alpha(1 - \beta)\gamma\tau_{i_1j_0k_1} \\ &\quad + (1 - \alpha)\beta\gamma\tau_{i_0j_1k_1} + \alpha\beta\gamma\tau_{i_1j_1k_1} \end{aligned} \tag{3.22}$$

where τ_{ijk} is the texel at location (i, j, k) in the three-dimensional texture image.

For a two-dimensional, two-dimensional array, or cube map textures,

$$\begin{aligned}\tau = & (1 - \alpha)(1 - \beta)\tau_{i_0j_0} + \alpha(1 - \beta)\tau_{i_1j_0} \\ & + (1 - \alpha)\beta\tau_{i_0j_1} + \alpha\beta\tau_{i_1j_1}\end{aligned}$$

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

$$l = \text{clamp}(\lfloor r + 0.5 \rfloor, 0, d_t - 1).$$

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

$$l = \text{clamp}(\lfloor t + 0.5 \rfloor, 0, h_t - 1).$$

For any texel in the equation above that refers to a border texel outside the defined range of the image, the texel value is taken from the texture border color as with NEAREST filtering.

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. See chapter 4 for discussion of framebuffer objects and their attachments.

- The current `DRAW_FRAMEBUFFER_BINDING` names a framebuffer object F .
- The texture is attached to one of the attachment points, A , of framebuffer object F .
- The value of `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 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*. A mipmap is an ordered set of arrays representing the same image; each array has a resolution lower than the previous one. If the image array of level $level_{base}$, excluding its border, has dimensions $w_t \times h_t \times d_t$, then there are $\lfloor \log_2(maxsize) \rfloor + 1$ levels in the mipmap. where

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

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

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

where

$$\begin{aligned} w_d &= 2^i \\ h_d &= \begin{cases} 1, & \text{for 1D and 1D array textures} \\ 2^i, & \text{otherwise} \end{cases} \\ d_d &= \begin{cases} 2^i, & \text{for 3D textures} \\ 1, & \text{otherwise} \end{cases} \end{aligned}$$

until the last array is reached with dimension $1 \times 1 \times 1$.

Each array in a mipmap is defined using **TexImage3D**, **TexImage2D**, **CopyTexImage2D**, **TexImage1D**, or **CopyTexImage1D**; the array being set is indicated with the level-of-detail argument *level*. Level-of-detail numbers proceed from $level_{base}$ for the original texel array through $p = \lfloor \log_2(maxsize) \rfloor + level_{base}$ with each unit increase indicating an array of half the dimensions of the previous one (rounded down to the next integer if fractional) as already described. All arrays from $level_{base}$ through $q = \min\{p, level_{max}\}$ must be defined, as discussed in section 3.9.10.

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 d th mipmap array is selected, where

$$d = \begin{cases} level_{base}, & \lambda \leq \frac{1}{2} \\ \lceil level_{base} + \lambda + \frac{1}{2} \rceil - 1, & \lambda > \frac{1}{2}, level_{base} + \lambda \leq q + \frac{1}{2} \\ q, & \lambda > \frac{1}{2}, level_{base} + \lambda > q + \frac{1}{2} \end{cases} \quad (3.23)$$

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 3.19, with w_s , h_s , and d_s equal to the width, height, and depth of the image array whose level is d .

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

$$d_1 = \begin{cases} q, & level_{base} + \lambda \geq q \\ \lfloor level_{base} + \lambda \rfloor, & \text{otherwise} \end{cases} \quad (3.24)$$

$$d_2 = \begin{cases} q, & level_{base} + \lambda \geq q \\ d_1 + 1, & \text{otherwise} \end{cases} \quad (3.25)$$

The rules for `NEAREST` or `LINEAR` filtering are then applied to each of the selected arrays, yielding two corresponding texture values τ_1 and τ_2 . Specifically, for level d_1 , the coordinate (u, v, w) is computed as in equation 3.19, with w_s , h_s , and d_s equal to the width, height, and depth of the image array whose level is d_1 . For level d_2 the coordinate (u', v', w') is computed as in equation 3.19, with w_s , h_s , and d_s equal to the width, height, and depth of the image array whose level is d_2 .

The final texture value is then found as

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

Automatic Mipmap Generation

If the value of texture parameter `GENERATE_MIPMAP` is `TRUE`, and a change is made to the interior or border texels of the $level_{base}$ array of a mipmap by one of the texture image specification operations defined in sections 3.9.1 through 3.9.3, then a³ complete set of mipmap arrays (as defined in section 3.9.10) will be computed. Array levels $level_{base} + 1$ through p are replaced with arrays derived from the modified $level_{base}$ array, regardless of their previous contents. All other mipmap arrays, including the $level_{base}$ array, are left unchanged by this computation.

The internal formats and border widths of the derived mipmap arrays all match those of the $level_{base}$ array, and the dimensions of the derived arrays follow the requirements described in section 3.9.10.

The contents of the derived arrays are computed by repeated, filtered reduction of the $level_{base}$ array. For one- and two-dimensional array textures, each layer is filtered independently. No particular filter algorithm is required, though a box filter is recommended as the default filter. In some implementations, filter quality may be affected by hints (section 5.6).

Automatic mipmap generation is available only for non-proxy texture image targets.

Manual Mipmap Generation

Mipmaps can be generated manually with the command

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

where *target* is one of `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`, or `TEXTURE_CUBE_MAP`. Mipmap generation affects the texture image attached to *target*. For cube map textures, an `INVALID_OPERATION` error is generated if the texture bound to *target* is not cube complete, as defined in section 3.9.10.

Mipmap generation replaces texel array levels $level_{base} + 1$ through q with arrays derived from the $level_{base}$ array, as described above for **Automatic Mipmap Generation**. All other mipmap arrays, including the $level_{base}$ array, are left unchanged by this computation. For arrays in the range $level_{base} + 1$ through q , inclusive, automatic and manual mipmap generation generate the same derived arrays, given identical $level_{base}$ arrays.

³Automatic mipmap generation is not performed for changes resulting from rendering operations targeting a texel array bound as a color buffer of a framebuffer object.

3.9.8 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 3.9.7, including the texture coordinate wrap modes specified in table 3.22. The level-of-detail $level_{base}$ texel array is always used for magnification.

Finally, there is the choice of c , the minification vs. magnification switch-over point. If the magnification filter is given by `LINEAR` and the minification filter is given by `NEAREST_MIPMAP_NEAREST` or `NEAREST_MIPMAP_LINEAR`, then $c = 0.5$. This is done to ensure that a minified texture does not appear “sharper” than a magnified texture. Otherwise $c = 0$.

3.9.9 Combined Depth/Stencil Textures

If the texture image has a base internal format of `DEPTH_STENCIL`, then the stencil index texture component is ignored. The texture value τ does not include a stencil index component, but includes only the depth component.

3.9.10 Texture Completeness

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

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

- The set of mipmap arrays $level_{base}$ through q (where q is defined in the **Mipmapping** discussion of section 3.9.7) were each specified with the same internal format.
- The border widths of each array are the same.
- The dimensions of the arrays follow the sequence described in the **Mipmapping** discussion of section 3.9.7.
- $level_{base} \leq level_{max}$
- Each dimension of the $level_{base}$ array is positive.

- If the internal format of the arrays is integer (see tables 3.16- 3.17, TEXTURE_MAG_FILTER must be NEAREST and TEXTURE_MIN_FILTER must be NEAREST or NEAREST_MIPMAP_NEAREST.

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

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

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

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

Effects of Completeness on Texture Application

If one-, two-, or three-dimensional texturing (but not cube map texturing) is enabled for a texture unit at the time a primitive is rasterized, if TEXTURE_MIN_FILTER is one that requires a mipmap, and if the texture image bound to the enabled texture target is not complete, then it is as if texture mapping were disabled for that texture unit.

If cube map texturing is enabled for a texture unit at the time a primitive is rasterized, and if the bound cube map texture is not cube complete, then it is as if texture mapping were disabled for that texture unit. Additionally, if TEXTURE_MIN_FILTER is one that requires a mipmap, and if the texture is not mipmap cube complete, then it is as if texture mapping were disabled for that texture unit.

Effects of Completeness on Texture Image Specification

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

3.9.11 Texture State and Proxy State

The state necessary for texture can be divided into two categories. First, there are the nine sets of mipmap arrays (one each for the one-, two-, and three-dimensional texture targets and six for the cube map texture targets) and their number. Each array has associated with it a width, height (two- and three-dimensional and cube map only), and depth (three-dimensional only), a border width, an integer describing the internal format of the image, eight integer values describing the resolutions of each of the red, green, blue, alpha, luminance, intensity, depth, and stencil components of the image, eight integer values describing the type (unsigned normalized, integer, floating-point, etc.) of each of the components, a boolean describing whether the image is compressed or not, and an integer size of a compressed image. Each initial texel array is null (zero width, height, and depth, zero border width, internal format 1, with the compressed flag set to `FALSE`, a zero compressed size, and zero-sized components). Next, there are the four sets of texture properties, corresponding to the one-, two-, three-dimensional, and cube map texture targets. Each set consists of the selected minification and magnification filters, the wrap modes for *s*, *t* (two- and three-dimensional and cube map only), and *r* (three-dimensional only), the `TEXTURE_BORDER_COLOR`, two floating-point numbers describing the minimum and maximum level of detail, two integers describing the base and maximum mipmap array, a boolean flag indicating whether the texture is resident, a boolean indicating whether automatic mipmap generation should be performed, three integers describing the depth texture mode, compare mode, and compare function, and the priority associated with each set of properties. The value of the resident flag is determined by the GL and may change as a result of other GL operations. The flag may only be queried, not set, by applications (see section 3.9.12). In the initial state, the value assigned to `TEXTURE_MIN_FILTER` is `NEAREST_MIPMAP_LINEAR`, and the value for `TEXTURE_MAG_FILTER` is `LINEAR`. *s*, *t*, and *r* wrap modes are all set to `REPEAT`. 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. `TEXTURE_PRIORITY` is 1.0, and `TEXTURE_BORDER_COLOR` is (0,0,0,0). The value of `GENERATE_MIPMAP` is false. The values of `DEPTH_TEXTURE_MODE`, `TEXTURE_COMPARE_MODE`, and `TEXTURE_COMPARE_FUNC` are `LUMINANCE`, `NONE`, and `LEQUAL` respectively. The initial value of `TEXTURE_RESIDENT` is determined by the GL.

In addition to image arrays for one-, two-, and three-dimensional textures, one- and two-dimensional array textures, and the six image arrays for the cube map texture, partially instantiated image arrays are maintained for one-, two-, and three-dimensional textures and one- and two-dimensional array textures. Additionally,

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

Proxy arrays for one- and two-dimensional textures and one- and two-dimensional array textures are operated on in the same way when **TexImage1D** is executed with *target* specified as `PROXY_TEXTURE_1D`, **TexImage2D** is executed with *target* specified as `PROXY_TEXTURE_2D` or `PROXY_TEXTURE_1D_ARRAY`, 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 associated with any of the proxy textures. Therefore `PROXY_TEXTURE_1D`, `PROXY_TEXTURE_2D`, and `PROXY_TEXTURE_3D`, and `PROXY_TEXTURE_CUBE_MAP` cannot be used as textures, and their images must never be queried using **GetTexImage**. The error `INVALID_ENUM` is generated if this is attempted. Likewise, there is no non level-related state associated with a proxy texture, and **GetTexParameteriv** or **GetTexParameterfv** may not be called with a proxy texture *target*. The error `INVALID_ENUM` is generated if this is attempted.

3.9.12 Texture Objects

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

A texture object is created by *binding* an unused name to `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`, or `TEXTURE_CUBE_MAP`. The binding is effected by calling

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

with *target* set to the desired texture target and *texture* set to the unused name. The resulting texture object is a new state vector, comprising all the state values listed in section 3.9.11, set to the same initial values. If the new texture object is bound to `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`, or `TEXTURE_CUBE_MAP`, it is and remains a one-, two-, three-dimensional, one- or two-dimensional array, or cube map texture respectively until it is deleted.

BindTexture may also be used to bind an existing texture object to either `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`, or `TEXTURE_CUBE_MAP`. The error `INVALID_OPERATION` is generated if an attempt is made to bind a texture object of different dimensionality than the specified *target*. If the bind is successful no change is made to the state of the bound texture object, and any previous binding to *target* is broken.

While a texture object is bound, GL operations on the target to which it is bound affect the bound object, and queries of the target to which it is bound return state from the bound object. If texture mapping of the dimensionality of the target to which a texture object is bound is enabled, the state of the bound texture object directs the texturing operation.

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

Texture objects are deleted by calling

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

textures contains *n* names of texture objects to be deleted. After a texture object is deleted, it has no contents or dimensionality, and its name is again unused. If

a texture that is currently bound to one of the targets `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, `TEXTURE_1D_ARRAY`, `TEXTURE_2D_ARRAY`, or `TEXTURE_CUBE_MAP` is deleted, it is as though **BindTexture** had been executed with the same *target* and *texture* zero. Additionally, special care must be taken when deleting a texture if any of the images of the texture are attached to a framebuffer object. See section 4.4.2 for details.

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

The command

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

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

An implementation may choose to establish a working set of texture objects on which binding operations are performed with higher performance. A texture object that is currently part of the working set is said to be *resident*. The command

```
boolean AreTexturesResident( sizei n, uint *textures,  
                             boolean *residences );
```

returns `TRUE` if all of the *n* texture objects named in *textures* are resident, or if the implementation does not distinguish a working set. If at least one of the texture objects named in *textures* is not resident, then `FALSE` is returned, and the residence of each texture object is returned in *residences*. Otherwise the contents of *residences* are not changed. If any of the names in *textures* are unused or are zero, `FALSE` is returned, the error `INVALID_VALUE` is generated, and the contents of *residences* are indeterminate. The residence status of a single bound texture object can also be queried by calling **GetTexParameteriv** or **GetTexParameterfv** with *target* set to the target to which the texture object is bound, and *pname* set to `TEXTURE_RESIDENT`.

AreTexturesResident indicates only whether a texture object is currently resident, not whether it could not be made resident. An implementation may choose to make a texture object resident only on first use, for example. The client may guide the GL implementation in determining which texture objects should be resident by specifying a priority for each texture object. The command

```
void PrioritizeTextures( sizei n, uint *textures,  
                        clampf *priorities );
```

sets the priorities of the n texture objects named in *textures* to the values in *priorities*. Each priority value is clamped to the range $[0,1]$ before it is assigned. Zero indicates the lowest priority, with the least likelihood of being resident. One indicates the highest priority, with the greatest likelihood of being resident. The priority of a single bound texture object may also be changed by calling **TexParameteri**, **TexParameterf**, **TexParameteriv**, or **TexParameterfv** with *target* set to the target to which the texture object is bound, *pname* set to `TEXTURE_PRIORITY`, and *param* or *params* specifying the new priority value (which is clamped to the range $[0,1]$ before being assigned). **PrioritizeTextures** silently ignores attempts to prioritize unused texture object names or zero (default textures).

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

3.9.13 Texture Environments and Texture Functions

The command

```
void TexEnv{if}( enum target, enum pname, T param );
void TexEnv{if}v( enum target, enum pname, T params );
```

sets parameters of the *texture environment* that specifies how texture values are interpreted when texturing a fragment, or sets per-texture-unit filtering parameters.

target must be one of `POINT_SPRITE`, `TEXTURE_ENV` or `TEXTURE_FILTER_CONTROL`. *pname* is a symbolic constant indicating the parameter to be set. In the first form of the command, *param* is a value to which to set a single-valued parameter; in the second form, *params* is a pointer to an array of parameters: either a single symbolic constant or a value or group of values to which the parameter should be set.

When *target* is `POINT_SPRITE`, point sprite rasterization behavior is affected as described in section 3.4.

When *target* is `TEXTURE_FILTER_CONTROL`, *pname* must be `TEXTURE_LOD_BIAS`. In this case the parameter is a single signed floating point value, $bias_{texunit}$, that biases the level of detail parameter λ as described in section 3.9.7.

When *target* is TEXTURE_ENV, the possible environment parameters are TEXTURE_ENV_MODE, TEXTURE_ENV_COLOR, COMBINE_RGB, COMBINE_ALPHA, RGB_SCALE, ALPHA_SCALE, SRCn_RGB, SRCn_ALPHA, OPERANDn_RGB, and OPERANDn_ALPHA, where $n = 0, 1$, or 2 . TEXTURE_ENV_MODE may be set to one of REPLACE, MODULATE, DECAL, BLEND, ADD, or COMBINE. TEXTURE_ENV_COLOR is set to an RGBA color by providing four single-precision floating-point values. If integers are provided for TEXTURE_ENV_COLOR, then they are converted to floating-point as specified in table 2.10 for signed integers.

The value of TEXTURE_ENV_MODE specifies a *texture function*. The result of this function depends on the fragment and the texel array value. The precise form of the function depends on the base internal formats of the texel arrays that were last specified.

C_f and A_f ⁴ are the primary color components of the incoming fragment; C_s and A_s are the components of the texture source color, derived from the filtered texture values R_t, G_t, B_t, A_t, L_t , and I_t as shown in table 3.23; C_c and A_c are the components of the texture environment color; C_p and A_p are the components resulting from the previous texture environment (for texture environment 0, C_p and A_p are identical to C_f and A_f , respectively); and C_v and A_v are the primary color components computed by the texture function.

If fragment color clamping is enabled, all of these color values, including the results, are clamped to the range $[0, 1]$. If fragment color clamping is disabled, the values are not clamped. The texture functions are specified in tables 3.24, 3.25, and 3.26.

If the value of TEXTURE_ENV_MODE is COMBINE, the form of the texture function depends on the values of COMBINE_RGB and COMBINE_ALPHA, according to table 3.26. The RGB and ALPHA results of the texture function are then multiplied by the values of RGB_SCALE and ALPHA_SCALE, respectively. If fragment color clamping is enabled, the arguments and results used in table 3.26 are clamped to $[0, 1]$. Otherwise, the results are unmodified.

The arguments *Arg0*, *Arg1*, and *Arg2* are determined by the values of SRCn_RGB, SRCn_ALPHA, OPERANDn_RGB and OPERANDn_ALPHA, where $n = 0, 1$, or 2 , as shown in tables 3.27 and 3.28. C_s^n and A_s^n denote the texture source color and alpha from the texture image bound to texture unit n .

The state required for the current texture environment, for each texture unit, consists of a six-valued integer indicating the texture function, an eight-valued integer indicating the RGB combiner function and a six-valued integer indicating the

⁴In the remainder of section 3.9.13, the notation C_x is used to denote each of the three components R_x, G_x , and B_x of a color specified by x . Operations on C_x are performed independently for each color component. The A component of colors is usually operated on in a different fashion, and is therefore denoted separately by A_x .

Texture Base Internal Format	Texture source color	
	C_s	A_s
ALPHA	$(0, 0, 0)$	A_t
LUMINANCE	(L_t, L_t, L_t)	1
LUMINANCE_ALPHA	(L_t, L_t, L_t)	A_t
INTENSITY	(I_t, I_t, I_t)	I_t
RED	$(R_t, 0, 0)$	1
RG	$(R_t, G_t, 0)$	1
RGB	(R_t, G_t, B_t)	1
RGBA	(R_t, G_t, B_t)	A_t

Table 3.23: Correspondence of filtered texture components to texture source components.

Texture Base Internal Format	REPLACE Function	MODULATE Function	DECAL Function
ALPHA	$C_v = C_p$ $A_v = A_s$	$C_v = C_p$ $A_v = A_p A_s$	<i>undefined</i>
LUMINANCE (or 1)	$C_v = C_s$ $A_v = A_p$	$C_v = C_p C_s$ $A_v = A_p$	<i>undefined</i>
LUMINANCE_ALPHA (or 2)	$C_v = C_s$ $A_v = A_s$	$C_v = C_p C_s$ $A_v = A_p A_s$	<i>undefined</i>
INTENSITY	$C_v = C_s$ $A_v = A_s$	$C_v = C_p C_s$ $A_v = A_p A_s$	<i>undefined</i>
RGB, RG, RED, or 3	$C_v = C_s$ $A_v = A_p$	$C_v = C_p C_s$ $A_v = A_p$	$C_v = C_s$ $A_v = A_p$
RGBA or 4	$C_v = C_s$ $A_v = A_s$	$C_v = C_p C_s$ $A_v = A_p A_s$	$C_v = C_p(1 - A_s) + C_s A_s$ $A_v = A_p$

Table 3.24: Texture functions REPLACE, MODULATE, and DECAL.

Texture Base Internal Format	BLEND Function	ADD Function
ALPHA	$C_v = C_p$ $A_v = A_p A_s$	$C_v = C_p$ $A_v = A_p A_s$
LUMINANCE (or 1)	$C_v = C_p(1 - C_s) + C_c C_s$ $A_v = A_p$	$C_v = C_p + C_s$ $A_v = A_p$
LUMINANCE_ALPHA (or 2)	$C_v = C_p(1 - C_s) + C_c C_s$ $A_v = A_p A_s$	$C_v = C_p + C_s$ $A_v = A_p A_s$
INTENSITY	$C_v = C_p(1 - C_s) + C_c C_s$ $A_v = A_p(1 - A_s) + A_c A_s$	$C_v = C_p + C_s$ $A_v = A_p + A_s$
RGB, RG, RED, or 3	$C_v = C_p(1 - C_s) + C_c C_s$ $A_v = A_p$	$C_v = C_p + C_s$ $A_v = A_p$
RGBA or 4	$C_v = C_p(1 - C_s) + C_c C_s$ $A_v = A_p A_s$	$C_v = C_p + C_s$ $A_v = A_p A_s$

Table 3.25: Texture functions BLEND and ADD.

ALPHA combiner function, six four-valued integers indicating the combiner RGB and ALPHA source arguments, three four-valued integers indicating the combiner RGB operands, three two-valued integers indicating the combiner ALPHA operands, and four floating-point environment color values. In the initial state, the texture and combiner functions are each MODULATE, the combiner RGB and ALPHA sources are each TEXTURE, PREVIOUS, and CONSTANT for sources 0, 1, and 2 respectively, the combiner RGB operands for sources 0 and 1 are each SRC_COLOR, the combiner RGB operand for source 2, as well as for the combiner ALPHA operands, are each SRC_ALPHA, and the environment color is (0, 0, 0, 0).

The state required for the texture filtering parameters, for each texture unit, consists of a single floating-point level of detail bias. The initial value of the bias is 0.0.

3.9.14 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. The format of the resulting texture sample is determined by the value of DEPTH_TEXTURE_MODE.

COMBINE_RGB	Texture Function
REPLACE	$Arg0$
MODULATE	$Arg0 * Arg1$
ADD	$Arg0 + Arg1$
ADD_SIGNED	$Arg0 + Arg1 - 0.5$
INTERPOLATE	$Arg0 * Arg2 + Arg1 * (1 - Arg2)$
SUBTRACT	$Arg0 - Arg1$
DOT3_RGB	$4 \times ((Arg0_r - 0.5) * (Arg1_r - 0.5) + (Arg0_g - 0.5) * (Arg1_g - 0.5) + (Arg0_b - 0.5) * (Arg1_b - 0.5))$
DOT3_RGBA	$4 \times ((Arg0_r - 0.5) * (Arg1_r - 0.5) + (Arg0_g - 0.5) * (Arg1_g - 0.5) + (Arg0_b - 0.5) * (Arg1_b - 0.5))$

COMBINE_ALPHA	Texture Function
REPLACE	$Arg0$
MODULATE	$Arg0 * Arg1$
ADD	$Arg0 + Arg1$
ADD_SIGNED	$Arg0 + Arg1 - 0.5$
INTERPOLATE	$Arg0 * Arg2 + Arg1 * (1 - Arg2)$
SUBTRACT	$Arg0 - Arg1$

Table 3.26: COMBINE texture functions. The scalar expression computed for the DOT3_RGB and DOT3_RGBA functions is placed into each of the 3 (RGB) or 4 (RGBA) components of the output. The result generated from COMBINE_ALPHA is ignored for DOT3_RGBA.

SRC n _RGB	OPERAND n _RGB	Argument
TEXTURE	SRC_COLOR	C_s
	ONE_MINUS_SRC_COLOR	$1 - C_s$
	SRC_ALPHA	A_s
	ONE_MINUS_SRC_ALPHA	$1 - A_s$
TEXTURE n	SRC_COLOR	C_s^n
	ONE_MINUS_SRC_COLOR	$1 - C_s^n$
	SRC_ALPHA	A_s^n
	ONE_MINUS_SRC_ALPHA	$1 - A_s^n$
CONSTANT	SRC_COLOR	C_c
	ONE_MINUS_SRC_COLOR	$1 - C_c$
	SRC_ALPHA	A_c
	ONE_MINUS_SRC_ALPHA	$1 - A_c$
PRIMARY_COLOR	SRC_COLOR	C_f
	ONE_MINUS_SRC_COLOR	$1 - C_f$
	SRC_ALPHA	A_f
	ONE_MINUS_SRC_ALPHA	$1 - A_f$
PREVIOUS	SRC_COLOR	C_p
	ONE_MINUS_SRC_COLOR	$1 - C_p$
	SRC_ALPHA	A_p
	ONE_MINUS_SRC_ALPHA	$1 - A_p$

Table 3.27: Arguments for COMBINE_RGB functions.

SRC n _ALPHA	OPERAND n _ALPHA	Argument
TEXTURE	SRC_ALPHA	A_s
	ONE_MINUS_SRC_ALPHA	$1 - A_s$
TEXTURE n	SRC_ALPHA	A_s^n
	ONE_MINUS_SRC_ALPHA	$1 - A_s^n$
CONSTANT	SRC_ALPHA	A_c
	ONE_MINUS_SRC_ALPHA	$1 - A_c$
PRIMARY_COLOR	SRC_ALPHA	A_f
	ONE_MINUS_SRC_ALPHA	$1 - A_f$
PREVIOUS	SRC_ALPHA	A_p
	ONE_MINUS_SRC_ALPHA	$1 - A_p$

Table 3.28: Arguments for COMBINE_ALPHA functions.

Depth Texture Comparison Mode

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

Let D_t be the depth texture value and D_{ref} be the reference value, defined as follows:

- For fixed-function, non-cubemap texture lookups, D_{ref} is the interpolated r texture coordinate.
- For fixed-function, cubemap texture lookups, D_{ref} is the interpolated q texture coordinate.
- For texture lookups generated by an OpenGL Shading Language lookup function, D_{ref} is the reference value for depth comparisons provided by the lookup function.

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

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

$$r = D_t$$

If the value of `TEXTURE_COMPARE_MODE` is `COMPARE_REF_TO_TEXTURE`, then r depends on the texture comparison function as shown in table 3.29.

The resulting r is assigned to R_t , L_t , I_t , or A_t if the value of `DEPTH_TEXTURE_MODE` is respectively `RED`, `LUMINANCE`, `INTENSITY`, or `ALPHA`.

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

3.9.15 sRGB Texture Color Conversion

If the currently bound texture's internal format is one of `SRGB`, `SRGB8`, `SRGB_ALPHA`, `SRGB8_ALPHA8`, `SLUMINANCE_ALPHA`, `SLUMINANCE8_ALPHA8`,

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 3.29: Depth texture comparison functions.

SLUMINANCE, SLUMINANCE8, COMPRESSED_SRGB, COMPRESSED_SRGB_ALPHA, COMPRESSED_SLUMINANCE, or COMPRESSED_SLUMINANCE_ALPHA, the red, green, and blue components are converted from an sRGB color space to a linear color space as part of filtering described in sections 3.9.7 and 3.9.8. Any alpha component is left unchanged. Ideally, implementations should perform this color conversion on each sample prior to filtering but implementations are allowed to perform this conversion after filtering (though this post-filtering approach is inferior to converting from sRGB prior to filtering).

The conversion from an sRGB encoded component, c_s , to a linear component, c_l , is as follows.

$$c_l = \begin{cases} \frac{c_s}{12.92}, & c_s \leq 0.04045 \\ \left(\frac{c_s + 0.055}{1.055}\right)^{2.4}, & c_s > 0.04045 \end{cases} \quad (3.26)$$

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

3.9.16 Shared Exponent Texture Color Conversion

If the currently bound texture's internal format is `RGB9_E5`, the red, green, blue, and shared bits are converted to color components (prior to filtering) using shared exponent decoding. The component red_s , $green_s$, $blue_s$, and exp_{shared} values (see section 3.9.1) are treated as unsigned integers and are converted to red , $green$, and $blue$ as follows:

$$\begin{aligned} red &= red_s 2^{exp_{shared}-B} \\ green &= green_s 2^{exp_{shared}-B} \\ blue &= blue_s 2^{exp_{shared}-B} \end{aligned}$$

3.9.17 Texture Application

Texturing is enabled or disabled using the generic **Enable** and **Disable** commands, respectively, with the symbolic constants `TEXTURE_1D`, `TEXTURE_2D`, `TEXTURE_3D`, or `TEXTURE_CUBE_MAP` to enable the one-, two, three-dimensional, or cube map texture, respectively. If both two- and one-dimensional textures are enabled, the two-dimensional texture is used. If the three-dimensional and either of the two- or one-dimensional textures is enabled, the three-dimensional texture is used. If the cube map texture and any of the three-, two-, or one-dimensional textures is enabled, then cube map texturing is used.

If all texturing is disabled, a rasterized fragment is passed on unaltered to the next stage of the GL (although its texture coordinates may be discarded). Otherwise, a texture value is found according to the parameter values of the currently bound texture image of the appropriate dimensionality using the rules given in sections 3.9.6 through 3.9.8. This texture value is used along with the incoming fragment in computing the texture function indicated by the currently bound texture environment. The result of this function replaces the incoming fragment's primary R, G, B, and A values. These are the color values passed to subsequent operations. Other data associated with the incoming fragment remain unchanged, except that the texture coordinates may be discarded.

Note that the texture value may contain R , G , B , A , L , I , or D components, but it does not contain an S component. If the texture's base internal format is `DEPTH_STENCIL`, for the purposes of texture application it is as if the base internal format were `DEPTH_COMPONENT`.

Each texture unit is enabled and bound to texture objects independently from the other texture units. Each texture unit follows the precedence rules for one-, two-, three-dimensional, and cube map textures. Thus texture units can be performing

texture mapping of different dimensionalities simultaneously. Each unit has its own enable and binding states.

Each texture unit is paired with an environment function, as shown in figure 3.11. The second texture function is computed using the texture value from the second texture, the fragment resulting from the first texture function computation and the second texture unit's environment function. If there is a third texture, the fragment resulting from the second texture function is combined with the third texture value using the third texture unit's environment function and so on. The texture unit selected by **ActiveTexture** determines which texture unit's environment is modified by **TexEnv** calls.

If the value of `TEXTURE_ENV_MODE` is `COMBINE`, the texture function associated with a given texture unit is computed using the values specified by `SRCn_RGB`, `SRCn_ALPHA`, `OPERANDn_RGB` and `OPERANDn_ALPHA`. If `TEXTUREn` is specified as `SRCn_RGB` or `SRCn_ALPHA`, the texture value from texture unit n will be used in computing the texture function for this texture unit.

Texturing is enabled and disabled individually for each texture unit. If texturing is disabled for one of the units, then the fragment resulting from the previous unit is passed unaltered to the following unit. Individual texture units beyond those specified by `MAX_TEXTURE_UNITS` are always treated as disabled.

If a texture unit is disabled or has an invalid or incomplete texture (as defined in section 3.9.10) bound to it, then blending is disabled for that texture unit. If the texture environment for a given enabled texture unit references a disabled texture unit, or an invalid or incomplete texture that is bound to another unit, then the results of texture blending are undefined.

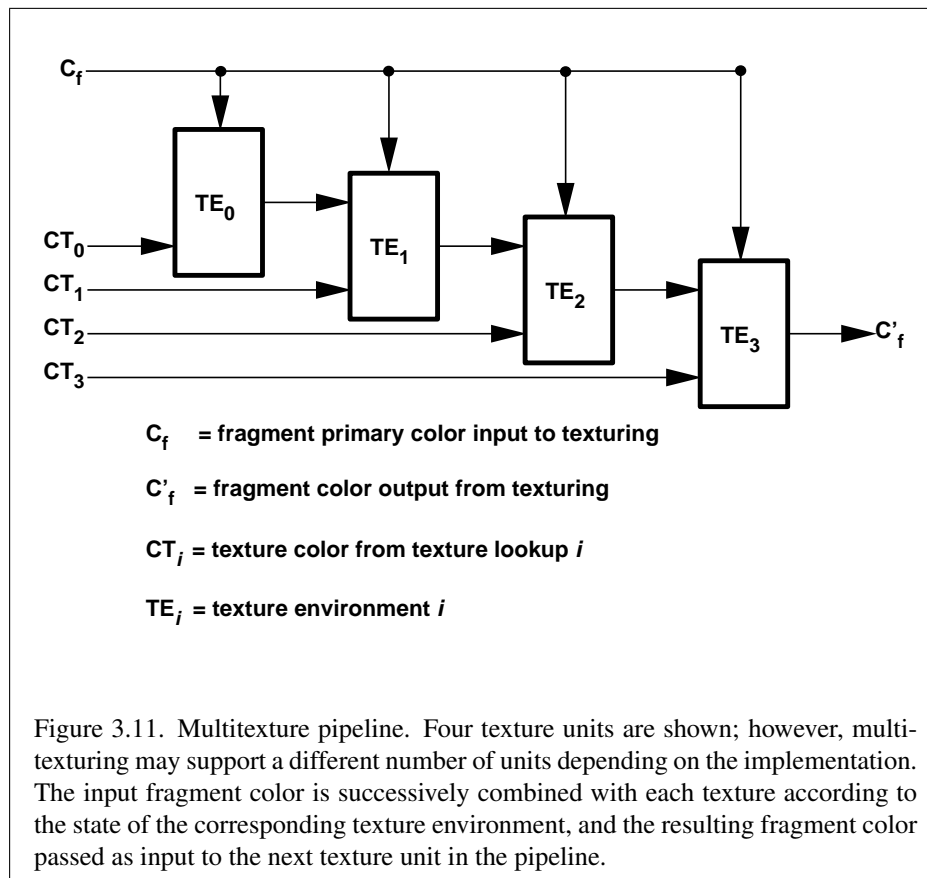
The required state, per texture unit, is four bits indicating whether each of one-, two-, three-dimensional, or cube map texturing is enabled or disabled. In the initial state, all texturing is disabled for all texture units.

3.10 Color Sum

At the beginning of color sum, a fragment has two RGBA colors: a primary color c_{pri} (which texturing, if enabled, may have modified) and a secondary color c_{sec} .

If color sum is enabled, the R, G, and B components of these two colors are summed to produce a single post-texturing RGBA color c . The A component of c is taken from the A component of c_{pri} ; the A component of c_{sec} is unused. If color sum is disabled, then c_{pri} is assigned to c . If fragment color clamping is enabled, the components of c are then clamped to the range $[0, 1]$.

Color sum is enabled or disabled using the generic **Enable** and **Disable** commands, respectively, with the symbolic constant `COLOR_SUM`. If lighting is enabled



and if a vertex shader is not active, the color sum stage is always applied, ignoring the value of `COLOR_SUM`.

The state required is a single bit indicating whether color sum is enabled or disabled. In the initial state, color sum is disabled.

Color sum has no effect in color index mode, or if a fragment shader is active.

3.11 Fog

If enabled, fog blends a fog color with a rasterized fragment's post-texturing color using a blending factor f . Fog is enabled and disabled with the **Enable** and **Disable** commands using the symbolic constant `FOG`.

This factor f is computed according to one of three equations:

$$f = \exp(-d \cdot c), \quad (3.27)$$

$$f = \exp(-(d \cdot c)^2), \text{ or } \quad (3.28)$$

$$f = \frac{e - c}{e - s} \quad (3.29)$$

If a vertex shader is active, or if the fog source, as defined below, is `FOG_COORD`, then c is the interpolated value of the fog coordinate for this fragment. Otherwise, if the fog source is `FRAGMENT_DEPTH`, then c is the eye-coordinate distance from the eye, $(0, 0, 0, 1)$ in eye coordinates, to the fragment center. The equation and the fog source, along with either d or e and s , is specified with

```
void Fog{if}( enum pname, T param );
void Fog{if}v( enum pname, T params );
```

If $pname$ is `FOG_MODE`, then $param$ must be, or $params$ must point to an integer that is one of the symbolic constants `EXP`, `EXP2`, or `LINEAR`, in which case equation 3.27, 3.28, or 3.29, respectively, is selected for the fog calculation (if, when 3.29 is selected, $e = s$, results are undefined). If $pname$ is `FOG_COORD_SRC`, then $param$ must be, or $params$ must point to an integer that is one of the symbolic constants `FRAGMENT_DEPTH` or `FOG_COORD`. If $pname$ is `FOG_DENSITY`, `FOG_START`, or `FOG_END`, then $param$ is or $params$ points to a value that is d , s , or e , respectively. If d is specified less than zero, the error `INVALID_VALUE` results.

An implementation may choose to approximate the eye-coordinate distance from the eye to each fragment center by $|z_e|$. Further, f need not be computed at

each fragment, but may be computed at each vertex and interpolated as other data are.

No matter which equation and approximation is used to compute f , the result is clamped to $[0, 1]$ to obtain the final f .

f is used differently depending on whether the GL is in RGBA or color index mode. In RGBA mode, if C_r represents a rasterized fragment's R, G, or B value, then the corresponding value produced by fog is

$$C = fC_r + (1 - f)C_f.$$

(The rasterized fragment's A value is not changed by fog blending.) The R, G, B, and A values of C_f are specified by calling **Fog** with *pname* equal to `FOG_COLOR`; in this case *params* points to four values comprising C_f . If these are not floating-point values, then they are converted to floating-point using the conversion given in table 2.10 for signed integers. If fragment color clamping is enabled, the components of C_r and C_f and the result C are clamped to the range $[0, 1]$ before the fog blend is performed.

In color index mode, the formula for fog blending is

$$I = i_r + (1 - f)i_f$$

where i_r is the rasterized fragment's color index and i_f is a single-precision floating-point value. $(1 - f)i_f$ is rounded to the nearest fixed-point value with the same number of bits to the right of the binary point as i_r , and the integer portion of I is masked (bitwise ANDed) with $2^n - 1$, where n is the number of bits in a color in the color index buffer (buffers are discussed in chapter 4). The value of i_f is set by calling **Fog** with *pname* set to `FOG_INDEX` and *param* being or *params* pointing to a single value for the fog index. The integer part of i_f is masked with $2^n - 1$.

The state required for fog consists of a three valued integer to select the fog equation, three floating-point values d , e , and s , an RGBA fog color and a fog color index, a two-valued integer to select the fog coordinate source, and a single bit to indicate whether or not fog is enabled. In the initial state, fog is disabled, `FOG_COORD_SRC` is `FRAGMENT_DEPTH`, `FOG_MODE` is `EXP`, $d = 1.0$, $e = 1.0$, and $s = 0.0$; $C_f = (0, 0, 0, 0)$ and $i_f = 0$.

Fog has no effect if a fragment shader is active.

3.12 Fragment Shaders

The sequence of operations that are applied to fragments that result from rasterizing a point, line segment, polygon, pixel rectangle or bitmap as described in

sections 3.9 through 3.11 is a fixed functionality method for processing such fragments. Applications can more generally describe the operations that occur on such fragments by using a *fragment shader*.

A fragment shader is an array of strings containing source code for the operations that are meant to occur on each fragment that results from rasterizing a point, line segment, polygon, pixel rectangle or bitmap. The language used for fragment shaders is described in the OpenGL Shading Language Specification.

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

Fragment shaders are created as described in section 2.20.1 using a *type* parameter of `FRAGMENT_SHADER`. They are attached to and used in program objects as described in section 2.20.2.

When the program object currently in use includes a fragment shader, its fragment shader is considered *active*, and is used to process fragments. If the program object has no fragment shader, or no program object is currently in use, the fixed-function fragment processing operations described in previous sections are used.

Results of rasterization are undefined if any of the selected draw buffers of the draw framebuffer have an integer format and no fragment shader is active.

3.12.1 Shader Variables

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

Fragment shaders can read varying variables that correspond to the attributes of the fragments produced by rasterization. The OpenGL Shading Language Specification defines a set of built-in varying variables that can be accessed by a fragment shader. These built-in varying variables include the data associated with a fragment that are used for fixed-function fragment processing, such as the fragment's position, color, secondary color, texture coordinates, fog coordinate, and eye z coordinate.

Additionally, when a vertex shader is active, it may define one or more *varying* variables (see section 2.20.3 and the OpenGL Shading Language Specification). These values are, if not flat shaded, interpolated across the primitive being ren-

dered. The results of these interpolations are available when varying variables of the same name are defined in the fragment shader.

User-defined varying variables are not saved in the current raster position. When processing fragments generated by the rasterization of a pixel rectangle or bitmap, that values of user-defined varying variables are undefined. Built-in varying variables have well-defined values.

A fragment shader can also write to varying out variables. Values written to these variables are used in the subsequent per-fragment operations. Varying out variables can be used to write floating-point, integer or unsigned integer values destined for buffers attached to a framebuffer object, or destined for color buffers attached to the default framebuffer. The **Shader Outputs** subsection of section 3.12.2 describes how to direct these values to buffers.

3.12.2 Shader Execution

If a fragment shader is active, the executable version of the fragment shader is used to process incoming fragment values that are the result of point, line segment, polygon, pixel rectangle or bitmap rasterization rather than the fixed-function fragment processing described in sections 3.9 through 3.11. In particular,

- The texture environments and texture functions described in section 3.9.13 are not applied.
- Texture application as described in section 3.9.17 is not applied.
- Color sum as described in section 3.10 is not applied.
- Fog as described in section 3.11 is not applied.

Texture Access

The **Shader Only Texturing** subsection of section 2.20.4 describes texture lookup functionality accessible to a vertex shader. The texel fetch and texture size query functionality described there also applies to fragment shaders.

When a texture lookup is performed in a fragment shader, the GL computes the filtered texture value τ in the manner described in sections 3.9.7 and 3.9.8, and converts it to a texture source color C_s according to table 3.23 (section 3.9.13). The GL returns a four-component vector (R_s, G_s, B_s, A_s) to the fragment shader. For the purposes of level-of-detail calculations, the derivatives $\frac{du}{dx}$, $\frac{du}{dy}$, $\frac{dv}{dx}$, $\frac{dv}{dy}$, $\frac{dw}{dx}$ and $\frac{dw}{dy}$ may be approximated by a differencing algorithm as detailed in section 8.8 of the OpenGL Shading Language specification.

Texture lookups involving textures with depth component data can either return the depth data directly or return the results of a comparison with the D_{ref} value (see section 3.9.14) used to perform the lookup. The comparison operation is requested in the shader by using any of the shadow sampler types and in the texture using the `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 3.9.10, 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 3.6.5). The $\frac{1}{w}$ value is computed from the w_c coordinate (see section 2.12), which is the result of the product of the projection matrix and the vertex's eye coordinates.

The built-in variables `gl_Color` and `gl_SecondaryColor` hold the R, G, B, and A components, respectively, of the fragment color and secondary color. If

the primary color or the secondary color components are represented by the GL as fixed-point values, they undergo an implied conversion to floating-point. This conversion must leave the values 0 and 1 invariant. Floating-point color components (resulting from a disabled vertex color clamp) are unmodified.

The built-in variable `gl_FrontFacing` is set to `TRUE` if the fragment is generated from a front facing primitive, and `FALSE` otherwise. For fragments generated from polygon, triangle, or quadrilateral primitives (including ones resulting from polygons rendered as points or lines), the determination is made by examining the sign of the area computed by equation 2.6 of section 2.19.1 (including the possible reversal of this sign controlled by **FrontFace**). If the sign is positive, fragments generated by the primitive are front facing; otherwise, they are back facing. All other fragments are considered front facing.

The built-in variable `gl_PrimitiveID` is filled with the number of primitives processed by the rasterizer since the last time **Begin** was called (directly or indirectly via vertex array functions). The first primitive generated after a **Begin** is numbered zero, and the primitive ID counter is incremented after every individual point, line, or polygon primitive is processed. For polygons drawn in point or line mode, the primitive ID counter is incremented only once, even though multiple points or lines may be drawn. For `QUADS` and `QUAD_STRIP` primitives that are decomposed into triangles, the primitive ID is incremented after each complete quad is processed.

The value of `gl_PrimitiveID` is undefined for fragments generated by `POLYGON` primitives or from **DrawPixels** or **Bitmap** commands. Additionally, `gl_PrimitiveID` is only defined under the same conditions that `gl_VertexID` is defined, as described under “Shader Inputs” in section 2.20.4.

Shader Outputs

The OpenGL Shading Language specification describes the values that may be output by a fragment shader. These outputs are split into two categories, user-defined varying out variables and built-in variables. The built-in variables are `gl_FragColor`, `gl_FragData[n]`, and `gl_FragDepth`. If fragment color clamping is enabled and the color buffer has a fixed- or floating-point format, the final fragment color, fragment data, or varying out variable values written by a fragment shader are clamped to the range $[0, 1]$ and are optionally converted to fixed-point as described in section 2.19.9. Only user-defined varying out variables declared as a floating-point type are clamped and may be converted. If fragment color clamping is disabled, or the color buffer has an integer format, the final fragment color, fragment data, or varying out variable values are not modified. For fixed-point depth buffers, the final fragment depth written by a fragment shader is

first clamped to $[0, 1]$ and then converted to fixed-point as if it were a window z value (see section 2.12.1). For floating-point depth buffers, conversion is not performed but clamping is. Note that the depth range computation is not applied here, only the conversion to fixed-point.

Color values written by a fragment shader may be floating-point, signed integer, or unsigned integer. If the color buffer has a fixed-point format, color values are assumed to be floating-point and are converted to fixed-point as described in section 2.19.9; otherwise no type conversion is applied. If the values written by the fragment shader do not match the format(s) of the corresponding color buffer(s), the result is undefined.

Writing to `gl_FragColor` specifies the fragment color (color number zero) that will be used by subsequent stages of the pipeline. Writing to `gl_FragData[n]` specifies the value of fragment color number n . Any colors, or color components, associated with a fragment that are not written by the fragment shader are undefined. A fragment shader may not statically assign values to more than one of `gl_FragColor`, `gl_FragData`, and any user-defined varying out variable. In this case, a compile or link error will result. A shader statically assigns a value to a variable if, after pre-processing, it contains a statement that would write to the variable, whether or not run-time flow of control will cause that statement to be executed.

Writing to `gl_FragDepth` specifies the depth value for the fragment being processed. If the active fragment shader does not statically assign a value to `gl_FragDepth`, then the depth value generated during rasterization is used by 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 assigned binding of a varying out variable would cause the GL to reference a non-existent fragment color number (one greater than or equal to `MAX_DRAW_BUFFERS`). **LinkProgram** will also fail if more than one varying out variable is bound to the same number. This type of aliasing is not allowed.

BindFragDataLocation may be issued before any shader objects are attached to a program object. Hence it is allowed to bind any name (except a name starting with `gl_`) to a color number, including a name that is never used as a varying out variable in any fragment shader object. Assigned bindings for variables that do not exist are ignored.

After a program object has been linked successfully, the bindings of varying out variable names to color numbers can be queried. The command

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

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

3.13 Antialiasing Application

If antialiasing is enabled for the primitive from which a rasterized fragment was produced, then the computed coverage value is applied to the fragment. In RGBA mode, the value is multiplied by the fragment's alpha (A) value to yield a final alpha value. In color index mode, the value is used to set the low order bits of the color index value as described in section 3.3. The coverage value is applied separately to each fragment color.

3.14 Multisample Point Fade

Finally, if multisampling is enabled and the rasterized fragment results from a point primitive, then the computed fade factor from equation 3.2 is applied to the fragment. In RGBA mode, the fade factor is multiplied by the fragment's alpha value

to yield a final alpha value. In color index mode, the fade factor has no effect. The fade factor is applied separately to each fragment color.

Chapter 4

Per-Fragment Operations and the Framebuffer

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

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

For the default framebuffer, the color buffers are the *front left* buffer, the *front right* buffer, the *back left* buffer, the *back right* buffer, and some number of *auxiliary* buffers. Typically the contents of the front buffers are displayed on a color monitor while the contents of the back buffers are invisible. (Monoscopic contexts display only the front left buffer; stereoscopic contexts display both the front left and the front right buffers.) The contents of the auxiliary buffers are never visible. All color buffers must have the same number of bitplanes, although an implementation or context may choose not to provide right buffers, back buffers, or auxiliary buffers at all. Further, an implementation or context may not provide depth, stencil, or accumulation buffers. If no default framebuffer is associated with the GL

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

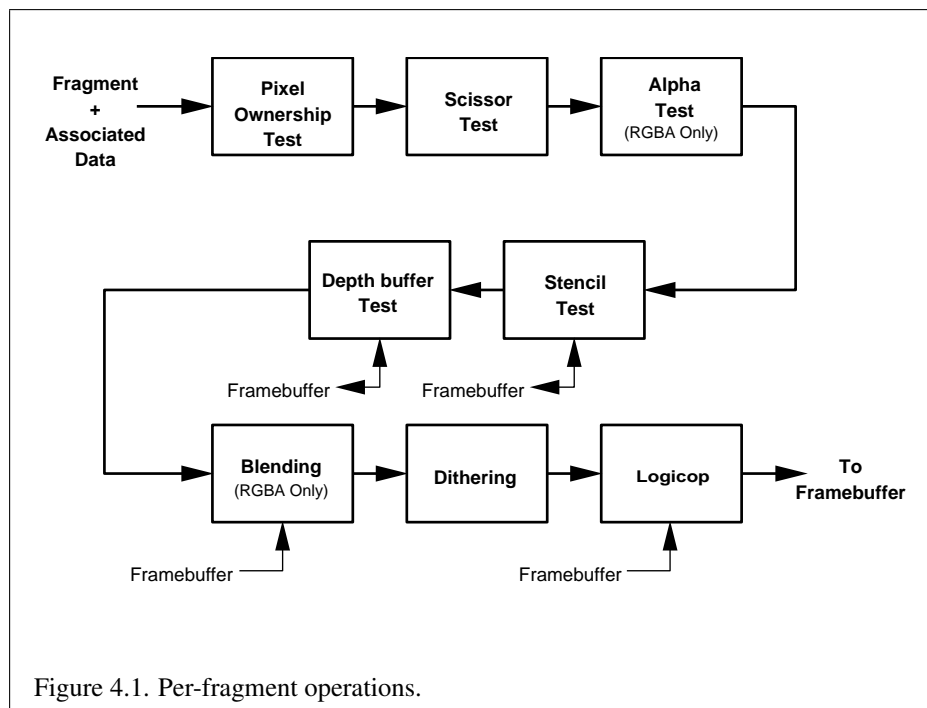
Each pixel in a color buffer consists of either a single unsigned integer color index or up to four color components. The four color components are named R, G, B, and A, in that order; color buffers are not required to have all four color components. R, G, B, and A components may be represented as unsigned fixed-point, floating-point, signed integer, or unsigned integer values; all components must have the same representation. Each pixel in a depth buffer consists of a single unsigned integer value in the format described in section 2.12.1 or a floating-point value. Each pixel in a stencil buffer consists of a single unsigned integer value. Each pixel in an accumulation buffer consists of up to four color components fixed-point values. If an accumulation buffer is present, it must have at least as many bitplanes per component as in the color buffers.

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

The GL has two active framebuffers; the *draw framebuffer* is the destination for rendering operations, and the *read framebuffer* is the source for readback operations. The same framebuffer may be used for both drawing and reading. Section 4.4.2 describes the mechanism for controlling framebuffer usage.

The default framebuffer is initially used as the draw and read framebuffer¹, 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.

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



4.1 Per-Fragment Operations

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

4.1.1 Pixel Ownership Test

The first test is to determine if the pixel at location (x_w, y_w) in the framebuffer is currently owned by the GL (more precisely, by this GL context). If it is not, the window system decides the fate the incoming fragment. Possible results are that the fragment is discarded or that some subset of the subsequent per-fragment operations are applied to the fragment. This test allows the window system to control the GL's behavior, for instance, when a GL window is obscured.

If the draw framebuffer is a framebuffer object (see section 4.2.1), the pixel ownership test always passes, since the pixels of framebuffer objects are owned by

the GL, not the window system. If the draw framebuffer is the default framebuffer, the window system controls pixel ownership.

4.1.2 Scissor Test

The scissor test determines if (x_w, y_w) lies within the scissor rectangle defined by four values. These values are set with

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

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

4.1.3 Multisample Fragment Operations

This step modifies fragment alpha and coverage values based on the values of `SAMPLE_ALPHA_TO_COVERAGE`, `SAMPLE_ALPHA_TO_ONE`, `SAMPLE_COVERAGE`, `SAMPLE_COVERAGE_VALUE`, and `SAMPLE_COVERAGE_INVERT`. No changes to the fragment alpha or coverage values are made at this step if `MULTISAMPLE` is disabled, or if the value of `SAMPLE_BUFFERS` is not one.

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

No specific algorithm is required for converting the sample alpha values to a temporary coverage value. It is intended that the number of 1's in the temporary coverage be proportional to the set of alpha values for the fragment, with all 1's

corresponding to the maximum of all alpha values, and all 0's corresponding to all alpha values being 0. The alpha values used to generate a coverage value are clamped to the range $[0, 1]$. It is also intended that the algorithm be pseudo-random in nature, to avoid image artifacts due to regular coverage sample locations. The algorithm can and probably should be different at different pixel locations. If it does differ, it should be defined relative to window, not screen, coordinates, so that rendering results are invariant with respect to window position.

Next, if `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 Alpha Test

This step applies only in RGBA mode, and only if the color buffer has a fixed-point or floating-point format. In color index mode, or if the color buffer has an integer format, proceed to the next operation.

The alpha test discards a fragment conditional on the outcome of a comparison between the incoming fragment's alpha value and a constant value. If multiple colors are written by a fragment shader, the alpha value of fragment color zero is used to determine the result of the alpha test. The comparison is enabled or disabled with the generic **Enable** and **Disable** commands using the symbolic constant `ALPHA_TEST`. When disabled, it is as if the comparison always passes. The test is controlled with

```
void AlphaFunc( enum func, clampf ref );
```

func is a symbolic constant indicating the alpha test function; *ref* is a reference value. When performing the alpha test, the GL will convert the reference value to the same representation as the the fragment's alpha value (floating-point or fixed-point). For fixed-point, the reference value is converted according to the rules given for an A component in section 2.19.9 and the fragment's alpha value is rounded to the nearest integer.

The possible constants specifying the test function are NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GEQUAL, GREATER, or NOTEQUAL, meaning pass the fragment never, always, if the fragment's alpha value is less than, less than or equal to, equal to, greater than or equal to, greater than, or not equal to the reference value, respectively.

The required state consists of the floating-point reference value, an eight-valued integer indicating the comparison function, and a bit indicating if the comparison is enabled or disabled. The initial state is for the reference value to be 0 and the function to be ALWAYS. Initially, the alpha test is disabled.

4.1.5 Stencil Test

The stencil test conditionally discards a fragment based on the outcome of a comparison between the value in the stencil buffer at location (x_w, y_w) and a reference value. The test is enabled or disabled with the **Enable** and **Disable** commands, using the symbolic constant STENCIL_TEST. When disabled, the stencil test and associated modifications are not made, and the fragment is always passed.

The stencil test is controlled with

```
void StencilFunc( enum func, int ref, uint mask );
void StencilFuncSeparate( enum face, enum func, int ref,
    uint mask );
void StencilOp( enum sfail, enum dpfail, enum dppass );
void StencilOpSeparate( enum face, enum sfail, enum dpfail,
    enum dppass );
```

There are two sets of stencil-related state, the front stencil state set and the back stencil state set. Stencil tests and writes use the front set of stencil state when processing fragments rasterized from non-polygon primitives (points, lines, bitmaps, image rectangles) and front-facing polygon primitives while the back set of stencil state is used when processing fragments rasterized from back-facing polygon primitives. For the purposes of stencil testing, a primitive is still considered a polygon even if the polygon is to be rasterized as points or lines due to the current polygon mode. Whether a polygon is front- or back-facing is determined in the same manner used for two-sided lighting and face culling (see sections 2.19.1 and 3.6.1).

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, bit-wise inverting it, incrementing without saturation, and decrementing without saturation.

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

The same symbolic values are given to indicate the stencil action if the depth buffer test (see section 4.1.6) fails (*dpfail*), or if it passes (*dppass*).

If the stencil test fails, the incoming fragment is discarded. The state required consists of the most recent values passed to **StencilFunc** or **StencilFuncSeparate** and to **StencilOp** or **StencilOpSeparate**, and a bit indicating whether stencil testing is enabled or disabled. In the initial state, stenciling is disabled, the front and back stencil reference value are both zero, the front and back stencil comparison functions are both ALWAYS, and the front and back stencil mask are both all ones. Initially, all three front and back stencil operations are KEEP.

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

4.1.6 Depth Buffer Test

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

The comparison is specified with

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

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

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

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

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

4.1.7 Occlusion Queries

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

When an occlusion query is started, the samples-passed count maintained by the GL is set to zero. When an occlusion query is active, the samples-passed count is incremented for each fragment that passes the depth test. If the value of `SAMPLE_BUFFERS` is 0, then the samples-passed count is incremented by 1 for each fragment. If the value of `SAMPLE_BUFFERS` is 1, then the samples-passed

count is incremented by the number of samples whose coverage bit is set. However, implementations, at their discretion, may instead increase the samples-passed count by the value of `SAMPLES` if any sample in the fragment is covered.

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

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

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

4.1.8 Blending

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

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

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

Blending is dependent on the incoming fragment's alpha value and that of the corresponding currently stored pixel. Blending applies only in RGBA mode; and only if the color buffer has a fixed-point or floating-point format. In color index mode, or if the color buffer has an integer format, proceed to the next operation.

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

```
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_BUFFERi`. If the color

buffer associated with `DRAW_BUFFERi` is one of `FRONT`, `BACK`, `LEFT`, `RIGHT`, or `FRONT_AND_BACK` (specifying multiple color buffers), then the state enabled or disabled is applicable for all of the buffers. Blending can be enabled or disabled for all draw buffers using **Enable** or **Disable** with the symbolic constant `BLEND`. If blending is disabled for a particular draw buffer, or if logical operation on color values is enabled (section 4.1.11), proceed to the next operation.

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

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

Blend Equation

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

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

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

Fixed-point destination (framebuffer) components are taken to be fixed-point values represented according to the scheme in section 2.19.9 (Final Color Processing). Constant color components, floating-point destination components, and source (fragment) components are taken to be floating point values. If source components are represented internally by the GL as fixed-point values, they are also interpreted according to section 2.19.9.

Prior to blending, each fixed-point color component undergoes an implied conversion to floating-point. This conversion must leave the values 0 and 1 invariant. Blending computations are treated as if carried out in floating-point.

If `FRAMEBUFFER_SRGB` is enabled and the value of `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING` for the framebuffer attachment corresponding to the destination buffer is `SRGB` (see section 6.1.3), the R, G, and B destination color values (after conversion from fixed-point to floating-point) are considered to be encoded for the sRGB color space and hence must be linearized prior to their use in blending. Each R, G, and B component is converted in the same fashion described for sRGB texture components in section 3.9.15.

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.

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* 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` for the destination RGB and alpha functions. The initial constant blend color is $(R, G, B, A) = (0, 0, 0, 0)$. Initially, blending is disabled for all draw buffers.

The value of the blend enable for draw buffer *i* can be queried by calling **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.

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

4.1.9 sRGB Conversion

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

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

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

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

$$c_s = c_l.$$

The resulting c_s values for R, G, and B, and the unmodified A form a new RGBA color value. If the color buffer is fixed-point, each component is clamped to the range $[0, 1]$ and then converted to a fixed-point value in the manner described in section 2.19.9. The resulting four values are sent to the subsequent dithering operation.

4.1.10 Dithering

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

In color index mode dithering selects either the largest representable index that is less than or equal to the incoming color value, c , or the smallest representable index that is greater than or equal to c . If one of the two indices does not exist, then the selection defaults to the other value.

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

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

4.1.11 Logical Operation

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

The logical operation on color indices is enabled or disabled with **Enable** or **Disable** using the symbolic constant `INDEX_LOGIC_OP`. (For compatibility with GL version 1.0, the symbolic constant `LOGIC_OP` may also be used.) The logical operation on color values is enabled or disabled with **Enable** or **Disable** using the symbolic constant `COLOR_LOGIC_OP`. If the logical operation is enabled for color values, it is as if blending were disabled, regardless of the value of `BLEND`. If multiple fragment colors are being written to multiple buffers (see section 4.2.1), the logical operation is computed and applied separately for each fragment color and the corresponding buffer.

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

The logical operation is selected by

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

op is a symbolic constant; the possible constants and corresponding operations are enumerated in table 4.3. In this table, s is the value of the incoming fragment and d is the value stored in the framebuffer. The numeric values assigned to the symbolic constants are the same as those assigned to the corresponding symbolic values in the X window system.

Logical operations are performed independently for each color index buffer that is selected for writing, or for each red, green, blue, and alpha value of each

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	$s \text{ xor } d$
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.

color buffer that is selected for writing. The required state is an integer indicating the logical operation, and two bits indicating whether the logical operation is enabled or disabled. The initial state is for the logic operation to be given by `COPY`, and to be disabled.

4.1.12 Additional Multisample Fragment Operations

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

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

Stencil, depth, blending, and dithering operations are performed for a pixel sample only if that sample's fragment coverage bit is a value of 1. If the corre-

sponding coverage bit is 0, no operations are performed for that sample.

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

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

4.2 Whole Framebuffer Operations

The preceding sections described the operations that occur as individual fragments are sent to the framebuffer. This section describes operations that control or affect the whole framebuffer.

4.2.1 Selecting a Buffer for Writing

The first such operation is controlling the color buffers into which each of the fragment color values is written. This is accomplished with either **DrawBuffer** or **DrawBuffers**.

The command

```
void DrawBuffer( enum buf );
```

defines the set of color buffers to which fragment color zero is written. *buf* must be one of the values from tables 4.4 or 4.5. In addition, acceptable values for *buf* depend on whether the GL is using the default framebuffer (i.e., `DRAW_FRAMEBUFFER_BINDING` is zero), or a framebuffer object (i.e.,

`DRAW_FRAMEBUFFER_BINDING` is non-zero). In the initial state, the GL is bound to the default framebuffer. For more information about framebuffer objects, see section 4.4.

If the GL is bound to the default framebuffer, then *buf* must be one of the values listed in table 4.4, which summarizes the constants and the buffers they indicate. In this case, *buf* is a symbolic constant specifying zero, one, two, or four buffers for writing. These constants refer to the four potentially visible buffers (front left, front right, back left, and back right), and to the auxiliary buffers. Arguments other than `AUXi` that omit reference to `LEFT` or `RIGHT` refer to both left and right buffers. Arguments other than `AUXi` that omit reference to `FRONT` or `BACK` refer to both front and back buffers. `AUXi` enables drawing only to auxiliary buffer *i*. Each `AUXi` adheres to `AUXi = AUX0 + i`, and *i* must be in the range 0 to the value of `AUX_BUFFERS` minus one.

If the GL is bound to a framebuffer object, *buf* must be one of the values listed in table 4.5, which summarizes the constants and the buffers they indicate. In this case, *buf* is a symbolic constant specifying a single color buffer for writing. Specifying `COLOR_ATTACHMENTi` enables drawing only to the image attached to the framebuffer at `COLOR_ATTACHMENTi`. Each `COLOR_ATTACHMENTi` adheres to `COLOR_ATTACHMENTi = COLOR_ATTACHMENT0 + i`. The initial value of `DRAW_BUFFER` for framebuffer objects is `COLOR_ATTACHMENT0`.

If the GL is bound to the default framebuffer and **DrawBuffer** is supplied with a constant (other than `NONE`) that does not indicate any of the color buffers allocated to the GL context, the error `INVALID_OPERATION` results.

If the GL is bound to a framebuffer object and *buf* is one of the constants from table 4.4, then the error `INVALID_OPERATION` results. If *buf* is `COLOR_ATTACHMENTm` and *m* is greater than or equal to the value of `MAX_COLOR_ATTACHMENTS`, then the error `INVALID_VALUE` results.

If **DrawBuffer** is supplied with a constant that is legal for neither the default framebuffer nor a framebuffer object, then the error `INVALID_ENUM` results.

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

The command

```
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

Symbolic Constant	Front Left	Front Right	Back Left	Back Right	Aux i
NONE					
FRONT_LEFT	•				
FRONT_RIGHT		•			
BACK_LEFT			•		
BACK_RIGHT				•	
FRONT	•	•			
BACK			•	•	
LEFT	•		•		
RIGHT		•		•	
FRONT_AND_BACK	•	•	•	•	
AUX i					•

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

Symbolic Constant	Meaning
NONE	No buffer
COLOR_ATTACHMENT i (see caption)	Output fragment color to image attached at color attachment point i

Table 4.5: Arguments to **DrawBuffer(s)** and **ReadBuffer** when the context is bound to a framebuffer object, and the buffers they indicate. i in COLOR_ATTACHMENT i may range from zero to the value of MAX_COLOR_ATTACHMENTS - 1.

Symbolic Constant	Front Left	Front Right	Back Left	Back Right	Aux <i>i</i>
NONE					
FRONT_LEFT	•				
FRONT_RIGHT		•			
BACK_LEFT			•		
BACK_RIGHT				•	
AUX <i>i</i>					•

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

for the constants in *bufs* depend on whether the GL is using the default framebuffer (i.e., `DRAW_FRAMEBUFFER_BINDING` is zero), or a framebuffer object (i.e., `DRAW_FRAMEBUFFER_BINDING` is non-zero). For more information about framebuffer objects, see section 4.4.

If the GL is bound to the default framebuffer, then each of the constants must be one of the values listed in table 4.6.

If the GL is bound to an framebuffer object, then each of the constants must be one of the values listed in table 4.5.

In both cases, the draw buffers being defined correspond in order to the respective fragment colors. The draw buffer for fragment colors beyond *n* is set to NONE.

The maximum number of draw buffers is implementation dependent and must be at least 1. 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 than once in the array pointed to by *bufs*. Specifying a buffer more than once will result in the error `INVALID_OPERATION`.

If fixed-function fragment shading is being performed, **DrawBuffers** specifies a set of draw buffers into which the fragment color is written.

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

defined varying out variables, the values of the fragment colors following shader execution are undefined, and may differ for each fragment color.

For both the default framebuffer and framebuffer objects, the constants `FRONT`, `BACK`, `LEFT`, `RIGHT`, and `FRONT_AND_BACK` are not valid in the *bufs* array passed to **DrawBuffers**, and will result in the error `INVALID_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 `FRONT` if there are no back buffers; otherwise it is `BACK`. For framebuffer objects, in the initial state the draw buffer for fragment color zero is `COLOR_ATTACHMENT0`. For both the default framebuffer and framebuffer objects, the initial state of draw buffers for fragment colors other than zero is `NONE`.

The value of the draw buffer selected for fragment color i can be queried by calling **GetIntegerv** with the symbolic constant `DRAW_BUFFER i` . `DRAW_BUFFER` is equivalent to `DRAW_BUFFER0`.

4.2.2 Fine Control of Buffer Updates

Writing of bits to each of the logical framebuffers after all per-fragment operations have been performed may be *masked*. The commands

```
void IndexMask( uint mask );
```

```

void ColorMask(boolean r, boolean g, boolean b,
               boolean a);
void ColorMaski(uint buf, boolean r, boolean g,
               boolean b, boolean a);

```

control writes to the active draw buffers.

The least significant n bits of *mask*, where n is the number of bits in a color index buffer, specify a mask. Where a 1 appears in this mask, the corresponding bit in the color index buffer (or buffers) is written; where a 0 appears, the bit is not written. This mask applies only in color index mode.

In RGBA mode, **ColorMask** and **ColorMaski** are used to mask the writing of R, G, B and A values to the draw buffer or buffers. **ColorMaski** sets the mask for a particular draw buffer. The mask for `DRAW_BUFFERi` 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_BUFFERi`. If `DRAW_BUFFERi` is one of FRONT, BACK, LEFT, RIGHT, or FRONT_AND_BACK (specifying multiple color buffers) then the mask is applied to all of the buffers.

ColorMask sets the mask for all draw buffers to the same values as specified by *r*, *g*, *b*, and *a*.

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

In the initial state, all bits (in color index mode) and all color values (in RGBA mode) are enabled for writing for all draw buffers.

The value of the color writemask for draw buffer *i* can be queried by calling **GetBooleani_v** with *target* COLOR_WRITEMASK and *index* *i*. The value of the color writemask for draw buffer zero may also be queried by calling **GetBooleanv** with *value* COLOR_WRITEMASK.

The depth buffer can be enabled or disabled for writing z_w values using

```

void DepthMask(boolean mask);

```

If *mask* is non-zero, the depth buffer is enabled for writing; otherwise, it is disabled. In the initial state, the depth buffer is enabled for writing.

The commands

```

void StencilMask(uint mask);
void StencilMaskSeparate(enum face, uint mask);

```

control the writing of particular bits into the stencil planes.

The least significant *s* bits of *mask* comprise an integer mask (*s* is the number of bits in the stencil buffer), just as for **IndexMask**. The *face* parameter of **StencilMaskSeparate** can be `FRONT`, `BACK`, or `FRONT_AND_BACK` and indicates whether the front or back stencil mask state is affected. **StencilMask** sets both front and back stencil mask state to identical values.

Fragments generated by front facing primitives use the front mask and fragments generated by back facing primitives use the back mask (see section 4.1.5). The clear operation always uses the front stencil write mask when clearing the stencil buffer.

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

Fine Control of Multisample Buffer Updates

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

4.2.3 Clearing the Buffers

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

```
void Clear(bitfield buf);
```

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

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

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

The command

```
void ClearIndex( float index );
```

sets the clear color index. *index* is converted to a fixed-point value with unspecified precision to the left of the binary point; the integer part of this value is then masked with $2^m - 1$, where m is the number of bits in a color index value stored in the framebuffer.

The command

```
void ClearDepth( clampd d );
```

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

The command

```
void ClearStencil( int s );
```

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

The command

```
void ClearAccum( float r, float g, float b, float a );
```

takes four floating-point arguments that are the values, in order, to which to set the R, G, B, and A values of the accumulation buffer (see the next section). These values are clamped to the range $[-1, 1]$ when they are specified.

When **Clear** is called, the only per-fragment operations that are applied (if enabled) are the pixel ownership test, the scissor test, and dithering. The masking operations described in section 4.2.2 are also applied. If a buffer is not present, then a **Clear** directed at that buffer has no effect. Fixed-point RGBA color buffers are cleared to color values derived by clamping each component of the clear color to $[0, 1]$ and converting to fixed-point according to the rules of section 2.19.9. The result of clearing integer color buffers is undefined.

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

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

```
ClearBuffer{if ui}v(enum buffer, const T *value );
```

where *buffer* identifies a buffer to clear, and *value* specifies the value or values to clear it to. If the default framebuffer is bound to the draw framebuffer, *buffer* must be one of FRONT_LEFT, FRONT_RIGHT, BACK_LEFT, BACK_RIGHT, AUX*i*, DEPTH_BUFFER, or STENCIL_BUFFER, identifying a color buffer, the depth buffer, or the stencil buffer. If a framebuffer object is bound to the draw framebuffer, *buffer* may be one of the framebuffer attachment points listed in table 4.12.

If *buffer* identifies an RGBA color buffer, *value* points to a four-element vector specifying the R, G, B, and A color to clear the buffer to. The **ClearBufferfv**, **ClearBufferiv**, and **ClearBufferuiv** commands should be used to clear fixed- and floating-point, signed integer, and unsigned integer color buffers respectively. Clamping and conversion for fixed-point color buffers are performed in the same fashion as **ClearColor**.

If *buffer* identifies a depth buffer (DEPTH_BUFFER or DEPTH_ATTACHMENT), *value* points to the single depth value to clear the buffer to. Clamping and type conversion for fixed-point depth buffers are performed in the same fashion as **ClearDepth**. Only **ClearBufferfv** should be used to clear depth buffers.

If *buffer* identifies a stencil buffer (STENCIL_BUFFER or STENCIL_ATTACHMENT), *value* points to the single stencil value to clear the buffer to. Masking and type conversion are performed in the same fashion as **ClearStencil**. Only **ClearBufferiv** should be used to clear stencil buffers.

The command

```
void ClearBufferfi(enum buffer, float depth, int stencil );
```

clears both depth and stencil buffers of the currently bound draw framebuffer. *buffer* must be DEPTH_STENCIL_ATTACHMENT. *depth* and *stencil* are the values to clear the depth and stencil buffers to, respectively. Clamping and type conversion of *depth* for fixed-point depth buffers is performed in the same fashion as **ClearDepth**. Masking of *stencil* for stencil buffers is performed in the same fashion as **ClearStencil**. **ClearBufferfi** is equivalent to clearing the depth

and stencil buffers separately, but may be faster when a buffer of internal format `DEPTH_STENCIL` is being cleared.

The result of **ClearBuffer** is undefined if no conversion between the type of the specified *value* and the type of the buffer being cleared is defined (for example, if **ClearBufferi** is called with a fixed- or floating-point *buffer*, or if **ClearBufferf** is called with a signed or unsigned integer buffer). This is not an error.

When **ClearBuffer** is called, the same per-fragment and masking operations defined for **Clear** are applied.

Errors

ClearBuffer generates an `INVALID_ENUM` error if *buffer* does not identify a valid buffer in or attachment of the currently bound draw framebuffer, or if *buffer* is not one of the buffer names specified as valid for that form of **ClearBuffer** command.

ClearBuffer generates an `INVALID_OPERATION` error if *buffer* identifies a color index buffer.

Clearing the Multisample Buffer

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

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

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

4.2.4 The Accumulation Buffer

Each portion of a pixel in the accumulation buffer consists of four values: one for each of R, G, B, and A. The accumulation buffer is controlled exclusively through the use of

```
void Accum(enum op, float value);
```

(except for clearing it). *op* is a symbolic constant indicating an accumulation buffer operation, and *value* is a floating-point value to be used in that operation. The possible operations are `ACCUM`, `LOAD`, `RETURN`, `MULT`, and `ADD`.

When the scissor test is enabled (section 4.1.2), then only those pixels within the current scissor box are updated by any **Accum** operation; otherwise, all pixels in the window are updated. The accumulation buffer operations apply identically

to every affected pixel, so we describe the effect of each operation on an individual pixel. Accumulation buffer values are taken to be signed values in the range $[-1, 1]$. Using `ACCUM` obtains R, G, B, and A components from the buffer currently selected for reading (section 4.3.2). If the color buffer is fixed-point, each component is considered as a fixed-point value in $[0, 1]$ (see section 2.19.9) and is converted to floating-point. Each result is then multiplied by *value*. The results of this multiplication are then added to the corresponding color component currently in the accumulation buffer, and the resulting color value replaces the current accumulation buffer color value.

The `LOAD` operation has the same effect as `ACCUM`, but the computed values replace the corresponding accumulation buffer components rather than being added to them.

The `RETURN` operation takes each color value from the accumulation buffer, multiplies each of the R, G, B, and A components by *value*. If fragment color clamping is enabled, the results are then clamped to the range $[0, 1]$. The resulting color value is placed in the buffers currently enabled for color writing as if it were a fragment produced from rasterization, except that the only per-fragment operations that are applied (if enabled) are the pixel ownership test, the scissor test (section 4.1.2), and dithering (section 4.1.10). Color masking (section 4.2.2) is also applied.

The `MULT` operation multiplies each R, G, B, and A in the accumulation buffer by *value* and then returns the scaled color components to their corresponding accumulation buffer locations. `ADD` is the same as `MULT` except that *value* is added to each of the color components.

The color components operated on by **Accum** must be clamped only if the operation is `RETURN`. In this case, a value sent to the enabled color buffers is first clamped to $[0, 1]$. Otherwise, results are undefined if the result of an operation on a color component is out of the range $[-1, 1]$.

If there is no accumulation buffer; if the `DRAW_FRAMEBUFFER` and `READ_FRAMEBUFFER` bindings (see section 4.4.4) do not refer to the same object; or if the GL is in color index mode, **Accum** generates the error `INVALID_OPERATION`.

No state (beyond the accumulation buffer itself) is required for accumulation buffering.

4.3 Drawing, Reading, and Copying Pixels

Pixels may be written to and read from the framebuffer using the **DrawPixels** and **ReadPixels** commands. **CopyPixels** can be used to copy a block of pixels from

one portion of the framebuffer to another.

4.3.1 Writing to the Stencil or Depth/Stencil Buffers

The operation of **DrawPixels** was described in section 3.7.4, except if the *format* argument was `STENCIL_INDEX` or `DEPTH_STENCIL`. In this case, all operations described for **DrawPixels** take place, but window (x, y) coordinates, each with the corresponding stencil index, or depth value and stencil index, are produced in lieu of fragments. Each coordinate-data pair is sent directly to the per-fragment operations, bypassing the texture, fog, and antialiasing application stages of rasterization. Each pair is then treated as a fragment for purposes of the pixel ownership and scissor tests; all other per-fragment operations are bypassed. Finally, each stencil index is written to its indicated location in the framebuffer, subject to the current front stencil mask (set with **StencilMask** or **StencilMaskSeparate**). If a depth component is present, and the setting of **DepthMask** is not `FALSE`, it is also written to the framebuffer; the setting of **DepthTest** is ignored.

The error `INVALID_OPERATION` results if the *format* argument is `STENCIL_INDEX` and there is no stencil buffer, or if *format* is `DEPTH_STENCIL` and there is not both a depth buffer and a stencil buffer.

4.3.2 Reading Pixels

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

Initially, zero is bound for the `PIXEL_PACK_BUFFER`, indicating that image read and query commands such as **ReadPixels** return pixels results into client memory pointer parameters. However, if a non-zero buffer object is bound as the current pixel pack buffer, then the pointer parameter is treated as an offset into the designated buffer object.

Pixels are read using

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

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

ReadPixels generates an `INVALID_OPERATION` error if `READ_FRAMEBUFFER_BINDING` (see section 4.4) is non-zero, the read frame-

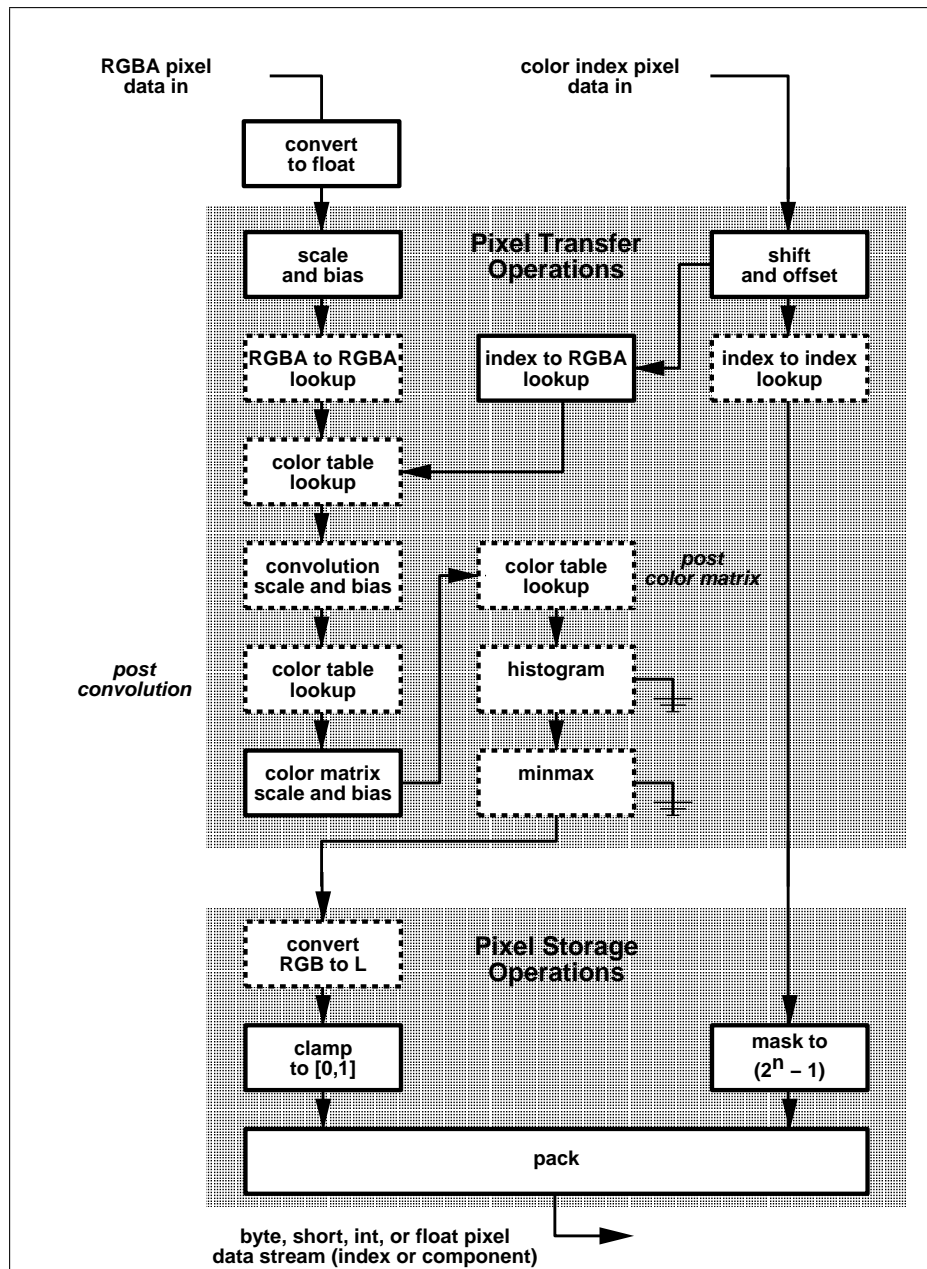


Figure 4.2. Operation of **ReadPixels**. Operations in dashed boxes may be enabled or disabled, except in the case of "convert RGB to L", which is only applied when reading color data in luminosity formats. RGBA and color index pixel paths are shown; depth and stencil pixel paths are not shown.

Parameter Name	Type	Initial Value	Valid Range
PACK_SWAP_BYTES	boolean	FALSE	TRUE/FALSE
PACK_LSB_FIRST	boolean	FALSE	TRUE/FALSE
PACK_ROW_LENGTH	integer	0	$[0, \infty)$
PACK_SKIP_ROWS	integer	0	$[0, \infty)$
PACK_SKIP_PIXELS	integer	0	$[0, \infty)$
PACK_ALIGNMENT	integer	4	1,2,4,8
PACK_IMAGE_HEIGHT	integer	0	$[0, \infty)$
PACK_SKIP_IMAGES	integer	0	$[0, \infty)$

Table 4.7: **PixelStore** parameters pertaining to **ReadPixels**, **GetColorTable**, **GetConvolutionFilter**, **GetSeparableFilter**, **GetHistogram**, **GetMinmax**, **GetPolygonStipple**, and **GetTexImage**.

buffer is framebuffer complete, and the value of `SAMPLE_BUFFERS` for the read framebuffer is greater than zero.

Obtaining Pixels from the Framebuffer

If the *format* is `DEPTH_COMPONENT`, then values are obtained from the depth buffer. If there is no depth buffer, the error `INVALID_OPERATION` occurs.

If there is a multisample buffer (the value of `SAMPLE_BUFFERS` is one), then values are obtained from the depth samples in this buffer. It is recommended that the depth value of the centermost sample be used, though implementations may choose any function of the depth sample values at each pixel.

If the *format* is `DEPTH_STENCIL`, then values are taken from both the depth buffer and the stencil buffer. If there is no depth buffer or if there is no stencil buffer, then the error `INVALID_OPERATION` occurs. If the *type* parameter is not `UNSIGNED_INT_24_8` or `FLOAT_32_UNSIGNED_INT_24_8_REV`, then the error `INVALID_ENUM` occurs.

If there is a multisample buffer, then values are obtained from the depth and stencil samples in this buffer. It is recommended that the depth and stencil values of the centermost sample be used, though implementations may choose any function of the depth and stencil sample values at each pixel.

If the *format* is `STENCIL_INDEX`, then values are taken from the stencil buffer; again, if there is no stencil buffer, the error `INVALID_OPERATION` occurs.

If there is a multisample buffer, then values are obtained from the stencil samples in this buffer. It is recommended that the stencil value of the centermost sam-

ple 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_ATTACHMENTi` enables reading from the image attached to the framebuffer at `COLOR_ATTACHMENTi`. For framebuffer objects, the initial setting for **ReadBuffer** is `COLOR_ATTACHMENT0`.

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

ReadPixels obtains values from the selected buffer from each pixel with lower left hand corner at $(x + i, y + j)$ for $0 \leq i < width$ and $0 \leq j < height$; this pixel is said to be the *i*th pixel in the *j*th row. If any of these pixels lies outside of the window allocated to the current GL context, or outside of the image attached to the currently bound framebuffer object, then the values obtained for those pixels are undefined. When `READ_FRAMEBUFFER_BINDING` is zero, values are also undefined for individual pixels that are not owned by the current context.

Otherwise, **ReadPixels** obtains values from the selected buffer, regardless of how those values were placed there.

If the GL is in RGBA mode, and *format* is one of RED, GREEN, BLUE, ALPHA, RG, RGB, RGBA, BGR, BGRA, LUMINANCE, or LUMINANCE_ALPHA, then red, green, blue, and alpha values are obtained from the selected buffer at each pixel location. If the framebuffer does not support alpha values then the A that is obtained is 1.0. If *format* is COLOR_INDEX and the GL is in RGBA mode then the error INVALID_OPERATION occurs. If the GL is in color index mode, and *format* is not DEPTH_COMPONENT, DEPTH_STENCIL, or STENCIL_INDEX, then the color index is obtained at each pixel location.

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

When READ_FRAMEBUFFER_BINDING is non-zero, the red, green, blue, and alpha values are obtained by first reading the internal component values of the corresponding value in the image attached to the selected logical buffer. Internal components are converted to an RGBA color by taking each R, G, B, and A component present according to the base internal format of the buffer (as shown in table 3.15). If G, B, or A values are not present in the internal format, they are taken to be zero, zero, and one respectively.

Conversion of RGBA values

This step applies only if the GL is in RGBA mode, and then only if *format* is not STENCIL_INDEX, DEPTH_COMPONENT, or DEPTH_STENCIL. The R, G, B, and A values form a group of elements. For a fixed-point color buffer, each element is taken to be a fixed-point value in $[0, 1]$ with m bits, where m is the number of bits in the corresponding color component of the selected buffer (see section 2.19.9). For an integer or floating-point color buffer, the elements are unmodified.

Conversion of Depth values

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

Pixel Transfer Operations

This step is actually the sequence of steps that was described separately in section 3.7.5. After the processing described in that section is completed, groups are processed as described in the following sections.

Conversion to L

This step applies only to RGBA component groups. If the *format* is either LUMINANCE or LUMINANCE_ALPHA, a value *L* is computed as

$$L = R + G + B$$

where *R*, *G*, and *B* are the values of the R, G, and B components. The single computed *L* component replaces the R, G, and B components in the group.

Final Conversion

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

For a floating-point RGBA color, if *type* is not one of `FLOAT`, `UNSIGNED_INT_5_9_9_9_REV`, or `UNSIGNED_INT_10F_11F_11F_REV`; if `CLAMP_READ_COLOR` is `TRUE`; or if `CLAMP_READ_COLOR` is `FIXED_ONLY` and the selected color buffer is a fixed-point buffer, each component is first clamped to $[0, 1]$. Then the appropriate conversion formula from table 4.9 is applied to the component.

In the special case of calling **ReadPixels** with *type* of `UNSIGNED_INT_10F_11F_11F_REV` and *format* of `RGB`, conversion is performed as follows: the returned data are packed into a series of `uint` values. The red, green, and blue components are converted to unsigned 11-bit floating-point, unsigned 11-bit floating-point, and unsigned 10-bit floating point as described in sections 2.1.3 and 2.1.4. The resulting red 11 bits, green 11 bits, and blue 10 bits are then packed as the 1st, 2nd, and 3rd components of the `UNSIGNED_INT_10F_11F_11F_REV` format as shown in table 3.11.

In the special case of calling **ReadPixels** with *type* of `UNSIGNED_INT_5_9_9_9_REV` and *format* `RGB`, the conversion is performed as follows: the returned data are packed into a series of `uint` values. The red, green, and blue components are converted to *red_s*, *green_s*, *blue_s*, and *exp_{shared}* integers as described in section 3.9.1 when *internalformat* is `RGB9_E5`. The *red_s*, *green_s*,

<i>type</i> Parameter	Index Mask
UNSIGNED_BYTE	$2^8 - 1$
BITMAP	1
BYTE	$2^7 - 1$
UNSIGNED_SHORT	$2^{16} - 1$
SHORT	$2^{15} - 1$
UNSIGNED_INT	$2^{32} - 1$
INT	$2^{31} - 1$
UNSIGNED_INT_24_8	$2^8 - 1$
FLOAT_32_UNSIGNED_INT_24_8_REV	$2^8 - 1$

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

$blue_s$, and exp_{shared} are then packed as the 1st, 2nd, 3rd, and 4th components of the UNSIGNED_INT_5_9_9_9_REV format as shown in table 3.11.

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

Placement in Pixel Pack Buffer or Client Memory

If a pixel pack buffer is bound (as indicated by a non-zero value of PIXEL_PACK_BUFFER_BINDING), *data* is an offset into the pixel pack buffer and the pixels are packed into the buffer relative to this offset; otherwise, *data* is a pointer to a block client memory and the pixels are packed into the client memory relative to the pointer. If a pixel pack buffer object is bound and packing the pixel data according to the pixel pack storage state would access memory beyond the size of the pixel pack buffer's memory size, an INVALID_OPERATION error results. If a pixel pack buffer object is bound and *data* is not evenly divisible by the number of basic machine units needed to store in memory the corresponding GL data type from table 3.5 for the *type* parameter, an INVALID_OPERATION error results.

Groups of elements are placed in memory just as they are taken from memory for **DrawPixels**. That is, the *i*th group of the *j*th row (corresponding to the *i*th pixel in the *j*th row) is placed in memory just where the *i*th group of the *j*th row would be taken from for **DrawPixels**. See **Unpacking** under section 3.7.4. The only difference is that the storage mode parameters whose names begin with PACK_ are used instead of those whose names begin with UNPACK_. If the *format* is RED, GREEN, BLUE, ALPHA, or LUMINANCE, only the corresponding single element is written. Likewise if the *format* is RG, LUMINANCE_ALPHA, RGB, or BGR, only the

<i>type</i> 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.

corresponding two or three elements are written. Otherwise all the elements of each group are written.

4.3.3 Copying Pixels

The command

```
void CopyPixels(int x, int y, sizei width, sizei height,
                 enum type );
```

transfers a rectangle of pixel values from one region of the read framebuffer to another in the draw framebuffer. Pixel copying is diagrammed in figure 4.3.

type is a symbolic constant that must be one of `COLOR`, `STENCIL`, `DEPTH`, or `DEPTH_STENCIL`, indicating that the values to be transferred are colors, stencil values, depth values, or depth/stencil values, respectively. The first four arguments have the same interpretation as the corresponding arguments to **ReadPixels**.

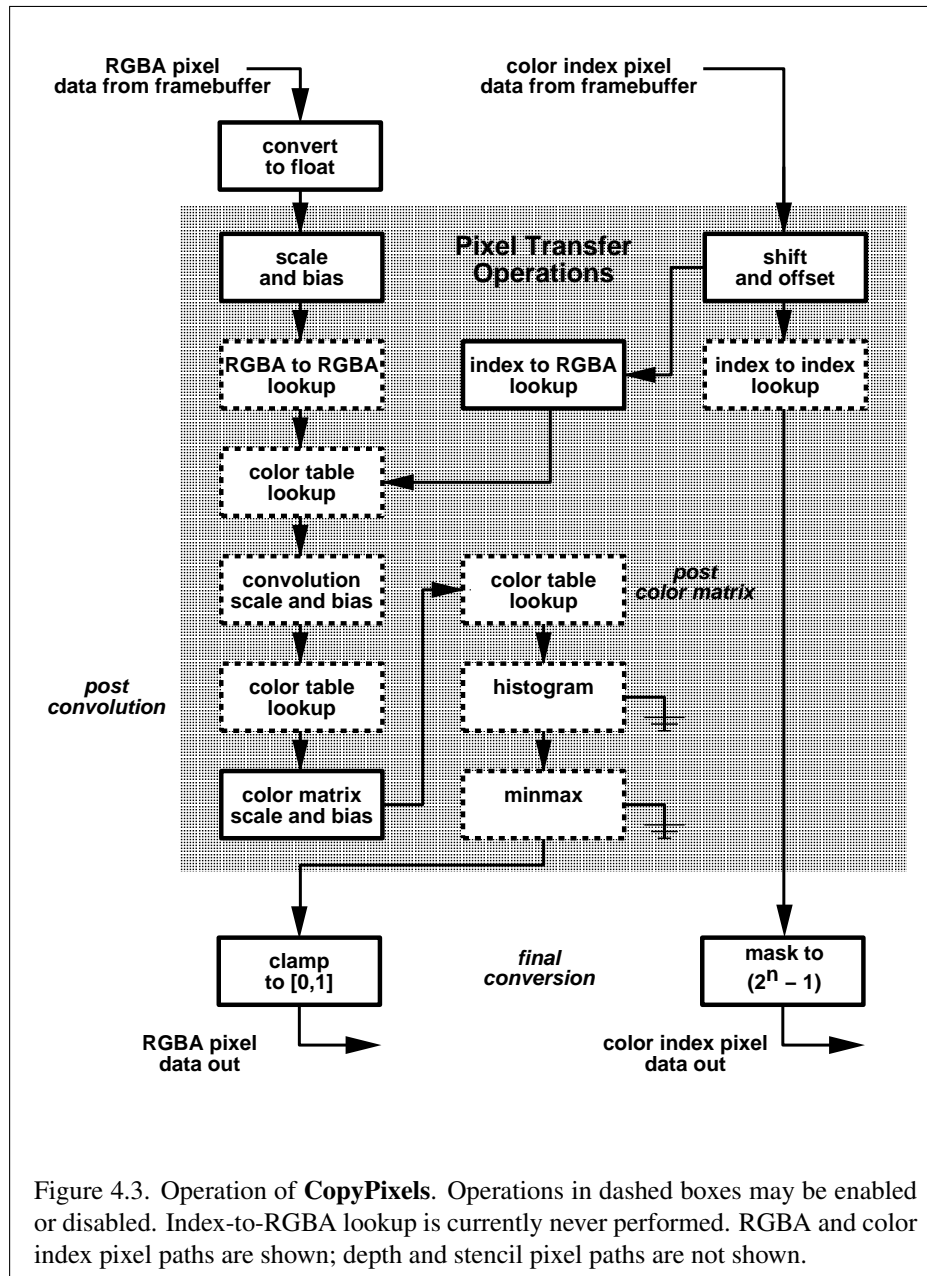
Values are obtained from the framebuffer, converted (if appropriate), then subjected to the pixel transfer operations described in section 3.7.5, just as if **ReadPixels** were called with the corresponding arguments. If the *type* is `STENCIL` or `DEPTH`, then it is as if the *format* for **ReadPixels** were `STENCIL_INDEX` or `DEPTH_COMPONENT`, respectively. If the *type* is `DEPTH_STENCIL`, then it is as if the *format* for **ReadPixels** were specified as described in table 4.10. If the *type* is `COLOR`, then if the GL is in `RGBA` mode, it is as if the *format* were `RGBA`, while if the GL is in color index mode, it is as if the *format* were `COLOR_INDEX`.

The groups of elements so obtained are then written to the framebuffer just as if **DrawPixels** had been given *width* and *height*, beginning with final conversion of elements. The effective *format* is the same as that already described.

Finally, the behavior of several GL operations is specified as if the arguments were passed to **CopyPixels**. These operations include **CopyTexImage***, **CopyTexSubImage***, **CopyColorTable**, **CopyColorSubTable**, and **CopyConvolutionFilter***. An `INVALID_FRAMEBUFFER_OPERATION` error will be generated if an attempt is made to execute one of these operations, or **CopyPixels**, while the object bound to `READ_FRAMEBUFFER_BINDING` (see section 4.4) is not framebuffer complete (as defined in section 4.4.4). An `INVALID_OPERATION` error will be generated if the object bound to `READ_FRAMEBUFFER_BINDING` is framebuffer complete and the value of `SAMPLE_BUFFERS` is greater than zero.

CopyPixels will generate an `INVALID_FRAMEBUFFER_OPERATION` error if the object bound to `DRAW_FRAMEBUFFER_BINDING` (see section 4.4) is not framebuffer complete.

If the read buffer contains integer or unsigned integer components, an `INVALID_OPERATION` error is generated.



DEPTH_BITS	STENCIL_BITS	<i>format</i>
zero	zero	DEPTH_STENCIL
zero	non-zero	DEPTH_COMPONENT
non-zero	zero	STENCIL_INDEX
non-zero	non-zero	DEPTH_STENCIL

Table 4.10: Effective **ReadPixels** format for DEPTH_STENCIL **CopyPixels** operation.

Blitting Pixel Rectangles

The command

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

transfers a rectangle of pixel values from one region of the read framebuffer to another in the draw framebuffer. There are some important distinctions from **CopyPixels**, as described below.

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

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

The actual region taken from the read framebuffer is limited to the intersection of the source buffers being transferred, which may include the color buffer selected by the read buffer, the depth buffer, and/or the stencil buffer depending on *mask*. The actual region written to the draw framebuffer is limited to the intersection of the destination buffers being written, which may include multiple draw buffers, the depth buffer, and/or the stencil buffer depending on *mask*. Whether or not the source or destination regions are altered due to these limits, the scaling and offset applied to pixels being transferred is performed as though no such limits were present.

If the source and destination rectangle dimensions do not match, the source image is stretched to fit the destination rectangle. *filter* must be `LINEAR` or `NEAREST`, and specifies the method of interpolation to be applied if the image is stretched. `LINEAR` filtering is allowed only for the color buffer; if *mask* includes `DEPTH_BUFFER_BIT` or `STENCIL_BUFFER_BIT`, and filter is not `NEAREST`, no copy is performed and an `INVALID_OPERATION` error is generated. If the source and destination dimensions are identical, no filtering is applied. If either the source or destination rectangle specifies a negative width or height ($X1 < X0$ or $Y1 < Y0$), the image is reversed in the corresponding direction. If both the source and destination rectangles specify a negative width or height for the same direction, no reversal is performed. If a linear filter is selected and the rules of `LINEAR` sampling would require sampling outside the bounds of a source buffer, it is as though `CLAMP_TO_EDGE` texture sampling were being performed. If a linear filter is selected and sampling would be required outside the bounds of the specified source region, but within the bounds of a source buffer, the implementation may choose to clamp while sampling or not.

If the source and destination buffers are identical, and the source and destination rectangles overlap, the result of the blit operation is undefined.

Blit operations bypass the fragment pipeline. The only fragment operations which affect a blit are the pixel ownership test and the scissor test.

If a buffer is specified in *mask* and does not exist in both the read and draw framebuffers, the corresponding bit is silently ignored.

If the color formats of the read and draw buffers do not match, and *mask* includes `COLOR_BUFFER_BIT`, pixel groups are converted to match the destination format as in **CopyPixels**. However, no pixel transfer operations are applied, and clamping behaves as if `CLAMP_FRAGMENT_COLOR` is set to `FIXED_ONLY`. Format conversion is not supported for all data types. If the read buffer contains floating-point values and any draw buffer does not contain floating-point values, or if the read buffer contains non-floating-point values and any draw buffer contains floating-point values, an `INVALID_OPERATION` error is generated.

Calling **BlitFramebuffer** will result in an `INVALID_FRAMEBUFFER_OPERATION` error if the objects bound to `DRAW_FRAMEBUFFER_BINDING` and `READ_FRAMEBUFFER_BINDING` are not framebuffer complete (section 4.4.4).

Calling **BlitFramebuffer** will result in an `INVALID_OPERATION` error if *mask* includes `DEPTH_BUFFER_BIT` or `STENCIL_BUFFER_BIT`, and the source and destination depth and stencil buffer formats do not match.

If `SAMPLE_BUFFERS` for the read framebuffer is greater than zero and `SAMPLE_BUFFERS` for the draw framebuffer is zero, the samples corresponding to each pixel location in the source are converted to a single sample before being

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 **Blit-Framebuffer** are not identical, if the formats of the read and draw framebuffers are not identical, or if the values of `SAMPLES` for the read and draw buffers are not identical.

If `SAMPLE_BUFFERS` for both the read and draw framebuffers are greater than zero, and the values of `SAMPLES` for the read and draw framebuffers are identical, the samples are copied without modification from the read framebuffer to the draw framebuffer. Otherwise, no copy is performed and an `INVALID_OPERATION` error is generated. Note that the samples in the draw buffer are not guaranteed to be at the same sample location as the read buffer, so rendering using this newly created buffer can potentially have geometry cracks or incorrect antialiasing. This may occur if the sizes of the framebuffers do not match, if the formats differ, or if the source and destination rectangles are not defined with the same $(X0, Y0)$ and $(X1, Y1)$ bounds.

4.3.4 Pixel Draw/Read State

The state required for pixel operations consists of the parameters that are set with **PixelStore**, **PixelTransfer**, and **PixelMap**. This state has been summarized in tables 3.1, 3.2, and 3.3. Additional state includes the current raster position (section 2.18), an integer indicating the current setting of **ReadBuffer**, and a three-valued integer controlling clamping during Final Conversion. The initial value of the read buffer is `FRONT` if there is no back buffer and `BACK` otherwise, and the initial value of read color clamping is `FIXED_ONLY`. State set with **PixelStore** is GL client state.

4.4 Framebuffer Objects

As described in chapter 1 and section 2.1, the GL renders into (and reads values from) a framebuffer. GL defines two classes of framebuffers: window system-provided and application-created.

Initially, the GL uses the default framebuffer. The storage, dimensions, 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 (accumulation and auxiliary buffers are not allowed). For each logical buffer, a framebuffer-attachable image can be attached to the framebuffer to store the rendered output for that logical buffer. Examples of framebuffer-attachable images include texture images and renderbuffer images. Renderbuffers are described further in section 4.4.2

By allowing the images of a renderbuffer to be attached to a framebuffer, the GL provides a mechanism to support *off-screen* rendering. Further, by allowing the images of a texture to be attached to a framebuffer, the GL provides a mechanism to support *render to texture*.

4.4.1 Binding and Managing Framebuffer Objects

The default framebuffer for rendering and readback operations is provided by the window system. In addition, named framebuffer objects can be created and operated upon. The namespace for framebuffer objects is the unsigned integers, with zero reserved by the GL for the default framebuffer.

A framebuffer object is created by binding a name returned by **GenFramebuffers** (see below) to `DRAW_FRAMEBUFFER` or `READ_FRAMEBUFFER`. The binding is effected by calling

```
void BindFramebuffer( enum target, uint framebuffer );
```

with *target* set to the desired framebuffer target and *framebuffer* set to the framebuffer object name. The resulting framebuffer object is a new state vector, comprising all the state values listed in table 6.28, as well as one set of the state values listed in table 6.29 for each attachment point of the framebuffer, set to the same initial values. There are `MAX_COLOR_ATTACHMENTS` color attachment points, plus one each for the depth and stencil attachment points.

BindFramebuffer may also be used to bind an existing framebuffer object to `DRAW_FRAMEBUFFER` and/or `READ_FRAMEBUFFER`. If the bind is successful no change is made to the state of the bound framebuffer object, and any previous binding to *target* is broken.

BindFramebuffer fails and an `INVALID_OPERATION` error is generated if *framebuffer* is not zero or a name returned from a previous call to **GenFramebuffers**, or if such a name has since been deleted with **DeleteFramebuffers**.

If a framebuffer object is bound to `DRAW_FRAMEBUFFER` or `READ_FRAMEBUFFER`, it becomes the target for rendering or readback operations, respectively, until it is deleted or another framebuffer is bound to the corresponding bind point. Calling **BindFramebuffer** with *target* set to `FRAMEBUFFER` binds *framebuffer* to both the draw and read targets.

While a framebuffer object is bound, GL operations on the target to which it is bound affect the images attached to the bound framebuffer object, and queries of the target to which it is bound return state from the bound object. Queries of the values specified in tables 6.51 and 6.31 are derived from the framebuffer object bound to `DRAW_FRAMEBUFFER`.

The initial state of `DRAW_FRAMEBUFFER` and `READ_FRAMEBUFFER` refers to the default framebuffer. In order that access to the default framebuffer is not lost, it is treated as a framebuffer object with the name of zero. The default framebuffer is therefore rendered to and read from while zero is bound to the corresponding targets. On some implementations, the properties of the default framebuffer can change over time (e.g., in response to window system events such as attaching the context to a new window system drawable.)

Framebuffer objects (those with a non-zero name) differ from the default framebuffer in a few important ways. First and foremost, unlike the default framebuffer, framebuffer objects have modifiable attachment points for each logical buffer in the framebuffer. Framebuffer-attachable images can be attached to and detached from these attachment points, which are described further in section 4.4.2. Also, the size and format of the images attached to framebuffer objects are controlled entirely within the GL interface, and are not affected by window system events, such as pixel format selection, window resizes, and display mode changes.

Additionally, when rendering to or reading from an application created-framebuffer object,

- The pixel ownership test always succeeds. In other words, framebuffer objects own all of their pixels.
- There are no visible color buffer bitplanes. This means there is no color buffer corresponding to the back, front, left, or right color bitplanes.
- The only color buffer bitplanes are the ones defined by the framebuffer attachment points named `COLOR_ATTACHMENT0` through `COLOR_ATTACHMENTn`.

- The only depth buffer bitplanes are the ones defined by the framebuffer attachment point `DEPTH_ATTACHMENT`.
- The only stencil buffer bitplanes are the ones defined by the framebuffer attachment point `STENCIL_ATTACHMENT`.
- There are no accumulation buffer bitplanes, so the value of the implementation-dependent state variables `ACCUM_RED_BITS`, `ACCUM_GREEN_BITS`, `ACCUM_BLUE_BITS`, and `ACCUM_ALPHA_BITS` are all zero.
- There are no `AUX` buffer bitplanes, so the value of the implementation-dependent state variable `AUX_BUFFERS` is zero.
- If the attachment sizes are not all identical, rendering will be limited to the largest area that can fit in all of the attachments (an intersection of rectangles having a lower left of $(0, 0)$ and an upper right of $(width, height)$ for each attachment).
- If the attachment sizes are not all identical, the values of pixels outside the common intersection area after rendering are undefined.

Framebuffer objects are deleted by calling

```
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 **Bind-Framebuffer** 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 6.29

There are two types of framebuffer-attachable images: the image of a renderbuffer object, and an image of a texture object.

Renderbuffer Objects

A renderbuffer is a data storage object containing a single image of a renderable internal format. GL provides the methods described below to allocate and delete a renderbuffer's image, and to attach a renderbuffer's image to a framebuffer object.

The name space for renderbuffer objects is the unsigned integers, with zero reserved for the GL. A renderbuffer object is created by binding a name returned by **GenRenderbuffers** (see below) to `RENDERBUFFER`. The binding is effected by calling

```
void BindRenderbuffer( enum target, uint renderbuffer );
```

with *target* set to `RENDERBUFFER` and *renderbuffer* set to the renderbuffer object name. If *renderbuffer* is not zero, then the resulting renderbuffer object is a new state vector, initialized with a zero-sized memory buffer, and comprising the state values listed in table 6.31. Any previous binding to *target* is broken.

BindRenderbuffer may also be used to bind an existing renderbuffer object. If the bind is successful, no change is made to the state of the newly bound renderbuffer object, and any previous binding to *target* is broken.

While a renderbuffer object is bound, GL operations on the target to which it is bound affect the bound renderbuffer object, and queries of the target to which a renderbuffer object is bound return state from the bound object.

The name zero is reserved. A renderbuffer object cannot be created with the name zero. If *renderbuffer* is zero, then any previous binding to *target* is broken and the *target* binding is restored to the initial state.

In the initial state, the reserved name zero is bound to `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 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

```
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. If the GL

Sized Internal Format	Base Internal Format	<i>S</i> bits
STENCIL_INDEX1	STENCIL_INDEX	1
STENCIL_INDEX4	STENCIL_INDEX	4
STENCIL_INDEX8	STENCIL_INDEX	8
STENCIL_INDEX16	STENCIL_INDEX	16

Table 4.11: Correspondence of sized internal formats to base internal formats for formats that can be used only with renderbuffers.

is unable to create a data store of the requested size, the error `OUT_OF_MEMORY` is generated.

Upon success, **RenderbufferStorageMultisample** deletes any existing data store for the renderbuffer image and the contents of the data store after calling **RenderbufferStorageMultisample** are undefined. `RENDERBUFFER_WIDTH` is set to *width*, `RENDERBUFFER_HEIGHT` is set to *height*, and `RENDERBUFFER_INTERNAL_FORMAT` is set to *internalformat*.

If *samples* is zero, then `RENDERBUFFER_SAMPLES` is set to zero. Otherwise *samples* represents a request for a desired minimum number of samples. Since different implementations may support different sample counts for multisampled rendering, the actual number of samples allocated for the renderbuffer image is implementation dependent. However, the resulting value for `RENDERBUFFER_SAMPLES` is guaranteed to be greater than or equal to *samples* and no more than the next larger sample count supported by the implementation.

A GL implementation may vary its allocation of internal component resolution based on any **RenderbufferStorage** parameter (except target), but the allocation and chosen internal format must not be a function of any other state and cannot be changed once they are established.

The command

```
void RenderbufferStorage(enum target, enum internalformat,
                        sizei width, sizei height);
```

is equivalent to calling **RenderbufferStorageMultisample** with *samples* equal to zero.

Required Renderbuffer Formats

Implementations are required to support the same internal formats for renderbuffers as the required formats for textures enumerated in section 3.9.1, with the exception

of the color formats labelled “texture-only”. Requesting one of these internal formats for a renderbuffer will allocate exactly the internal component sizes and types shown for that format in tables 3.16- 3.18.

Implementations must support creation of renderbuffers in these required formats with up to the value of `MAX_SAMPLES` multisamples.

Attaching Renderbuffer Images to a Framebuffer

A renderbuffer can be attached as one of the logical buffers of the currently bound framebuffer object by calling

```
void FramebufferRenderbuffer( enum target,
                             enum attachment, enum renderbuffertarget,
                             uint renderbuffer );
```

target must be `DRAW_FRAMEBUFFER`, `READ_FRAMEBUFFER`, or `FRAMEBUFFER`. `FRAMEBUFFER` is equivalent to `DRAW_FRAMEBUFFER`. An `INVALID_OPERATION` error is generated if the value of the corresponding binding is zero. *attachment* should be set to one of the attachment points of the framebuffer listed in table 4.12.

renderbuffertarget must be `RENDERBUFFER` and *renderbuffer* should be set to the name of the renderbuffer object to be attached to the framebuffer. *renderbuffer* must be either zero or the name of an existing renderbuffer object of type *renderbuffertarget*, otherwise an `INVALID_OPERATION` error is generated. If *renderbuffer* is zero, then the value of *renderbuffertarget* is ignored.

If *renderbuffer* is not zero and if **FramebufferRenderbuffer** is successful, then the renderbuffer named *renderbuffer* will be used as the logical buffer identified by *attachment* of the framebuffer currently bound to *target*. The value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` for the specified attachment point is set to `RENDERBUFFER` and the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` is set to *renderbuffer*. All other state values of the attachment point specified by *attachment* are set to their default values listed in table 6.29. 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 6.29.

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
<code>COLOR_ATTACHMENT<i>i</i></code> (see caption)
<code>DEPTH_ATTACHMENT</code>
<code>STENCIL_ATTACHMENT</code>
<code>DEPTH_STENCIL_ATTACHMENT</code>

Table 4.12: Framebuffer attachment points. *i* in `COLOR_ATTACHMENTi` 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 4.12.

If *texture* is zero, the image identified by *attachment*, if any, will be detached from the framebuffer currently bound to *target*. *textarget*, *level*, and *layer* are ignored. All state values of the attachment point specified by *attachment* are set to their default values listed in table 6.29.

If *texture* is not zero, then *texture* must either name an existing texture object with a target of *textarget*, or *texture* must name an existing cube map texture and *textarget* must be one of `TEXTURE_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, or `TEXTURE_CUBE_MAP_NEGATIVE_Z`. Otherwise, an `INVALID_OPERATION` error is generated.

level specifies the mipmap level of the texture image to be attached to the framebuffer.

If *textarget* is `TEXTURE_3D`, then *level* must be greater than or equal to zero and less than or equal to \log_2 of the value of `MAX_3D_TEXTURE_SIZE`. If *textarget* is one of `TEXTURE_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, or `TEXTURE_CUBE_MAP_NEGATIVE_Z`, then *level* must be greater than or equal to zero and less than or equal to \log_2 of the value of `MAX_CUBE_MAP_TEXTURE_SIZE`. For all other values of *textarget*, *level* must be greater than or equal to zero and no larger than \log_2 of the value of `MAX_TEXTURE_SIZE`. Otherwise, an `INVALID_VALUE` error is generated.

layer specifies the layer of a 2-dimensional image within a 3-dimensional 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_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, or `TEXTURE_CUBE_MAP_NEGATIVE_Z`.

For **FramebufferTexture3D**, if *texture* is not zero, then *textarget* must be `TEXTURE_3D`.

If *texture* is not zero, and if **FramebufferTexture*** is successful, then the specified texture image will be used as the logical buffer identified by *attachment* of the framebuffer currently bound to *tar-*

get. The value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` for the specified attachment point is set to `TEXTURE` and the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` is set to *texture*. Additionally, the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL` for the named attachment point is set to *level*. If *texture* is a cube map texture, then the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE` for the named attachment point is set to *textarget*. If *texture* is a 3D texture, then the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER` for the named attachment point is set to *layer*. All other state values of the attachment point specified by *attachment* are set to their default values listed in table 6.29. No change is made to the state of the texture object, and any previous attachment to the *attachment* logical buffer of the framebuffer object bound to framebuffer *target* is broken. If the attachment is not successful, then no change is made to the state of either the texture object or the framebuffer object.

Setting *attachment* to the value `DEPTH_STENCIL_ATTACHMENT` is a special case causing both the depth and stencil attachments of the framebuffer object to be set to *texture*. *texture* must have base internal format `DEPTH_STENCIL`, or the depth and stencil framebuffer attachments will be incomplete (see section 4.4.4).

The command

```
void FramebufferTextureLayer( enum target,
                               enum attachment, uint texture, int level, int layer );
```

operates identically to **FramebufferTexture3D**, except that it attaches a single layer of a three-dimensional texture or a one- or two-dimensional array texture. *layer* is an integer indicating the layer number, and is treated identically to the *layer* parameter in **FramebufferTexture3D**. The error `INVALID_VALUE` is generated if *layer* is negative. The error `INVALID_OPERATION` is generated if *texture* is non-zero and is not the name of a three dimensional texture or one- or two-dimensional array texture. Unlike **FramebufferTexture3D**, no *textarget* parameter is accepted.

If *texture* is non-zero and the command does not result in an error, the framebuffer attachment state corresponding to *attachment* is updated as in the other **FramebufferTexture** commands, except that `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER` is set to *layer*.

If a texture object is deleted while its image is attached to one or more attachment points in the currently bound framebuffer, then it is as if **FramebufferTexture*** had been called, with a *texture* of zero, for each attachment point to which this image was attached in the currently bound framebuffer. In other words, this texture image is first detached from all attachment points in the currently bound

framebuffer. Note that the texture image is specifically **not** detached from any other framebuffer objects. Detaching the texture image from any other framebuffer objects is the responsibility of the application.

4.4.3 Rendering When an Image of a Bound Texture Object is Also Attached to the Framebuffer

The mechanisms for attaching textures to a framebuffer object do not prevent a one- or two-dimensional texture level, a face of a cube map texture level, or a layer of a two-dimensional array or three-dimensional texture from being attached to the draw framebuffer while the same texture is bound to a texture unit. While any of these conditions hold, texturing operations accessing that image will produce undefined results, as described at the end of section 3.9.7. Conditions resulting in such undefined behavior are defined in more detail below. Such undefined texturing operations are likely to leave the final results of the shader or fixed-function fragment processing operations undefined, and should be avoided.

Special precautions need to be taken to avoid attaching a texture image to the currently bound framebuffer while the texture object is currently bound and enabled for texturing. Doing so could lead to the creation of a *feedback loop* between the writing of pixels by the GL's rendering operations and the simultaneous reading of those same pixels when used as texels in the currently bound texture. In this scenario, the framebuffer will be considered framebuffer complete (see section 4.4.4), but the values of fragments rendered while in this state will be undefined. The values of texture samples may be undefined as well, as described at the end of the **Scale Factor and Level of Detail** subsection of section 3.9.7.

Specifically, the values of rendered fragments are undefined if all of the following conditions are true:

- an image from texture object *T* is attached to the currently bound framebuffer at attachment point *A*
- the texture object *T* is currently bound to a texture unit *U*, and
- the current fixed-function texture state or programmable vertex and/or fragment processing state makes it possible (see below) to sample from the texture object *T* bound to texture unit *U*

while either of the following conditions are true:

- the value of `TEXTURE_MIN_FILTER` for texture object *T* is `NEAREST` or `LINEAR`, and the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL`

for attachment point A is equal to the value of `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 range specified by the current values of `TEXTURE_BASE_LEVEL` to q , inclusive, for the texture object T . (q is defined in the **Mipmapping** discussion of section 3.9.7).

For the purpose of this discussion, it is *possible* to sample from the texture object T bound to texture unit U if any of the following are true:

- Programmable fragment processing is disabled and the target of texture object T is enabled according to the texture target precedence rules of section 3.9.17
- The active fragment or vertex shader contains any instructions that might sample from the texture object T bound to U , even if those instructions might only be executed conditionally.

Note that if `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.)

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 3.15 are *color-renderable*: ALPHA, RED, RG, RGB, and RGBA. The sized internal formats from table 3.16 that have a color-renderable base internal format are also color-renderable. No other formats, including compressed internal formats, are color-renderable.
- An internal format is *depth-renderable* if it is DEPTH_COMPONENT or one of the formats from table 3.18 whose base internal format is DEPTH_COMPONENT or DEPTH_STENCIL. No other formats are depth-renderable.
- An internal format is *stencil-renderable* if it is STENCIL_INDEX or DEPTH_STENCIL, if it is one of the STENCIL_INDEX formats from table 4.11, or if it is one of the formats from table 3.18 whose base internal format is DEPTH_STENCIL. No other formats are stencil-renderable.

Framebuffer Attachment Completeness

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE for the framebuffer attachment point *attachment* is not NONE, then it is said that a framebuffer-attachable image, named *image*, is attached to the framebuffer at the attachment point. *image* is identified by the state in *attachment* as described in section 4.4.2.

The framebuffer attachment point *attachment* is said to be *framebuffer attachment complete* if the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE for *attachment* is NONE (i.e., no image is attached), or if all of the following conditions are true:

- *image* is a component of an existing object with the name specified by FRAMEBUFFER_ATTACHMENT_OBJECT_NAME, and of the type specified by FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE.
- The width and height of *image* are non-zero.
- If FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is TEXTURE and FRAMEBUFFER_ATTACHMENT_OBJECT_NAME names a three-dimensional texture, then FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER must be smaller than the depth of the texture.
- If FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is TEXTURE and FRAMEBUFFER_ATTACHMENT_OBJECT_NAME names a one- or two-dimensional array texture, then FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER must be smaller than the number of layers in the texture.

- If *attachment* is `COLOR_ATTACHMENTi`, then *image* must have a color-renderable internal format.
- If *attachment* is `DEPTH_ATTACHMENT`, then *image* must have a depth-renderable internal format.
- If *attachment* is `STENCIL_ATTACHMENT`, then *image* must have a stencil-renderable internal format.

Whole Framebuffer Completeness

Each rule below is followed by an error token enclosed in { brackets }. The meaning of these errors is explained below and under “Effects of Framebuffer Completeness on Framebuffer Operations” later in section 4.4.4.

The framebuffer object *target* is said to be *framebuffer complete* if all the following conditions are true:

- *target* is the default framebuffer, and the default framebuffer exists.

{ FRAMEBUFFER_UNDEFINED }

- All framebuffer attachment points are *framebuffer attachment complete*.

{ FRAMEBUFFER_INCOMPLETE_ATTACHMENT }

- There is at least one image attached to the framebuffer.

{ FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT }

- The value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` must not be `NONE` for any color attachment point(s) named by `DRAW_BUFFERi`.

{ FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER }

- If `READ_BUFFER` is not `NONE`, then the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` must not be `NONE` for the color attachment point named by `READ_BUFFER`.

{ FRAMEBUFFER_INCOMPLETE_READ_BUFFER }

- The combination of internal formats of the attached images does not violate an implementation-dependent set of restrictions.

{ FRAMEBUFFER_UNSUPPORTED }

- The value of `RENDERBUFFER_SAMPLES` is the same for all attached renderbuffers; and, if the attached images are a mix of renderbuffers and textures, the value of `RENDERBUFFER_SAMPLES` is zero for all attached renderbuffers.

{ `FRAMEBUFFER_INCOMPLETE_MULTISAMPLE` }

The token in brackets after each clause of the framebuffer completeness rules specifies the return value of **CheckFramebufferStatus** (see below) that is generated when that clause is violated. If more than one clause is violated, it is implementation-dependent which value will be returned by **CheckFramebufferStatus**.

Performing any of the following actions may change whether the framebuffer is considered complete or incomplete:

- Binding to a different framebuffer with **BindFramebuffer**.
- Attaching an image to the framebuffer with **FramebufferTexture*** or **FramebufferRenderbuffer**.
- Detaching an image from the framebuffer with **FramebufferTexture*** or **FramebufferRenderbuffer**.
- Changing the internal format of a texture image that is attached to the framebuffer by calling **CopyTexImage*** or **CompressedTexImage***.
- Changing the internal format of a renderbuffer that is attached to the framebuffer by calling **RenderbufferStorage**.
- Deleting, with **DeleteTextures** or **DeleteRenderbuffers**, an object containing an image that is attached to a framebuffer object that is bound to the framebuffer.
- Changing the read buffer or one of the draw buffers.
- Associating a different window system-provided drawable, or no drawable, with the default framebuffer using a window system binding API such as those described in section 1.7.2.

Although the GL defines a wide variety of internal formats for framebuffer-attachable images, such as texture images and renderbuffer images, some implementations may not support rendering to particular combinations of internal formats. If the combination of formats of the images attached to a framebuffer object

are not supported by the implementation, then the framebuffer is not complete under the clause labeled `FRAMEBUFFER_UNSUPPORTED`.

Implementations are required to support certain combinations of framebuffer internal formats as described under “Required Framebuffer Formats” in section 4.4.4.

Because of the *implementation-dependent* clause of the framebuffer completeness test in particular, and because framebuffer completeness can change when the set of attached images is modified, it is strongly advised, though not required, that an application check to see if the framebuffer is complete prior to rendering. The status of the framebuffer object currently bound to *target* can be queried by calling

```
enum CheckFramebufferStatus( enum target );
```

target must be `DRAW_FRAMEBUFFER`, `READ_FRAMEBUFFER`, or `FRAMEBUFFER`. `FRAMEBUFFER` is equivalent to `DRAW_FRAMEBUFFER`. If **CheckFramebufferStatus** is called within a **Begin/End** pair, an `INVALID_OPERATION` error is generated. If **CheckFramebufferStatus** generates an error, zero is returned.

Otherwise, a value is returned that identifies whether or not the framebuffer bound to *target* is complete, and if not complete the value identifies one of the rules of framebuffer completeness that is violated. If the framebuffer is complete, then `FRAMEBUFFER_COMPLETE` is returned.

The values of `SAMPLE_BUFFERS` and `SAMPLES` are derived from the attachments of the currently bound framebuffer object. If the current `DRAW_FRAMEBUFFER_BINDING` is not framebuffer complete, then both `SAMPLE_BUFFERS` and `SAMPLES` are undefined. Otherwise, `SAMPLES` is equal to the value of `RENDERBUFFER_SAMPLES` for the attached images (which all must have the same value for `RENDERBUFFER_SAMPLES`). Further, `SAMPLE_BUFFERS` is one if `SAMPLES` is non-zero. Otherwise, `SAMPLE_BUFFERS` is zero.

Required Framebuffer Formats

Implementations must support framebuffer objects with up to `MAX_COLOR_ATTACHMENTS` color attachments, a depth attachment, and a stencil attachment. Each color attachment may be in any of the required color formats for textures and renderbuffers described in sections 3.9.1 and 4.4.2. The depth attachment may be in any of the required depth or combined depth+stencil formats described in those sections, and the stencil attachment may be in any of the required combined depth+stencil formats.

There must be at least one default framebuffer format allowing creation of a default framebuffer supporting front-buffered rendering.

Effects of Framebuffer Completeness on Framebuffer Operations

Attempting to render to or read from a framebuffer which is not framebuffer complete will generate an `INVALID_FRAMEBUFFER_OPERATION` error. This means that rendering commands such as **Begin**, **RasterPos**, any command that performs an implicit **Begin**, as well as commands that read the framebuffer such as **ReadPixels**, **CopyTexImage**, and **CopyTexSubImage**, will generate the error `INVALID_FRAMEBUFFER_OPERATION` if called while the framebuffer is not framebuffer complete.

4.4.5 Effects of Framebuffer State on Framebuffer Dependent Values

The values of the state variables listed in table 6.51 may change when a change is made to `DRAW_FRAMEBUFFER_BINDING`, to the state of the currently bound 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 6.51 are implementation defined.

When `DRAW_FRAMEBUFFER_BINDING` is non-zero, if the currently bound framebuffer object is not framebuffer complete, then the values of the state variables listed in table 6.51 are undefined.

When `DRAW_FRAMEBUFFER_BINDING` is non-zero and the currently bound framebuffer object is framebuffer complete, then the values of the state variables listed in table 6.51 are completely determined by `DRAW_FRAMEBUFFER_BINDING`, the state of the currently bound framebuffer object, and the state of the images attached to the currently bound framebuffer object. The values of `RED_BITS`, `GREEN_BITS`, `BLUE_BITS`, and `ALPHA_BITS` are defined only if all color attachments of the draw framebuffer have identical formats, in which case the color component depths of color attachment zero are returned. The values returned for `DEPTH_BITS` and `STENCIL_BITS` are the depth or stencil component depth of the corresponding attachment of the draw framebuffer, respectively. The actual sizes of the color, depth, or stencil bit planes can be obtained by querying an attachment point using **GetFramebufferAttachmentParameteriv**, or querying the object attached to that point. If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` at a particular attachment point is `RENDERBUFFER`, the sizes may be determined by calling **GetRenderbufferParameteriv** as described in section 6.1.3. If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` at a particular attachment point is `TEXTURE`, the sizes may be determined by calling **GetTexParameter**, as described in section 6.1.3.

4.4.6 Mapping between Pixel and Element in Attached Image

When `DRAW_FRAMEBUFFER_BINDING` is non-zero, an operation that writes to the framebuffer modifies the image attached to the selected logical buffer, and an operation that reads from the framebuffer reads from the image attached to the selected logical buffer.

If the attached image is a renderbuffer image, then the window coordinates (x_w, y_w) corresponds to the value in the renderbuffer image at the same coordinates.

If the attached image is a texture image, then the window coordinates (x_w, y_w) correspond to the texel (i, j, k) from figure 3.10 as follows:

$$i = (x_w - b)$$

$$j = (y_w - b)$$

$$k = (layer - b)$$

where b is the texture image's border width and $layer$ is the value of `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER` for the selected logical buffer. For a two-dimensional texture, k and $layer$ are irrelevant; for a one-dimensional texture, j , k , and $layer$ are irrelevant.

(x_w, y_w) corresponds to a border texel if x_w , y_w , or $layer$ is less than the border width, or if x_w , y_w , or $layer$ is greater than or equal to the border width plus the width, height, or depth, respectively, of the texture image.

Conversion to Framebuffer-Attachable Image Components

When an enabled color value is written to the framebuffer while the draw framebuffer binding is non-zero, for each draw buffer the R, G, B, and A values are converted to internal components as described in table 3.15, according to the table row corresponding to the internal format of the framebuffer-attachable image attached to the selected logical buffer, and the resulting internal components are written to the image attached to logical buffer. The masking operations described in section 4.2.2 are also effective.

Conversion to RGBA Values

When a color value is read or is used as the source of a logical operation or blending while the read framebuffer binding is non-zero, the components of the framebuffer-attachable image that is attached to the logical buffer selected by `READ_BUFFER` are first converted to R, G, B, and A values according to table 3.23 and the internal format of the attached image.

Chapter 5

Special Functions

This chapter describes additional GL functionality that does not fit easily into any of the preceding chapters. This functionality consists of evaluators (used to model curves and surfaces), selection (used to locate rendered primitives on the screen), feedback (which returns GL results before rasterization), display lists (used to designate a group of GL commands for later execution by the GL), flushing and finishing (used to synchronize the GL command stream), and hints.

5.1 Evaluators

Evaluators provide a means to use a polynomial or rational polynomial mapping to produce vertex, normal, and texture coordinates, and colors. The values so produced are sent on to further stages of the GL as if they had been provided directly by the client. Transformations, lighting, primitive assembly, rasterization, and per-pixel operations are not affected by the use of evaluators.

Consider the R^k -valued polynomial $\mathbf{p}(u)$ defined by

$$\mathbf{p}(u) = \sum_{i=0}^n B_i^n(u) \mathbf{R}_i \quad (5.1)$$

with $\mathbf{R}_i \in R^k$ and

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i},$$

the i th Bernstein polynomial of degree n (recall that $0^0 \equiv 1$ and $\binom{n}{0} \equiv 1$). Each \mathbf{R}_i is a *control point*. The relevant command is

```
void Map1{fd}(enum target, T u1, T u2, int stride,
               int order, T points );
```


<i>target</i>	<i>k</i>	Values
MAP1_VERTEX_3	3	x, y, z vertex coordinates
MAP1_VERTEX_4	4	x, y, z, w vertex coordinates
MAP1_INDEX	1	color index
MAP1_COLOR_4	4	R, G, B, A
MAP1_NORMAL	3	x, y, z normal coordinates
MAP1_TEXTURE_COORD_1	1	s texture coordinate
MAP1_TEXTURE_COORD_2	2	s, t texture coordinates
MAP1_TEXTURE_COORD_3	3	s, t, r texture coordinates
MAP1_TEXTURE_COORD_4	4	s, t, r, q texture coordinates

Table 5.1: Values specified by the *target* to **Map1**. Values are given in the order in which they are taken.

target is a symbolic constant indicating the range of the defined polynomial. Its possible values, along with the evaluations that each indicates, are given in table 5.1. *order* is equal to $n + 1$; The error `INVALID_VALUE` is generated if *order* is less than one or greater than `MAX_EVAL_ORDER`. *points* is a pointer to a set of $n + 1$ blocks of storage. Each block begins with k single-precision floating-point or double-precision floating-point values, respectively. The rest of the block may be filled with arbitrary data. Table 5.1 indicates how k depends on *target* and what the k values represent in each case.

stride is the number of single- or double-precision values (as appropriate) in each block of storage. The error `INVALID_VALUE` results if *stride* is less than k . The order of the polynomial, *order*, is also the number of blocks of storage containing control points.

u_1 and u_2 give two floating-point values that define the endpoints of the pre-image of the map. When a value u' is presented for evaluation, the formula used is

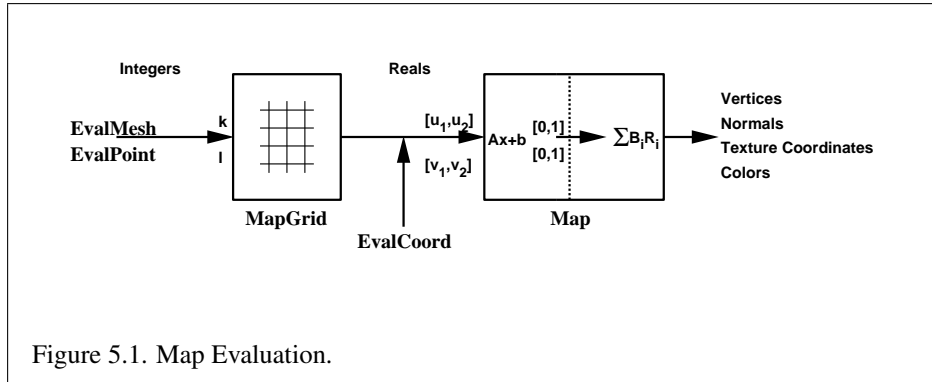
$$\mathbf{p}'(u') = \mathbf{p}\left(\frac{u' - u_1}{u_2 - u_1}\right).$$

The error `INVALID_VALUE` results if $u_1 = u_2$.

Map2 is analogous to **Map1**, except that it describes bivariate polynomials of the form

$$\mathbf{p}(u, v) = \sum_{i=0}^n \sum_{j=0}^m B_i^n(u) B_j^m(v) \mathbf{R}_{ij}.$$

The form of the **Map2** command is



```
void Map2{fd}(enum target, T u1, T u2, int ustride,
               int uorder, T v1, T v2, int vstride, int vorder, T points);
```

target is a range type selected from the same group as is used for **Map1**, except that the string MAP1 is replaced with MAP2. *points* is a pointer to $(n + 1)(m + 1)$ blocks of storage ($uorder = n + 1$ and $vorder = m + 1$; the error INVALID_VALUE is generated if either *uorder* or *vorder* is less than one or greater than MAX_EVAL_ORDER). The values comprising R_{ij} are located

$$(ustride)i + (vstride)j$$

values (either single- or double-precision floating-point, as appropriate) past the first value pointed to by *points*. u_1 , u_2 , v_1 , and v_2 define the pre-image rectangle of the map; a domain point (u', v') is evaluated as

$$\mathbf{p}'(u', v') = \mathbf{p}\left(\frac{u' - u_1}{u_2 - u_1}, \frac{v' - v_1}{v_2 - v_1}\right).$$

The evaluation of a defined map is enabled or disabled with **Enable** and **Disable** using the constant corresponding to the map as described above. The evaluator map generates only coordinates for texture unit TEXTURE0. The error INVALID_VALUE results if either *ustride* or *vstride* is less than k , or if u_1 is equal to u_2 , or if v_1 is equal to v_2 . If the value of ACTIVE_TEXTURE is not TEXTURE0, calling **Map**{12} generates the error INVALID_OPERATION.

Figure 5.1 describes map evaluation schematically; an evaluation of enabled maps is effected in one of two ways. The first way is to use

```
void EvalCoord{12}{fd}(T arg);
void EvalCoord{12}{fd}v(T arg);
```

EvalCoord1 causes evaluation of the enabled one-dimensional maps. The argument is the value (or a pointer to the value) that is the domain coordinate, u' . **EvalCoord2** causes evaluation of the enabled two-dimensional maps. The two values specify the two domain coordinates, u' and v' , in that order.

When one of the **EvalCoord** commands is issued, all currently enabled maps of the indicated dimension are evaluated. Then, for each enabled map, it is as if a corresponding GL command were issued with the resulting coordinates, with one important difference. The difference is that when an evaluation is performed, the GL uses evaluated values instead of current values for those evaluations that are enabled (otherwise, the current values are used). The order of the effective commands is immaterial, except that **Vertex** (for vertex coordinate evaluation) must be issued last. Use of evaluators has no effect on the current color, normal, or texture coordinates. If **ColorMaterial** is enabled, evaluated color values affect the result of the lighting equation as if the current color was being modified, but no change is made to the tracking lighting parameters or to the current color.

No command is effectively issued if the corresponding map (of the indicated dimension) is not enabled. If more than one evaluation is enabled for a particular dimension (e.g. `MAP1_TEXTURE_COORD_1` and `MAP1_TEXTURE_COORD_2`), then only the result of the evaluation of the map with the highest number of coordinates is used.

Finally, if either `MAP2_VERTEX_3` or `MAP2_VERTEX_4` is enabled, then the normal to the surface is computed. Analytic computation, which sometimes yields normals of length zero, is one method which may be used. If automatic normal generation is enabled, then this computed normal is used as the normal associated with a generated vertex. Automatic normal generation is controlled with **Enable** and **Disable** with the symbolic constant `AUTO_NORMAL`. If automatic normal generation is disabled, then a corresponding normal map, if enabled, is used to produce a normal. If neither automatic normal generation nor a normal map are enabled, then no normal is sent with a vertex resulting from an evaluation (the effect is that the current normal is used).

For `MAP_VERTEX_3`, let $\mathbf{q} = \mathbf{p}$. For `MAP_VERTEX_4`, let $\mathbf{q} = (x/w, y/w, z/w)$, where $(x, y, z, w) = \mathbf{p}$. Then let

$$\mathbf{m} = \frac{\partial \mathbf{q}}{\partial u} \times \frac{\partial \mathbf{q}}{\partial v}.$$

Then the generated analytic normal, \mathbf{n} , is given by $\mathbf{n} = \mathbf{m}$ if a vertex shader is active, or else by $\mathbf{n} = \frac{\mathbf{m}}{\|\mathbf{m}\|}$.

The second way to carry out evaluations is to use a set of commands that provide for efficient specification of a series of evenly spaced values to be mapped. This method proceeds in two steps. The first step is to define a grid in the domain.

This is done using

```
void MapGrid1{fd}( int  $n$ , T  $u'_1$ , T  $u'_2$  );
```

for a one-dimensional map or

```
void MapGrid2{fd}( int  $n_u$ , T  $u'_1$ , T  $u'_2$ , int  $n_v$ , T  $v'_1$ ,  
T  $v'_2$  );
```

for a two-dimensional map. In the case of **MapGrid1** u'_1 and u'_2 describe an interval, while n describes the number of partitions of the interval. The error `INVALID_VALUE` results if $n \leq 0$. For **MapGrid2**, (u'_1, v'_1) specifies one two-dimensional point and (u'_2, v'_2) specifies another. n_u gives the number of partitions between u'_1 and u'_2 , and n_v gives the number of partitions between v'_1 and v'_2 . If either $n_u \leq 0$ or $n_v \leq 0$, then the error `INVALID_VALUE` occurs.

Once a grid is defined, an evaluation on a rectangular subset of that grid may be carried out by calling

```
void EvalMesh1( enum  $mode$ , int  $p_1$ , int  $p_2$  );
```

$mode$ is either `POINT` or `LINE`. The effect is the same as performing the following code fragment, with $\Delta u' = (u'_2 - u'_1)/n$:

```
Begin ( $type$ ) ;  
  for  $i = p_1$  to  $p_2$  step 1.0  
    EvalCoord1 ( $i * \Delta u' + u'_1$ ) ;  
End () ;
```

where **EvalCoord1f** or **EvalCoord1d** is substituted for **EvalCoord1** as appropriate. If $mode$ is `POINT`, then $type$ is `POINTS`; if $mode$ is `LINE`, then $type$ is `LINE_STRIP`. The one requirement is that if either $i = 0$ or $i = n$, then the value computed from $i * \Delta u' + u'_1$ is precisely u'_1 or u'_2 , respectively.

The corresponding commands for two-dimensional maps are

```
void EvalMesh2( enum  $mode$ , int  $p_1$ , int  $p_2$ , int  $q_1$ ,  
int  $q_2$  );
```

$mode$ must be `FILL`, `LINE`, or `POINT`. When $mode$ is `FILL`, then these commands are equivalent to the following, with $\Delta u' = (u'_2 - u'_1)/n$ and $\Delta v' = (v'_2 - v'_1)/m$:

```

for  $i = q_1$  to  $q_2 - 1$  step 1.0
  Begin (QUAD_STRIP) ;
  for  $j = p_1$  to  $p_2$  step 1.0
    EvalCoord2 ( $j * \Delta u' + u'_1$  ,  $i * \Delta v' + v'_1$ ) ;
    EvalCoord2 ( $j * \Delta u' + u'_1$  ,  $(i + 1) * \Delta v' + v'_1$ ) ;
  End () ;

```

If *mode* is `LINE`, then a call to **EvalMesh2** is equivalent to

```

for  $i = q_1$  to  $q_2$  step 1.0
  Begin (LINE_STRIP) ;
  for  $j = p_1$  to  $p_2$  step 1.0
    EvalCoord2 ( $j * \Delta u' + u'_1$  ,  $i * \Delta v' + v'_1$ ) ;
  End () ;
for  $i = p_1$  to  $p_2$  step 1.0
  Begin (LINE_STRIP) ;
  for  $j = q_1$  to  $q_2$  step 1.0
    EvalCoord2 ( $i * \Delta u' + u'_1$  ,  $j * \Delta v' + v'_1$ ) ;
  End () ;

```

If *mode* is `POINT`, then a call to **EvalMesh2** is equivalent to

```

Begin (POINTS) ;
  for  $i = q_1$  to  $q_2$  step 1.0
    for  $j = p_1$  to  $p_2$  step 1.0
      EvalCoord2 ( $j * \Delta u' + u'_1$  ,  $i * \Delta v' + v'_1$ ) ;
  End () ;

```

Again, in all three cases, there is the requirement that $0 * \Delta u' + u'_1 = u'_1$, $n * \Delta u' + u'_1 = u'_2$, $0 * \Delta v' + v'_1 = v'_1$, and $m * \Delta v' + v'_1 = v'_2$.

An evaluation of a single point on the grid may also be carried out:

```
void EvalPoint1(int  $p$ );
```

Calling it is equivalent to the command

```
EvalCoord1 ( $p * \Delta u' + u'_1$ ) ;
```

with $\Delta u'$ and u'_1 defined as above.

```
void EvalPoint2(int  $p$ , int  $q$ );
```

is equivalent to the command

EvalCoord2 ($p * \Delta u' + u'_1$, $q * \Delta v' + v'_1$) ;

The state required for evaluators potentially consists of 9 one-dimensional map specifications and 9 two-dimensional map specifications, as well as corresponding flags for each specification indicating which are enabled. Each map specification consists of one or two orders, an appropriately sized array of control points, and a set of two values (for a one-dimensional map) or four values (for a two-dimensional map) to describe the domain. The maximum possible order, for either u or v , is implementation dependent (one maximum applies to both u and v), but must be at least 8. Each control point consists of between one and four floating-point values (depending on the type of the map). Initially, all maps have order 1 (making them constant maps). All vertex coordinate maps produce the coordinates (0, 0, 0, 1) (or the appropriate subset); all normal coordinate maps produce (0, 0, 1); RGBA maps produce (1, 1, 1, 1); color index maps produce 1.0; and texture coordinate maps produce (0, 0, 0, 1). In the initial state, all maps are disabled. A flag indicates whether or not automatic normal generation is enabled for two-dimensional maps. In the initial state, automatic normal generation is disabled. Also required are two floating-point values and an integer number of grid divisions for the one-dimensional grid specification and four floating-point values and two integer grid divisions for the two-dimensional grid specification. In the initial state, the bounds of the domain interval for 1-D is 0 and 1.0, respectively; for 2-D, they are (0, 0) and (1.0, 1.0), respectively. The number of grid divisions is 1 for 1-D and 1 in both directions for 2-D. If any evaluation command is issued when no vertex map is enabled for the map dimension being evaluated, nothing happens.

5.2 Selection

Selection is used to determine which primitives are drawn into some region of a window. The region is defined by the current model-view and perspective matrices.

Selection works by returning an array of integer-valued *names*. This array represents the current contents of the *name stack*. This stack is controlled with the commands

```
void InitNames( void );
void PopName( void );
void PushName( uint name );
void LoadName( uint name );
```

InitNames empties (clears) the name stack. **PopName** pops one name off the top of the name stack. **PushName** causes *name* to be pushed onto the name stack.

LoadName replaces the value on the top of the stack with *name*. Loading a name onto an empty stack generates the error `INVALID_OPERATION`. Popping a name off of an empty stack generates `STACK_UNDERFLOW`; pushing a name onto a full stack generates `STACK_OVERFLOW`. The maximum allowable depth of the name stack is implementation dependent but must be at least 64.

In selection mode, framebuffer updates as described in chapter 4 are not performed. The GL is placed in selection mode with

```
int RenderMode( enum mode );
```

mode is a symbolic constant: one of `RENDER`, `SELECT`, or `FEEDBACK`. `RENDER` is the default, corresponding to rendering as described until now. `SELECT` specifies selection mode, and `FEEDBACK` specifies feedback mode (described below). Use of any of the name stack manipulation commands while the GL is not in selection mode has no effect.

Selection is controlled using

```
void SelectBuffer( sizei n, uint *buffer );
```

buffer is a pointer to an array of unsigned integers (called the selection array) to be potentially filled with names, and *n* is an integer indicating the maximum number of values that can be stored in that array. Placing the GL in selection mode before **SelectBuffer** has been called results in an error of `INVALID_OPERATION` as does calling **SelectBuffer** while in selection mode.

In selection mode, if a point, line, polygon, or the valid coordinates produced by a **RasterPos** command intersects the clip volume (section 2.17) then this primitive (or **RasterPos** command) causes a selection *hit*. **WindowPos** commands always generate a selection hit, since the resulting raster position is always valid. In the case of polygons, no hit occurs if the polygon would have been culled, but selection is based on the polygon itself, regardless of the setting of **PolygonMode**. When in selection mode, whenever a name stack manipulation command is executed or **RenderMode** is called and there has been a hit since the last time the stack was manipulated or **RenderMode** was called, then a *hit record* is written into the selection array.

A hit record consists of the following items in order: a non-negative integer giving the number of elements on the name stack at the time of the hit, a minimum depth value, a maximum depth value, and the name stack with the bottommost element first. The minimum and maximum depth values are the minimum and maximum taken over all the window coordinate *z* values of each (post-clipping) vertex of each primitive that intersects the clipping volume since the last hit record was

written. The minimum and maximum (each of which lies in the range $[0, 1]$) are each multiplied by $2^{32} - 1$ and rounded to the nearest unsigned integer to obtain the values that are placed in the hit record. No depth offset arithmetic (section 3.6.5) is performed on these values.

Hit records are placed in the selection array by maintaining a pointer into that array. When selection mode is entered, the pointer is initialized to the beginning of the array. Each time a hit record is copied, the pointer is updated to point at the array element after the one into which the topmost element of the name stack was stored. If copying the hit record into the selection array would cause the total number of values to exceed n , then as much of the record as fits in the array is written and an overflow flag is set.

Selection mode is exited by calling **RenderMode** with an argument value other than **SELECT**. When called while in selection mode, **RenderMode** returns the number of hit records copied into the selection array and resets the **SelectBuffer** pointer to its last specified value. Values are not guaranteed to be written into the selection array until **RenderMode** is called. If the selection array overflow flag was set, then **RenderMode** returns -1 and clears the overflow flag. The name stack is cleared and the stack pointer reset whenever **RenderMode** is called.

The state required for selection consists of the address of the selection array and its maximum size, the name stack and its associated pointer, a minimum and maximum depth value, and several flags. One flag indicates the current **RenderMode** value. In the initial state, the GL is in the **RENDER** mode. Another flag is used to indicate whether or not a hit has occurred since the last name stack manipulation. This flag is reset upon entering selection mode and whenever a name stack manipulation takes place. One final flag is required to indicate whether the maximum number of copied names would have been exceeded. This flag is reset upon entering selection mode. This flag, the address of the selection array, and its maximum size are GL client state.

5.3 Feedback

The GL is placed in feedback mode by calling **RenderMode** with **FEEDBACK**. When in feedback mode, framebuffer updates as described in chapter 4 are not performed. Instead, information about primitives that would have otherwise been rasterized is returned to the application via the *feedback buffer*.

Feedback is controlled using

```
void FeedbackBuffer( sizei n, enum type, float *buffer );
```


buffer is a pointer to an array of floating-point values into which feedback information will be placed, and *n* is a number indicating the maximum number of values that can be written to that array. *type* is a symbolic constant describing the information to be fed back for each vertex (see figure 5.2). The error `INVALID_OPERATION` results if the GL is placed in feedback mode before a call to **FeedbackBuffer** has been made, or if a call to **FeedbackBuffer** is made while in feedback mode.

While in feedback mode, each primitive that would be rasterized (or bitmap or call to **DrawPixels** or **CopyPixels**, if the raster position is valid) generates a block of values that get copied into the feedback array. If doing so would cause the number of entries to exceed the maximum, the block is partially written so as to fill the array (if there is any room left at all). The first block of values generated after the GL enters feedback mode is placed at the beginning of the feedback array, with subsequent blocks following. Each block begins with a code indicating the primitive type, followed by values that describe the primitive's vertices and associated data. Entries are also written for bitmaps and pixel rectangles. Feedback occurs after polygon culling (section 3.6.1) and **PolygonMode** interpretation of polygons (section 3.6.4) has taken place. It may also occur after polygons with more than three edges are broken up into triangles (if the GL implementation renders polygons by performing this decomposition). *x*, *y*, and *z* coordinates returned by feedback are window coordinates; if *w* is returned, it is in clip coordinates. No depth offset arithmetic (section 3.6.5) is performed on the *z* values. In the case of bitmaps and pixel rectangles, the coordinates returned are those of the current raster position.

The texture coordinates and colors returned are those resulting from the clipping operations described in section 2.19.8. Only coordinates for texture unit `TEXTURE0` are returned even for implementations which support multiple texture units. The colors returned are the primary colors.

The ordering rules for GL command interpretation also apply in feedback mode. Each command must be fully interpreted and its effects on both GL state and the values to be written to the feedback buffer completed before a subsequent command may be executed.

Feedback mode is exited by calling **RenderMode** with an argument value other than `FEEDBACK`. When called while in feedback mode, **RenderMode** returns the number of values placed in the feedback array and resets the feedback array pointer to be *buffer*. The return value never exceeds the maximum number of values passed to **FeedbackBuffer**.

If writing a value to the feedback buffer would cause more values to be written than the specified maximum number of values, then the value is not written and an overflow flag is set. In this case, **RenderMode** returns `-1` when it is called, after which the overflow flag is reset. While in feedback mode, values are not guaranteed

Type	coordinates	color	texture	total values
2D	x, y	—	—	2
3D	x, y, z	—	—	3
3D_COLOR	x, y, z	k	—	$3 + k$
3D_COLOR_TEXTURE	x, y, z	k	4	$7 + k$
4D_COLOR_TEXTURE	x, y, z, w	k	4	$8 + k$

Table 5.2: Correspondence of feedback type to number of values per vertex. k is 1 in color index mode and 4 in RGBA mode.

to be written into the feedback buffer before **RenderMode** is called.

Figure 5.2 gives a grammar for the array produced by feedback. Each primitive is indicated with a unique identifying value followed by some number of vertices. A vertex is fed back as some number of floating-point values determined by the feedback *type*. Table 5.2 gives the correspondence between feedback *buffer* and the number of values returned for each vertex.

The command

```
void PassThrough( float token );
```

may be used as a marker in feedback mode. *token* is returned as if it were a primitive; it is indicated with its own unique identifying value. The ordering of any **PassThrough** commands with respect to primitive specification is maintained by feedback. **PassThrough** may not occur between **Begin** and **End**. It has no effect when the GL is not in feedback mode.

The state required for feedback is the pointer to the feedback array, the maximum number of values that may be placed there, and the feedback *type*. An overflow flag is required to indicate whether the maximum allowable number of feedback values has been written; initially this flag is cleared. These state variables are GL client state. Feedback also relies on the same mode flag as selection to indicate whether the GL is in feedback, selection, or normal rendering mode.

5.4 Display Lists

A display list is simply a group of GL commands and arguments that has been stored for subsequent execution. The GL may be instructed to process a particular display list (possibly repeatedly) by providing a number that uniquely specifies it. Doing so causes the commands within the list to be executed just as if they were given normally. The only exception pertains to commands that rely upon client

feedback-list:		pixel-rectangle:
feedback-item feedback-list		DRAW_PIXEL_TOKEN vertex
feedback-item		COPY_PIXEL_TOKEN vertex
feedback-item:		passthrough:
point		PASS_THROUGH_TOKEN <i>f</i>
line-segment		
polygon		vertex:
bitmap		2D:
pixel-rectangle		<i>f f</i>
passthrough		3D:
		<i>f f f</i>
point:		3D_COLOR:
POINT_TOKEN vertex		<i>f f f</i> color
line-segment:		3D_COLOR_TEXTURE:
LINE_TOKEN vertex vertex		<i>f f f</i> color tex
LINE_RESET_TOKEN vertex vertex		4D_COLOR_TEXTURE:
polygon:		<i>f f f f</i> color tex
POLYGON_TOKEN <i>n</i> polygon-spec		
polygon-spec:		color:
polygon-spec vertex		<i>f f f f</i>
vertex vertex vertex		<i>f</i>
bitmap:		tex:
BITMAP_TOKEN vertex		<i>f f f f</i>

Figure 5.2: Feedback syntax. *f* is a floating-point number. *n* is a floating-point integer giving the number of vertices in a polygon. The symbols ending with `_TOKEN` are symbolic floating-point constants. The labels under the “vertex” rule show the different data returned for vertices depending on the feedback *type*. `LINE_TOKEN` and `LINE_RESET_TOKEN` are identical except that the latter is returned only when the line stipple is reset for that line segment.

state. When such a command is accumulated into the display list (that is, when issued, not when executed), the client state in effect at that time applies to the command. Only server state is affected when the command is executed. As always, pointers which are passed as arguments to commands are dereferenced when the command is issued. (Vertex array pointers are dereferenced when the commands **ArrayElement**, **DrawArrays**, **DrawElements**, or **DrawRangeElements** are accumulated into a display list.)

A display list is begun by calling

```
void NewList(uint n, enum mode);
```

n is a positive integer to which the display list that follows is assigned, and *mode* is a symbolic constant that controls the behavior of the GL during display list creation. If *mode* is `COMPILE`, then commands are not executed as they are placed in the display list. If *mode* is `COMPILE_AND_EXECUTE` then commands are executed as they are encountered, then placed in the display list. If *n* = 0, then the error `INVALID_VALUE` is generated.

After calling **NewList** all subsequent GL commands are placed in the display list (in the order the commands are issued) until a call to

```
void EndList(void);
```

occurs, after which the GL returns to its normal command execution state. It is only when **EndList** occurs that the specified display list is actually associated with the index indicated with **NewList**. The error `INVALID_OPERATION` is generated if **EndList** is called without a previous matching **NewList**, or if **NewList** is called a second time before calling **EndList**. The error `OUT_OF_MEMORY` is generated if **EndList** is called and the specified display list cannot be stored because insufficient memory is available. In this case GL implementations of revision 1.1 or greater insure that no change is made to the previous contents of the display list, if any, and that no other change is made to the GL state, except for the state changed by execution of GL commands when the display list mode is `COMPILE_AND_EXECUTE`.

Once defined, a display list is executed by calling

```
void CallList(uint n);
```

n gives the index of the display list to be called. This causes the commands saved in the display list to be executed, in order, just as if they were issued without using a display list. If *n* = 0, then the error `INVALID_VALUE` is generated.

The command

```
void CallLists( size_t n, enum type, void *lists );
```

provides an efficient means for executing a number of display lists. *n* is an integer indicating the number of display lists to be called, and *lists* is a pointer that points to an array of offsets. Each offset is constructed as determined by *lists* as follows. First, *type* may be one of the constants `BYTE`, `UNSIGNED_BYTE`, `SHORT`, `UNSIGNED_SHORT`, `INT`, `UNSIGNED_INT`, or `FLOAT` indicating that the array pointed to by *lists* is an array of bytes, unsigned bytes, shorts, unsigned shorts, integers, unsigned integers, or floats, respectively. In this case each offset is found by simply converting each array element to an integer (floating point values are truncated to negative infinity). Further, *type* may be one of `2_BYTES`, `3_BYTES`, or `4_BYTES`, indicating that the array contains sequences of 2, 3, or 4 unsigned bytes, in which case each integer offset is constructed according to the following algorithm:

```
offset ← 0
for i = 1 to b
    offset ← offset shifted left 8 bits
    offset ← offset + byte
    advance to next byte in the array
```

b is 2, 3, or 4, as indicated by *type*. If *n* = 0, **CallLists** does nothing.

Each of the *n* constructed offsets is taken in order and added to a display list base to obtain a display list number. For each number, the indicated display list is executed. The base is set by calling

```
void ListBase( uint base );
```

to specify the offset.

Indicating a display list index that does not correspond to any display list has no effect. **CallList** or **CallLists** may appear inside a display list. (If the *mode* supplied to **NewList** is `COMPILE_AND_EXECUTE`, then the appropriate lists are executed, but the **CallList** or **CallLists**, rather than those lists' constituent commands, is placed in the list under construction.) To avoid the possibility of infinite recursion resulting from display lists calling one another, an implementation dependent limit is placed on the nesting level of display lists during display list execution. This limit must be at least 64.

Two commands are provided to manage display list indices.

```
uint GenLists( size_t s );
```

returns an integer n such that the indices $n, \dots, n+s-1$ are previously unused (i.e. there are s previously unused display list indices starting at n). **GenLists** also has the effect of creating an empty display list for each of the indices $n, \dots, n+s-1$, so that these indices all become used. **GenLists** returns 0 if there is no group of s contiguous previously unused display list indices, or if $s = 0$.

```
boolean IsList( uint list );
```

returns TRUE if *list* is the index of some display list.

A contiguous group of display lists may be deleted by calling

```
void DeleteLists( uint list, sizei range );
```

where *list* is the index of the first display list to be deleted and *range* is the number of display lists to be deleted. All information about the display lists is lost, and the indices become unused. Indices to which no display list corresponds are ignored. If *range* = 0, nothing happens.

Certain commands, when called while compiling a display list, are not compiled into the display list but are executed immediately. These commands fall in several categories including

Display lists: **GenLists** and **DeleteLists**.

Render modes: **FeedbackBuffer**, **SelectBuffer**, and **RenderMode**.

Vertex arrays: **ClientActiveTexture**, **ColorPointer**, **EdgeFlagPointer**, **FogCoordPointer**, **IndexPointer**, **InterleavedArrays**, **NormalPointer**, **SecondaryColorPointer**, **TexCoordPointer**, **VertexAttribPointer**, **VertexAttribIPointer**, **VertexPointer**, **GenVertexArrays**, **DeleteVertexArrays**, and **BindVertexArray**.

Client state: **EnableClientState**, **DisableClientState**, **EnableVertexAttribArray**, **DisableVertexAttribArray**, **PushClientAttrib**, and **PopClientAttrib**.

Pixels and textures: **PixelStore**, **ReadPixels**, **GenTextures**, **DeleteTextures**, **AreTexturesResident**, and **GenerateMipmap**.

Occlusion queries: **GenQueries** and **DeleteQueries**.

Vertex buffer objects: **GenBuffers**, **DeleteBuffers**, **BindBuffer**, **BindBufferRange**, **BindBufferBase**, **TransformFeedbackVaryings**, **BufferData**, **BufferSubData**, **MapBuffer**, and **UnmapBuffer**.

Framebuffer and renderbuffer objects: **GenFramebuffers**, **BindFramebuffer**, **DeleteFramebuffers**, **CheckFramebufferStatus**, **GenRenderbuffers**, **BindRenderbuffer**, **DeleteRenderbuffers**, **RenderbufferStorage**, **RenderbufferStorageMultisample**, **FramebufferTexture1D**, **FramebufferTexture2D**, **FramebufferTexture3D**, **FramebufferTextureLayer**, **FramebufferRenderbuffer**, and **BlitFramebuffer**.

Program and shader objects: **CreateProgram**, **CreateShader**, **DeleteProgram**, **DeleteShader**, **AttachShader**, **DetachShader**, **BindAttribLocation**, **BindFragDataLocation**, **CompileShader**, **ShaderSource**, **LinkProgram**, and **ValidateProgram**.

GL command stream management: **Finish**, and **Flush**.

Other queries: All query commands whose names begin with **Get** and **Is** (see chapter 6).

GL commands that source data from buffer objects dereference the buffer object data in question at display list compile time, rather than encoding the buffer ID and buffer offset into the display list. Only GL commands that are executed immediately, rather than being compiled into a display list, are permitted to use a buffer object as a data sink.

TexImage3D, **TexImage2D**, **TexImage1D**, **Histogram**, and **ColorTable** are executed immediately when called with the corresponding proxy arguments `PROXY_TEXTURE_3D` or `PROXY_TEXTURE_2D_ARRAY`; `PROXY_TEXTURE_2D` `PROXY_TEXTURE_1D_ARRAY`, or `PROXY_TEXTURE_CUBE_MAP`; `PROXY_TEXTURE_1D`; `PROXY_HISTOGRAM`; and `PROXY_COLOR_TABLE`, `PROXY_POST_CONVOLUTION_COLOR_TABLE`, or `PROXY_POST_COLOR_MATRIX_COLOR_TABLE`.

When a program object is in use, a display list may be executed whose vertex attribute calls do not match up exactly with what is expected by the vertex shader contained in that program object. Handling of this mismatch is described in section 2.20.3.

Display lists require one bit of state to indicate whether a GL command should be executed immediately or placed in a display list. In the initial state, commands are executed immediately. If the bit indicates display list creation, an index is required to indicate the current display list being defined. Another bit indicates, during display list creation, whether or not commands should be executed as they are compiled into the display list. One integer is required for the current **ListBase** setting; its initial value is zero. Finally, state must be maintained to indicate which integers are currently in use as display list indices. In the initial state, no indices are in use.

5.5 Flush and Finish

The command

```
void Flush( void );
```

indicates that all commands that have previously been sent to the GL must complete in finite time.

The command

```
void Finish( void );
```

forces all previous GL commands to complete. **Finish** does not return until all effects from previously issued commands on GL client and server state and the framebuffer are fully realized.

5.6 Hints

Certain aspects of GL behavior, when there is room for variation, may be controlled with hints. A hint is specified using

```
void Hint( enum target, enum hint );
```

target is a symbolic constant indicating the behavior to be controlled, and *hint* is a symbolic constant indicating what type of behavior is desired. The possible *targets* are described in table 5.3; for each *target*, *hint* must be one of `FASTEST`, indicating that the most efficient option should be chosen; `NICEST`, indicating that the highest quality option should be chosen; and `DONT_CARE`, indicating no preference in the matter.

For the texture compression hint, a *hint* of `FASTEST` indicates that texture images should be compressed as quickly as possible, while `NICEST` indicates that the texture images be compressed with as little image degradation as possible. `FASTEST` should be used for one-time texture compression, and `NICEST` should be used if the compression results are to be retrieved by **GetCompressedTexImage** (section 6.1.4) for reuse.

The interpretation of hints is implementation dependent. An implementation may ignore them entirely.

The initial value of all hints is `DONT_CARE`.

Target	Hint description
PERSPECTIVE_CORRECTION_HINT	Quality of parameter interpolation
POINT_SMOOTH_HINT	Point sampling quality
LINE_SMOOTH_HINT	Line sampling quality
POLYGON_SMOOTH_HINT	Polygon sampling quality
FOG_HINT	Fog quality (calculated per-pixel or per-vertex)
GENERATE_MIPMAP_HINT	Quality and performance of automatic mipmap level generation
TEXTURE_COMPRESSION_HINT	Quality and performance of texture image compression
FRAGMENT_SHADER_DERIVATIVE_HINT	Derivative accuracy for fragment processing built-in functions dFdx, dFdy and fwidth

Table 5.3: Hint targets and descriptions.

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 is interpreted as either 1 or 0, and a floating-point value is rounded to the nearest integer, unless the value is an RGBA color component, a **DepthRange** value, a depth buffer clear value, or a normal coordinate. In these cases, the **Get** command converts the floating-point value to an integer according to the `INT` entry of table 4.9; a value not in $[-1, 1]$ converts to an undefined value. If **GetFloatv** is called, a boolean value is interpreted as either 1.0 or 0.0, an integer is coerced to floating-point, and a double-precision floating-point value is converted to single-precision. Analogous conversions are carried out in the case of **GetDoublev**. If a value is so large in magnitude that it cannot be represented with the requested type, then the nearest value representable using the requested type is returned.

Unless otherwise indicated, multi-valued state variables return their multiple values in the same order as they are given as arguments to the commands that set them. For instance, the two **DepthRange** parameters are returned in the order *n* followed by *f*. Similarly, points for evaluator maps are returned in the order that they appeared when passed to **Map1**. **Map2** returns R_{ij} in the $[(uorder)i + j]$ th block of values (see page 297 for *i*, *j*, *uorder*, and R_{ij}).

Matrices may be queried and returned in transposed form by calling **GetBooleanv**, **GetIntegerv**, **GetFloatv**, and **GetDoublev** with *pname* set to one of `TRANSPOSE_MODELVIEW_MATRIX`, `TRANSPOSE_PROJECTION_MATRIX`, `TRANSPOSE_TEXTURE_MATRIX`, or `TRANSPOSE_COLOR_MATRIX`. The effect of

```
GetFloatv (TRANSPOSE_MODELVIEW_MATRIX, m) ;
```

is the same as the effect of the command sequence

```
GetFloatv (MODELVIEW_MATRIX, m) ;  
m = mT ;
```

Similar conversions occur when querying `TRANSPOSE_PROJECTION_MATRIX`, `TRANSPOSE_TEXTURE_MATRIX`, and `TRANSPOSE_COLOR_MATRIX`.

If fragment color clamping is enabled, querying of the texture border color, texture environment color, fog color, alpha test reference value, blend color, and RGBA clear color will clamp the corresponding state values to $[0, 1]$ before returning them. This behavior provides compatibility with previous versions of the GL that clamped these values when specified.

Most texture state variables are qualified by the value of `ACTIVE_TEXTURE` to determine which server texture state vector is queried. Client texture state variables such as texture coordinate array pointers are qualified by the value of `CLIENT_ACTIVE_TEXTURE`. Tables 6.5, 6.6, 6.12, 6.19, 6.22, and 6.47 indicate those state variables which are qualified by `ACTIVE_TEXTURE` or `CLIENT_ACTIVE_TEXTURE` during state queries. Queries of texture state variables corresponding to texture coordinate processing units (namely, **TexGen** state and enables, and matrices) will generate an `INVALID_OPERATION` error if the value of `ACTIVE_TEXTURE` is greater than or equal to `MAX_TEXTURE_COORDS`. All other texture state queries will result in an `INVALID_OPERATION` error if the value of `ACTIVE_TEXTURE` is greater than or equal to `MAX_COMBINED_TEXTURE_IMAGE_UNITS`.

Vertex array state variables are qualified by the value of `VERTEX_ARRAY_BINDING` to determine which vertex array object is queried. Tables 6.6 through 6.9 define the set of state stored in a vertex array object.

6.1.3 Enumerated Queries

Other commands exist to obtain state variables that are identified by a category (clip plane, light, material, etc.) as well as a symbolic constant. These are

```
void GetClipPlane( enum plane, double eqn[4] );
```

```

void GetLight{if}v( enum light, enum value, T data );
void GetMaterial{if}v( enum face, enum value, T data );
void GetTexEnv{if}v( enum env, enum value, T data );
void GetTexGen{ifd}v( enum coord, enum value, T data );
void GetTexParameter{if}v( enum target, enum value,
    T data );
void GetTexParameterI{i ui}v( enum target, enum value,
    T data );
void GetTexLevelParameter{if}v( enum target, int lod,
    enum value, T data );
void GetPixelMap{ui us f}v( enum map, T data );
void GetMap{ifd}v( enum map, enum value, T data );
void GetBufferParameteriv( enum target, enum value,
    T data );
void GetFramebufferAttachmentParameteriv( enum target,
    enum attachment, enum pname, int *params );
void GetRenderbufferParameteriv( enum target, enum pname,
    int* params );

```

GetLightiv, **GetMaterialiv**, **GetTexEnviv**, **GetTexGeniv**, and **GetTexParameteriv** convert floating point state to integer values in the same manner as **GetIntegerv** (see section 6.1.2).

GetClipPlane always returns four double-precision values in *eqn*; these are the coefficients of the plane equation of *plane* in eye coordinates (these coordinates are those that were computed when the plane was specified).

GetLight places information about *value* (a symbolic constant) for *light* (also a symbolic constant) in *data*. **POSITION** or **SPOT_DIRECTION** returns values in eye coordinates (again, these are the coordinates that were computed when the position or direction was specified).

GetMaterial, **GetTexGen**, **GetTexEnv**, **GetTexParameter**, and **GetBufferParameter** are similar to **GetLight**, placing information about *value* for the target indicated by their first argument into *data*. The *face* argument to **GetMaterial** must be either **FRONT** or **BACK**, indicating the front or back material, respectively. The *env* argument to **GetTexEnv** must be either **POINT_SPRITE**, **TEXTURE_ENV**, or **TEXTURE_FILTER_CONTROL**. The *coord* argument to **GetTexGen** must be one of **S**, **T**, **R**, or **Q**. For **GetTexGen**, **EYE_LINEAR** coefficients are returned in the eye coordinates that were computed when the plane was specified; **OBJECT_LINEAR** coefficients are returned in object coordinates.

GetTexParameter parameter *target* may be one of **TEXTURE_1D**, **TEXTURE_2D**, **TEXTURE_3D**, **TEXTURE_1D_ARRAY**, **TEXTURE_2D_ARRAY**, or

TEXTURE_CUBE_MAP, indicating the currently bound one-, two-, three-dimensional, one- or two-dimensional array, or cube map texture object. **GetTexLevelParameter** parameter *target* may be one of TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, TEXTURE_CUBE_MAP_POSITIVE_X, TEXTURE_CUBE_MAP_NEGATIVE_X, TEXTURE_CUBE_MAP_POSITIVE_Y, TEXTURE_CUBE_MAP_NEGATIVE_Y, TEXTURE_CUBE_MAP_POSITIVE_Z, TEXTURE_CUBE_MAP_NEGATIVE_Z, PROXY_TEXTURE_1D, PROXY_TEXTURE_2D, PROXY_TEXTURE_3D, PROXY_TEXTURE_1D_ARRAY, PROXY_TEXTURE_2D_ARRAY, or PROXY_TEXTURE_CUBE_MAP, indicating the one-, two-, or three-dimensional texture, the one- or two-dimensional array texture, one of the six distinct 2D images making up the cube map texture object, or the one-, two-, three-dimensional, one- or two-dimensional array, or cube map proxy state vector. Note that TEXTURE_CUBE_MAP is not a valid *target* parameter for **GetTexLevelParameter**, because it does not specify a particular cube map face. *value* is a symbolic value indicating which texture parameter is to be obtained. For **GetTexParameter**, *value* must be either TEXTURE_RESIDENT, or one of the symbolic values in table 3.20. Querying *value* TEXTURE_BORDER_COLOR with **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. The *lod* argument to **GetTexLevelParameter** determines which level-of-detail's state is returned. If the *lod* argument is less than zero or if it is larger than the maximum allowable level-of-detail then the error INVALID_VALUE occurs.

For texture images with uncompressed internal formats, queries of *value* of TEXTURE_RED_TYPE, TEXTURE_GREEN_TYPE, TEXTURE_BLUE_TYPE, TEXTURE_ALPHA_TYPE, TEXTURE_LUMINANCE_TYPE, TEXTURE_DEPTH_TYPE, and TEXTURE_INTENSITY_TYPE return the data type used to store the component. Types NONE, UNSIGNED_NORMALIZED, FLOAT, INT, and UNSIGNED_INT respectively indicate missing, unsigned normalized integer, floating-point, signed unnormalized integer, and unsigned unnormalized integer components. Queries of *value* of TEXTURE_RED_SIZE, TEXTURE_GREEN_SIZE, TEXTURE_BLUE_SIZE, TEXTURE_ALPHA_SIZE, TEXTURE_LUMINANCE_SIZE, TEXTURE_INTENSITY_SIZE, TEXTURE_DEPTH_SIZE, TEXTURE_STENCIL_SIZE, and TEXTURE_SHARED_SIZE return the actual resolutions of the stored image array components, not the resolutions specified when the image array was defined. For texture images with 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 bytes) of the compressed texture image that would be returned by **GetCompressedTexImage** (section 6.1.4). Querying `TEXTURE_COMPRESSED_IMAGE_SIZE` is not allowed on texture images with an uncompressed internal format or on proxy targets and will result in an `INVALID_OPERATION` error if attempted.

Queries of *value* `TEXTURE_WIDTH`, `TEXTURE_HEIGHT`, `TEXTURE_DEPTH`, and `TEXTURE_BORDER` return the width, height, depth, and border as specified when the image array was created. The internal format of the image array is queried as `TEXTURE_INTERNAL_FORMAT`, or as `TEXTURE_COMPONENTS` for compatibility with GL version 1.0.

For **GetPixelMap**, the *map* must be a map name from table 3.3. For **GetMap**, *map* must be one of the map types described in section 5.1, and *value* must be one of `ORDER`, `COEFF`, or `DOMAIN`. The **GetPixelMapfv**, **GetPixelMapuiv**, and **GetPixelMapusv** commands write all the values in the named pixel map to *data*. **GetPixelMapuiv** and **GetPixelMapusv** convert floating point pixel map values to integers according to the `UNSIGNED_INT` and `UNSIGNED_SHORT` entries, respectively, of table 4.9. If a pixel pack buffer is bound (as indicated by a non-zero value of `PIXEL_PACK_BUFFER_BINDING`), *data* is an offset into the pixel pack buffer; otherwise, *data* is a pointer to client memory. All pixel storage and pixel transfer modes are ignored when returning a pixel map. *n* machine units are written where *n* is the size of the pixel map times the size of `FLOAT`, `UNSIGNED_INT`, or `UNSIGNED_SHORT` respectively in basic machine units. If a pixel pack buffer object is bound and *data* + *n* is greater than the size of the pixel buffer, an `INVALID_OPERATION` error results. If a pixel pack buffer object is bound and *data* is not evenly divisible by the number of basic machine units needed to store in memory a `FLOAT`, `UNSIGNED_INT`, or `UNSIGNED_SHORT` respectively, an `INVALID_OPERATION` error results.

For **GetFramebufferAttachmentParameteriv**, *target* must be `DRAW_FRAMEBUFFER`, `READ_FRAMEBUFFER`, or `FRAMEBUFFER`. `FRAMEBUFFER` is equivalent to `DRAW_FRAMEBUFFER`.

If the default framebuffer is bound to *target*, then *attachment* must be one of `FRONT_LEFT`, `FRONT_RIGHT`, `BACK_LEFT`, `BACK_RIGHT`, `AUXi`, `DEPTH_BUFFER`, or `STENCIL_BUFFER`, identifying a color buffer, the depth buffer, or the stencil buffer, and *pname* may be `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` or `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME`.

If a framebuffer object is bound to *target*, then *attachment* must be one of the attachment points of the framebuffer listed in table 4.12

and *pname* may be one of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE`, `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME`, `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL`, `FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE`, or `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER`.

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.

In either case, *pname* may also be one of `FRAMEBUFFER_ATTACHMENT_RED_SIZE`, `FRAMEBUFFER_ATTACHMENT_GREEN_SIZE`, `FRAMEBUFFER_ATTACHMENT_BLUE_SIZE`, `FRAMEBUFFER_ATTACHMENT_ALPHA_SIZE`, `FRAMEBUFFER_ATTACHMENT_DEPTH_SIZE`, `FRAMEBUFFER_ATTACHMENT_STENCIL_SIZE`, `FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE`, or `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING`.

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. 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`, `UNSIGNED_NORMALIZED`, or `INDEX` for floating-point, signed integer, unsigned integer, unsigned fixed-point, or index components respectively. Only color buffers may have index or integer components. If *pname* is `FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING`, *param* will contain the encoding of components of the specified attachment, one of `LINEAR` or `SRGB` for linear or sRGB-encoded components, respectively. Only color buffer components may be sRGB-encoded; such components are treated as described in sections 4.1.8 and 4.1.9. For the default framebuffer, color encoding is determined by the implementation. For framebuffer objects, components are sRGB-encoded if the internal format of a color attachment is one of the color-renderable SRGB formats described in section 3.9.15.

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.
- Otherwise, an `INVALID_ENUM` error is generated.

If the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` is `TEXTURE`, then

- If *pname* is `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME`, then *params* will contain the name of the texture object which contains the attached image.
- If *pname* is `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL`, then *params* will contain the mipmap level of the texture object which contains the attached image.
- If *pname* is `FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE` and the texture object named `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` is a cube map texture, then *params* will contain the cube map face of the cube-map texture object which contains the attached image. Otherwise *params* will contain the value zero.
- If *pname* is `FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER` and the texture object named `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` is a three-dimensional texture or a one- or two-dimensional array texture, then *params* will contain the number of the texture layer which contains the attached image. Otherwise *params* will contain the value zero.
- Otherwise, an `INVALID_ENUM` error is generated.

For **GetRenderbufferParameteriv**, *target* must be `RENDERBUFFER` and *pname* must be one of the symbolic values in table 6.31. If the renderbuffer currently bound to *target* is zero, then an `INVALID_OPERATION` error is generated.

Upon successful return from **GetRenderbufferParameteriv**, if *pname* is `RENDERBUFFER_WIDTH`, `RENDERBUFFER_HEIGHT`, `RENDERBUFFER_INTERNAL_FORMAT`, or `RENDERBUFFER_SAMPLES`, then *params* will contain the width in pixels, height in pixels, internal format, or number of samples, respectively, of the image of the renderbuffer currently bound to *target*.

If *pname* is `RENDERBUFFER_RED_SIZE`, `RENDERBUFFER_GREEN_SIZE`, `RENDERBUFFER_BLUE_SIZE`, `RENDERBUFFER_ALPHA_SIZE`, `RENDERBUFFER_DEPTH_SIZE`, or `RENDERBUFFER_STENCIL_SIZE`, then *params* will contain the actual resolutions, (not the resolutions specified when the image array was defined), for the red, green, blue, alpha depth, or stencil components, respectively, of the image of the renderbuffer currently bound to *target*.

Otherwise, an `INVALID_ENUM` error is generated.

6.1.4 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`, and `TEXTURE_2D_ARRAY` indicate a one-, two-, or three-dimensional or one- or two-dimensional array texture respectively. `TEXTURE_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, and `TEXTURE_CUBE_MAP_NEGATIVE_Z` indicate the respective face of a cube map texture. *lod* is a level-of-detail number, *format* is a pixel format from table 3.6, *type* is a pixel type from table 3.5.

Calling **GetTexImage** with a color format (one of `RED`, `GREEN`, `BLUE`, `ALPHA`, `RG`, `RGB`, `BGR`, `RGBA`, `BGRA`, `LUMINANCE`, or `LUMINANCE_ALPHA`) when the base internal format of the texture image is not a color format; with a format of `DEPTH_COMPONENT` when the base internal format is not `DEPTH_COMPONENT` or `DEPTH_STENCIL`; or with a format of `DEPTH_STENCIL` when the base internal format is not `DEPTH_STENCIL`, causes the error `INVALID_OPERATION`.

GetTexImage obtains component groups from a texture image with the indicated level-of-detail. If *format* is a color format then the components are assigned among R, G, B, and A according to table 6.1, starting with the first group in the first row, and continuing by obtaining groups in order from each row and proceeding from the first row to the last, and from the first image to the last for three-dimensional textures. One- and two-dimensional array textures are treated as two- and three-dimensional images, respectively, where the layers are treated as rows or images. If *format* is `DEPTH_COMPONENT`, then each depth component is assigned with the same ordering of rows and images. If *format* is `DEPTH_STENCIL`, then each depth component and each stencil index is assigned with the same ordering of rows and images.

These groups are then packed and placed in client or pixel buffer object memory. If a pixel pack buffer is bound (as indicated by a non-zero value of `PIXEL_PACK_BUFFER_BINDING`), *img* is an offset into the pixel pack buffer; otherwise, *img* is a pointer to client memory. No pixel transfer operations are performed on this image, but pixel storage modes that are applicable to **ReadPixels** are applied.

Base Internal Format	R	G	B	A
ALPHA	0	0	0	A_i
LUMINANCE (or 1)	L_i	0	0	1
LUMINANCE_ALPHA (or 2)	L_i	0	0	A_i
INTENSITY	I_i	0	0	1
RED	R_i	0	0	1
RG	R_i	G_i	0	1
RGB (or 3)	R_i	G_i	B_i	1
RGBA (or 4)	R_i	G_i	B_i	A_i

Table 6.1: Texture, table, and filter return values. R_i , G_i , B_i , A_i , L_i , and I_i are components of the internal format that are assigned to pixel values R, G, B, and A. If a requested pixel value is not present in the internal format, the specified constant value is used.

For three-dimensional and two-dimensional array textures, pixel storage operations are applied as if the image were two-dimensional, except that the additional pixel storage state values `PACK_IMAGE_HEIGHT` and `PACK_SKIP_IMAGES` are applied. The correspondence of texels to memory locations is as defined for **TexImage3D** in section 3.9.1.

The row length, number of rows, image depth, and number of images are determined by the size of the texture image (including any borders). Calling **GetTexImage** with *lod* less than zero or larger than the maximum allowable causes the error `INVALID_VALUE`. Calling **GetTexImage** with a *format* of `COLOR_INDEX` or `STENCIL_INDEX` causes the error `INVALID_ENUM`. 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 a `FLOAT`, `UNSIGNED_INT`, or `UNSIGNED_SHORT` respectively, 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* bytes of compressed image data to the pixel pack buffer or client memory pointed to by *img*, where *n* is the value of

TEXTURE_COMPRESSED_IMAGE_SIZE for the texture. The compressed image data is formatted according to the definition of the texture's internal format. All pixel storage and pixel transfer modes are ignored when returning a compressed texture image.

Calling **GetCompressedTexImage** with an *lod* value less than zero or greater than the maximum allowable causes an `INVALID_VALUE` error. Calling **GetCompressedTexImage** with a texture image stored with an uncompressed internal format causes an `INVALID_OPERATION` error. If a pixel pack buffer object is bound and *img + n* is greater than the size of the buffer, an `INVALID_OPERATION` error results.

The command

```
boolean IsTexture( uint texture );
```

returns `TRUE` if *texture* is the name of a texture object. If *texture* is zero, or is a non-zero value that is not the name of a texture object, or if an error condition occurs, **IsTexture** returns `FALSE`. A name returned by **GenTextures**, but not yet bound, is not the name of a texture object.

6.1.5 Stipple Query

The command

```
void GetPolygonStipple( void *pattern );
```

obtains the polygon stipple. The pattern is packed into pixel pack buffer or client memory according to the procedure given in section 4.3.2 for **ReadPixels**; it is as if the *height* and *width* passed to that command were both equal to 32, the *type* were `BITMAP`, and the *format* were `COLOR_INDEX`.

6.1.6 Color Matrix Query

The scale and bias variables are queried using **GetFloatv** with *pname* set to the appropriate variable name. The top matrix on the color matrix stack is returned by **GetFloatv** called with *pname* set to `COLOR_MATRIX` or `TRANSPOSE_COLOR_MATRIX`. The depth of the color matrix stack, and the maximum depth of the color matrix stack, are queried with **GetIntegerv**, setting *pname* to `COLOR_MATRIX_STACK_DEPTH` and `MAX_COLOR_MATRIX_STACK_DEPTH` respectively.

6.1.7 Color Table Query

The current contents of a color table are queried using

```
void GetColorTable( enum target, enum format, enum type,
                    void *table );
```

target must be one of the *regular* color table names listed in table 3.4. *format* and *type* accept the same values as do the corresponding parameters of **GetTexImage**, except that a format of `DEPTH_COMPONENT` causes the error `INVALID_ENUM`. The one-dimensional color table image is returned to pixel pack buffer or client memory starting at *table*. No pixel transfer operations are performed on this image, but pixel storage modes that are applicable to **ReadPixels** are performed. Color components that are requested in the specified *format*, but which are not included in the internal format of the color lookup table, are returned as zero. The assignments of internal color components to the components requested by *format* are described in table 6.1.

The functions

```
void GetColorTableParameter{if}v( enum target,
                                   enum pname, T params );
```

are used for integer and floating point query.

target must be one of the regular or proxy color table names listed in table 3.4. *pname* is one of `COLOR_TABLE_SCALE`, `COLOR_TABLE_BIAS`, `COLOR_TABLE_FORMAT`, `COLOR_TABLE_WIDTH`, `COLOR_TABLE_RED_SIZE`, `COLOR_TABLE_GREEN_SIZE`, `COLOR_TABLE_BLUE_SIZE`, `COLOR_TABLE_ALPHA_SIZE`, `COLOR_TABLE_LUMINANCE_SIZE`, or `COLOR_TABLE_INTENSITY_SIZE`. The value of the specified parameter is returned in *params*.

6.1.8 Convolution Query

The current contents of a convolution filter image are queried with the command

```
void GetConvolutionFilter( enum target, enum format,
                           enum type, void *image );
```

target must be `CONVOLUTION_1D` or `CONVOLUTION_2D`. *format* and *type* accept the same values as do the corresponding parameters of **GetTexImage**, except that a format of `DEPTH_COMPONENT` causes the error `INVALID_ENUM`. The one-dimensional or two-dimensional images is returned to pixel pack buffer or client

memory starting at *image*. Pixel processing and component mapping are identical to those of **GetTexImage**.

The current contents of a separable filter image are queried using

```
void GetSeparableFilter( enum target, enum format,
                        enum type, void *row, void *column, void *span );
```

target must be SEPARABLE_2D. *format* and *type* accept the same values as do the corresponding parameters of **GetTexImage**. The row and column images are returned to pixel pack buffer or client memory starting at *row* and *column* respectively. *span* is currently unused. Pixel processing and component mapping are identical to those of **GetTexImage**.

The functions

```
void GetConvolutionParameter{if}v( enum target,
                                   enum pname, T params );
```

are used for integer and floating point query. *target* must be CONVOLUTION_1D, CONVOLUTION_2D, or SEPARABLE_2D. *pname* is one of CONVOLUTION_BORDER_COLOR, CONVOLUTION_BORDER_MODE, CONVOLUTION_FILTER_SCALE, CONVOLUTION_FILTER_BIAS, CONVOLUTION_FORMAT, CONVOLUTION_WIDTH, CONVOLUTION_HEIGHT, MAX_CONVOLUTION_WIDTH, or MAX_CONVOLUTION_HEIGHT. The value of the specified parameter is returned in *params*.

6.1.9 Histogram Query

The current contents of the histogram table are queried using

```
void GetHistogram( enum target, boolean reset,
                   enum format, enum type, void* values );
```

target must be HISTOGRAM. *type* and *format* accept the same values as do the corresponding parameters of **GetTexImage**, except that a format of DEPTH_COMPONENT causes the error INVALID_ENUM. The one-dimensional histogram table image is returned to pixel pack buffer or client memory starting at *type*. Pixel processing and component mapping are identical to those of **GetTexImage**, except that instead of applying the Final Conversion pixel storage mode, component values are simply clamped to the range of the target data type.

If *reset* is TRUE, then all counters of all elements of the histogram are reset to zero. Counters are reset whether returned or not.

No counters are modified if *reset* is FALSE.

Calling

```
void ResetHistogram( enum target );
```

resets all counters of all elements of the histogram table to zero. *target* must be HISTOGRAM.

It is not an error to reset or query the contents of a histogram table with zero entries.

The functions

```
void GetHistogramParameter{if}v( enum target,  
    enum pname, T params );
```

are used for integer and floating point query. *target* must be HISTOGRAM or PROXY_HISTOGRAM. *pname* is one of HISTOGRAM_FORMAT, HISTOGRAM_WIDTH, HISTOGRAM_RED_SIZE, HISTOGRAM_GREEN_SIZE, HISTOGRAM_BLUE_SIZE, HISTOGRAM_ALPHA_SIZE, or HISTOGRAM_LUMINANCE_SIZE. *pname* may be HISTOGRAM_SINK only for *target* HISTOGRAM. The value of the specified parameter is returned in *params*.

6.1.10 Minmax Query

The current contents of the minmax table are queried using

```
void GetMinmax( enum target, boolean reset, enum format,  
    enum type, void* values );
```

target must be MINMAX. *type* and *format* accept the same values as do the corresponding parameters of **GetTexImage**, except that a format of DEPTH_COMPONENT causes the error INVALID_ENUM. A one-dimensional image of width 2 is returned to pixel pack buffer or client memory starting at *values*. Pixel processing and component mapping are identical to those of **GetTexImage**.

If *reset* is TRUE, then each minimum value is reset to the maximum representable value, and each maximum value is reset to the minimum representable value. All values are reset, whether returned or not.

No values are modified if *reset* is FALSE.

Calling

```
void ResetMinmax( enum target );
```

resets all minimum and maximum values of *target* to their maximum and minimum representable values, respectively, *target* must be MINMAX.

The functions

```
void GetMinmaxParameter{if}v( enum target, enum pname,
                               T params );
```

are used for integer and floating point query. *target* must be `MINMAX`. *pname* is `MINMAX_FORMAT` or `MINMAX_SINK`. The value of the specified parameter is returned in *params*.

6.1.11 Pointer and String Queries

The command

```
void GetPointerv( enum pname, void **params );
```

obtains the pointer or pointers named *pname* in the array *params*. The possible values for *pname* are `SELECTION_BUFFER_POINTER` and `FEEDBACK_BUFFER_POINTER`, which respectively return the pointers set with **SelectBuffer** and **FeedbackBuffer**; and `VERTEX_ARRAY_POINTER`, `NORMAL_ARRAY_POINTER`, `COLOR_ARRAY_POINTER`, `SECONDARY_COLOR_ARRAY_POINTER`, `INDEX_ARRAY_POINTER`, `TEXTURE_COORD_ARRAY_POINTER`, `FOG_COORD_ARRAY_POINTER`, and `EDGE_FLAG_ARRAY_POINTER`, which respectively return the corresponding value stored in the currently bound vertex array object. Each *pname* returns a single pointer value.

String queries return pointers to UTF-8 encoded, NULL-terminated static strings describing properties of the current GL context ¹. The command

```
ubyte *GetString( enum name );
```

accepts *name* values of `VENDOR`, `RENDERER`, `VERSION`, `SHADING_LANGUAGE_VERSION`, and `EXTENSIONS`. The format of the `RENDERER` and `VENDOR` strings is implementation dependent. The `EXTENSIONS` string contains a space separated list of extension names (the extension names themselves do not contain any spaces). The `VERSION` and `SHADING_LANGUAGE_VERSION` strings are laid out as follows:

```
<version number><space><vendor-specific information>
```

¹Applications making copies of these static strings should never use a fixed-length buffer, because the strings may grow unpredictably between releases, resulting in buffer overflow when copying. This is particularly true of the `EXTENSIONS` string, which has become extremely long in some GL implementations.

The version number is either of the form *major_number.minor_number* or *major_number.minor_number.release_number*, where the numbers all have one or more digits. The *release_number* and vendor specific information are optional. However, if present, then they pertain to the server and their format and contents are implementation dependent.

GetString returns the version number (in the `VERSION` string) and the extension names (in the `EXTENSIONS` string) that can be supported by the current GL context. Thus, if the client and server support different versions and/or extensions, a compatible version and list of extensions is returned.

The GL version may also be queried by calling **GetIntegerv** with *values* `MAJOR_VERSION` and `MINOR_VERSION`, which respectively return the same values as *major_number* and *minor_number* in the `VERSION` string, and *value* `CONTEXT_FLAGS`, which returns a set of flags defining additional properties of a context. If `CONTEXT_FLAG_FORWARD_COMPATIBLE_BIT` is set in `CONTEXT_FLAGS`, then the context is a forward-compatible context as defined in appendix E, and the deprecated features described in that appendix are not supported; otherwise the context is a full context, and all features described in the specification are supported.

Indexed strings are queried with the command

```
ubyte *GetStringi( enum name, uint index );
```

target is the name of the indexed state and *index* is the index of the particular element being queried. *target* may only be `EXTENSIONS`, indicating that the extension name corresponding to the *index*th supported extension should be returned. *index* may range from zero to the value of `NUM_EXTENSIONS` minus one. All extension names, and only the extension names returned in **GetString**(`EXTENSIONS`) will be returned as individual names, but there is no defined relationship between the order in which names appear in the non-indexed string and the order in which they appear in the indexed query. There is also no defined relationship between any particular extension name and the *index* values; an extension name may correspond to a different *index* in different GL contexts and/or implementations.

An `INVALID_VALUE` error is generated if *index* is outside the valid range for the indexed state *target*.

6.1.12 Asynchronous Queries

The command

```
boolean IsQuery( uint id );
```

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:

$$n = \min\{32, \lceil \log_2(\text{maxViewportWidth} \times \text{maxViewportHeight} \times 2) \rceil\}.$$

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[ui]v**, the result and availability information returned will always be from the last query issued. The results from any queries before the last one will be lost if they are not retrieved before starting a new query on the same *target* and *id*.

6.1.13 Buffer Object Queries

The command

```
boolean IsBuffer( uint buffer );
```

returns `TRUE` if *buffer* is the name of a buffer object. If *buffer* is zero, or if *buffer* is a non-zero value that is not the name of a buffer object, **IsBuffer** returns `FALSE`.

The command

```
void GetBufferSubData( enum target, intptr offset,  
                        sizeiptr size, void *data );
```

queries the data contents of a buffer object. *target* is `ARRAY_BUFFER`, `ELEMENT_ARRAY_BUFFER`, `PIXEL_PACK_BUFFER`, or `PIXEL_UNPACK_BUFFER`. *offset* and *size* indicate the range of data in the buffer object that is to be queried, in terms of basic machine units. *data* specifies a region of client memory, *size* basic machine units in length, into which the data is to be retrieved.

An error is generated if **GetBufferSubData** is executed for a buffer object that is currently mapped.

While the data store of a buffer object is mapped, the pointer to the data store can be queried by calling

```
void GetBufferPointerv( enum target, enum pname,  
                        void **params );
```

with *target* set to `ARRAY_BUFFER`, `ELEMENT_ARRAY_BUFFER`, `PIXEL_PACK_BUFFER`, or `PIXEL_UNPACK_BUFFER` and *pname* set to `BUFFER_MAP_POINTER`. The single buffer map pointer is returned in **params*. **GetBufferPointerv** returns the `NULL` pointer value if the buffer's data store is not currently mapped, or if the requesting client did not map the buffer object's data store, and the implementation is unable to support mappings on multiple clients.

To query which buffer objects are bound to the array of transform feedback binding points and will be used when transform feedback is active, call **GetIntegeriv** with *param* set to `TRANSFORM_FEEDBACK_BUFFER_BINDING`. *index* must be in the range zero to the value of `MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS - 1`. The name of the buffer object bound to *index* is returned in *values*. If no buffer object is bound for *index*, zero is returned in *values*.

To query the starting offset or size of the range of each buffer object binding used for transform feedback, call **GetIntegeriv** with *param* set to `TRANSFORM_FEEDBACK_BUFFER_START` or `TRANSFORM_FEEDBACK_BUFFER_SIZE` respectively. *index* must be in the range 0 to the value of `MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS - 1`. If the parameter (starting offset or size) was not specified when the buffer object was bound, zero is returned. If no buffer object is bound to *index*, -1 is returned.

6.1.14 Vertex Array Object Queries

The command

```
boolean IsVertexArray( uint array );
```

returns `TRUE` if *array* is the name of a vertex array object previously returned by **GenVertexArrays**. If *array* is zero, or a non-zero value that is not the name of a vertex array object, **IsVertexArray** returns `FALSE`. No error is generated if *array* is not a valid vertex array object name.

6.1.15 Shader and Program Queries

State stored in shader or program objects can be queried by commands that accept shader or program object names. These commands will generate the error `INVALID_VALUE` if the provided name is not the name of either a shader or program object, and `INVALID_OPERATION` if the provided name identifies an object of the other type. If an error is generated, variables used to hold return values are not modified.

The command

```
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, 0 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, 0 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, 0 is returned. If *pname* is `ACTIVE_UNIFORMS`, the number of active uniforms is returned. If no active uniforms exist, 0 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, 0 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.

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 GetVertexAttribliv( 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 error INVALID_OPERATION is generated if *index* is zero, as there is no current value for generic attribute zero.

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* for program object *program* in the array *params*. The type of the uniform at *location* determines the number of values returned. The error `INVALID_OPERATION` is generated if *program* has not been linked successfully, or if *location* is not a valid location for *program*. In order to query the values of an array of uniforms, a **GetUniform*** command needs to be issued for each array element. If the uniform queried is a matrix, the values of the matrix are returned in column major order. If an error occurred, the return parameter *params* will be unmodified.

6.1.16 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`.

6.1.17 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`.

6.1.18 Saving and Restoring State

Besides providing a means to obtain the values of state variables, the GL also provides a means to save and restore groups of state variables. The **PushAttrib**, **PushClientAttrib**, **PopAttrib** and **PopClientAttrib** commands are used for this purpose. The commands

```
void PushAttrib( bitfield mask );  
void PushClientAttrib( bitfield mask );
```

take a bitwise OR of symbolic constants indicating which groups of state variables to push onto an attribute stack. **PushAttrib** uses a server attribute stack while **PushClientAttrib** uses a client attribute stack. Each constant refers to a group of state variables. The classification of each variable into a group is indicated in the following tables of state variables. The error `STACK_OVERFLOW` is generated if **PushAttrib** or **PushClientAttrib** is executed while the corresponding stack depth is `MAX_ATTRIB_STACK_DEPTH` or `MAX_CLIENT_ATTRIB_STACK_DEPTH` respectively. Bits set in *mask* that do not correspond to an attribute group are ignored. The special *mask* values `ALL_ATTRIB_BITS` and `CLIENT_ALL_ATTRIB_BITS` may be used to push all stackable server and client state, respectively.

The commands

```
void PopAttrib(void);
void PopClientAttrib(void);
```

reset the values of those state variables that were saved with the last corresponding **PushAttrib** or **PopClientAttrib**. Those not saved remain unchanged. The error `STACK_UNDERFLOW` is generated if **PopAttrib** or **PopClientAttrib** is executed while the respective stack is empty.

Table 6.2 shows the attribute groups with their corresponding symbolic constant names and stacks.

When **PushAttrib** is called with `TEXTURE_BIT` set, the priorities, border colors, filter modes, wrap modes, and other state of the currently bound texture objects (see table 6.20), as well as the current texture bindings and enables, are pushed onto the attribute stack. (Unbound texture objects are not pushed or restored.) When an attribute set that includes texture information is popped, the bindings and enables are first restored to their pushed values, then the bound texture object's parameters are restored to their pushed values.

Operations on attribute groups push or pop texture state within that group for all texture units. When state for a group is pushed, all state corresponding to `TEXTURE0` is pushed first, followed by state corresponding to `TEXTURE1`, and so on up to and including the state corresponding to `TEXTURE k` where $k + 1$ is the value of `MAX_TEXTURE_UNITS`. When state for a group is popped, texture state is restored in the opposite order that it was pushed, starting with state corresponding to `TEXTURE k` and ending with `TEXTURE0`. Identical rules are observed for client texture state push and pop operations. Matrix stacks are never pushed or popped with **PushAttrib**, **PushClientAttrib**, **PopAttrib**, or **PopClientAttrib**.

The depth of each attribute stack is implementation dependent but must be at least 16. The state required for each attribute stack is potentially 16 copies of each state variable, 16 masks indicating which groups of variables are stored in each

Stack	Attribute	Constant
server	accum-buffer	ACCUM_BUFFER_BIT
server	color-buffer	COLOR_BUFFER_BIT
server	current	CURRENT_BIT
server	depth-buffer	DEPTH_BUFFER_BIT
server	enable	ENABLE_BIT
server	eval	EVAL_BIT
server	fog	FOG_BIT
server	hint	HINT_BIT
server	lighting	LIGHTING_BIT
server	line	LINE_BIT
server	list	LIST_BIT
server	multisample	MULTISAMPLE_BIT
server	pixel	PIXEL_MODE_BIT
server	point	POINT_BIT
server	polygon	POLYGON_BIT
server	polygon-stipple	POLYGON_STIPPLE_BIT
server	scissor	SCISSOR_BIT
server	stencil-buffer	STENCIL_BUFFER_BIT
server	texture	TEXTURE_BIT
server	transform	TRANSFORM_BIT
server	viewport	VIEWPORT_BIT
server		ALL_ATTRIB_BITS
client	vertex-array	CLIENT_VERTEX_ARRAY_BIT
client	pixel-store	CLIENT_PIXEL_STORE_BIT
client	select	can't be pushed or pop'd
client	feedback	can't be pushed or pop'd
client		CLIENT_ALL_ATTRIB_BITS

Table 6.2: Attribute groups

stack entry, and an attribute stack pointer. In the initial state, both attribute stacks are empty.

In the tables that follow, a type is indicated for each variable. Table 6.3 explains these types. The type actually identifies all state associated with the indicated description; in certain cases only a portion of this state is returned. This is the case with all matrices, where only the top entry on the stack is returned; with clip planes, where only the selected clip plane is returned, with parameters describing lights, where only the value pertaining to the selected light is returned; with textures, where only the selected texture or texture parameter is returned; and with evaluator maps, where only the selected map is returned. Finally, a “–” in the attribute column indicates that the indicated value is not included in any attribute group (and thus can not be pushed or popped with **PushAttrib**, **PushClientAttrib**, **PopAttrib**, or **PopClientAttrib**).

The M and m entries for initial minmax table values represent the maximum and minimum possible representable values, respectively.

6.2 State Tables

The tables on the following pages indicate which state variables are obtained with what commands. State variables that can be obtained using any of **GetBooleanv**, **GetIntegerv**, **GetFloatv**, or **GetDoublev** are listed with just one of these commands – the one that is most appropriate given the type of the data to be returned. These state variables cannot be obtained using **IsEnabled**. However, state variables for which **IsEnabled** is listed as the query command can also be obtained using **GetBooleanv**, **GetIntegerv**, **GetFloatv**, and **GetDoublev**. State variables for which any other command is listed as the query command can be obtained only by using that command.

State table entries which are required only by the imaging subset (see section 3.7.2) are typeset against a gray background.

Type code	Explanation
B	Boolean
BMU	Basic machine units
C	Color (floating-point R, G, B, and A values)
CI	Color index (floating-point index value)
T	Texture coordinates (floating-point (s, t, r, q) values)
N	Normal coordinates (floating-point (x, y, z) values)
V	Vertex, including associated data
Z	Integer
Z^+	Non-negative integer
Z_k, Z_{k*}	k -valued integer ($k*$ indicates k is minimum)
R	Floating-point number
R^+	Non-negative floating-point number
$R^{[a,b]}$	Floating-point number in the range $[a, b]$
R^k	k -tuple of floating-point numbers
P	Position ((x, y, z, w) floating-point coordinates)
D	Direction ((x, y, z) floating-point coordinates)
M^4	4×4 floating-point matrix
S	NULL-terminated string
I	Image
A	Attribute stack entry, including mask
Y	Pointer (data type unspecified)
$n \times type$	n copies of type $type$ ($n*$ indicates n is minimum)

Table 6.3: State Variable Types

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
-	Z_{11}	-	0	When $\neq 0$, indicates begin/end object	2.6.1	-
-	V	-	-	Previous vertex in Begin/End line	2.6.1	-
-	B	-	-	Indicates if <i>line-vertex</i> is the first	2.6.1	-
-	V	-	-	First vertex of a Begin/End line loop	2.6.1	-
-	Z^+	-	-	Line stipple counter	3.5	-
-	$n \times V$	-	-	Vertices inside of Begin/End polygon	2.6.1	-
-	Z^+	-	-	Number of <i>polygon-vertices</i>	2.6.1	-
-	$2 \times V$	-	-	Previous two vertices in a Begin/End triangle strip	2.6.1	-
-	Z_3	-	-	Number of vertices so far in triangle strip: 0, 1, or more	2.6.1	-
-	Z_2	-	-	Triangle strip A/B vertex pointer	2.6.1	-
-	$3 \times V$	-	-	Vertices of the quad under construction	2.6.1	-
-	Z_4	-	-	Number of vertices so far in quad strip: 0, 1, 2, or more	2.6.1	-

Table 6.4. GL Internal begin-end state variables (inaccessible)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CURRENT.COLOR	C	GetFloatv	1,1,1,1	Current color	2.7	current
CURRENT.SECONDARY.COLOR	C'	GetFloatv	0,0,0,1	Current secondary color	2.7	current
CURRENT.INDEX	CI	GetIntegerv	1	Current color index	2.7	current
CURRENT.TEXTURE.COORDS	$2 * \times T$	GetFloatv	0,0,0,1	Current texture coordinates	2.7	current
CURRENT.NORMAL	N	GetFloatv	0,0,1	Current normal	2.7	current
CURRENT.FOG.COORD	R	GetFloatv	0	Current fog coordinate	2.7	current
-	C	-	-	Color associated with last vertex	2.6	-
-	CI	-	-	Color index associated with last vertex	2.6	-
-	T	-	-	Texture coordinates associated with last vertex	2.6	-
CURRENT.RASTER.POSITION	R^4	GetFloatv	0,0,0,1	Current raster position	2.18	current
CURRENT.RASTER.DISTANCE	R^+	GetFloatv	0	Current raster distance	2.18	current
CURRENT.RASTER.COLOR	C	GetFloatv	1,1,1,1	Color associated with raster position	2.18	current
CURRENT.RASTER.SECONDARY.COLOR	C'	GetFloatv	0,0,0,1	Secondary color associated with raster position	2.18	current
CURRENT.RASTER.INDEX	CI	GetIntegerv	1	Color index associated with raster position	2.18	current
CURRENT.RASTER.TEXTURE.COORDS	$2 * \times T$	GetFloatv	0,0,0,1	Texture coordinates associated with raster position	2.18	current
CURRENT.RASTER.POSITION.VALID	B	GetBooleanv	TRUE	Raster position valid bit	2.18	current
EDGE.FLAG	B	GetBooleanv	TRUE	Edge flag	2.6.2	current

Table 6.5. Current Values and Associated Data

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_ARRAY	B	IsEnabled	FALSE	Vertex array enable	2.8	vertex-array
VERTEX_ARRAY_SIZE	Z^+	GetIntegerv	4	Coordinates per vertex	2.8	vertex-array
VERTEX_ARRAY_TYPE	Z_4	GetIntegerv	FLOAT	Type of vertex coordinates	2.8	vertex-array
VERTEX_ARRAY_STRIDE	Z^+	GetIntegerv	0	Stride between vertices	2.8	vertex-array
VERTEX_ARRAY_POINTER	Y	GetPointerv	0	Pointer to the vertex array	2.8	vertex-array
NORMAL_ARRAY	B	IsEnabled	FALSE	Normal array enable	2.8	vertex-array
NORMAL_ARRAY_TYPE	Z_5	GetIntegerv	FLOAT	Type of normal coordinates	2.8	vertex-array
NORMAL_ARRAY_STRIDE	Z^+	GetIntegerv	0	Stride between normals	2.8	vertex-array
NORMAL_ARRAY_POINTER	Y	GetPointerv	0	Pointer to the normal array	2.8	vertex-array
FOG_COORD_ARRAY	B	IsEnabled	FALSE	Fog coord array enable	2.8	vertex-array
FOG_COORD_ARRAY_TYPE	Z_2	GetIntegerv	FLOAT	Type of fog coord components	2.8	vertex-array
FOG_COORD_ARRAY_STRIDE	Z^+	GetIntegerv	0	Stride between fog coords	2.8	vertex-array
FOG_COORD_ARRAY_POINTER	Y	GetPointerv	0	Pointer to the fog coord array	2.8	vertex-array
COLOR_ARRAY	B	IsEnabled	FALSE	Color array enable	2.8	vertex-array
COLOR_ARRAY_SIZE	Z^+	GetIntegerv	4	Color components per vertex	2.8	vertex-array
COLOR_ARRAY_TYPE	Z_8	GetIntegerv	FLOAT	Type of color components	2.8	vertex-array
COLOR_ARRAY_STRIDE	Z^+	GetIntegerv	0	Stride between colors	2.8	vertex-array
COLOR_ARRAY_POINTER	Y	GetPointerv	0	Pointer to the color array	2.8	vertex-array

Table 6.6. Vertex Array Object State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SECONDARY_COLOR_ARRAY	B	IsEnabled	FALSE	Secondary color array enable	2.8	vertex-array
SECONDARY_COLOR_ARRAY_SIZE	Z^+	GetIntegerv	3	Secondary color components per vertex	2.8	vertex-array
SECONDARY_COLOR_ARRAY_TYPE	Z_8	GetIntegerv	FLOAT	Type of secondary color components	2.8	vertex-array
SECONDARY_COLOR_ARRAY_STRIDE	Z^+	GetIntegerv	0	Stride between secondary colors	2.8	vertex-array
SECONDARY_COLOR_ARRAY_POINTER	Y	GetPointerv	0	Pointer to the secondary color array	2.8	vertex-array
INDEX_ARRAY	B	IsEnabled	FALSE	Index array enable	2.8	vertex-array
INDEX_ARRAY_TYPE	Z_4	GetIntegerv	FLOAT	Type of indices	2.8	vertex-array
INDEX_ARRAY_STRIDE	Z^+	GetIntegerv	0	Stride between indices	2.8	vertex-array
INDEX_ARRAY_POINTER	Y	GetPointerv	0	Pointer to the index array	2.8	vertex-array
TEXTURE_COORD_ARRAY	$2 * \times B$	IsEnabled	FALSE	Texture coordinate array enable	2.8	vertex-array
TEXTURE_COORD_ARRAY_SIZE	$2 * \times Z^+$	GetIntegerv	4	Coordinates per element	2.8	vertex-array
TEXTURE_COORD_ARRAY_TYPE	$2 * \times Z_4$	GetIntegerv	FLOAT	Type of texture coordinates	2.8	vertex-array
TEXTURE_COORD_ARRAY_STRIDE	$2 * \times Z^+$	GetIntegerv	0	Stride between texture coordinates	2.8	vertex-array
TEXTURE_COORD_ARRAY_POINTER	$2 * \times Y$	GetPointerv	0	Pointer to the texture coordinate array	2.8	vertex-array

Table 6.7. Vertex Array Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_ATTRIB_ARRAY_ENABLED	$16 * \times B$	GetVertexAttrib	FALSE	Vertex attrib array enable	2.8	vertex-array
VERTEX_ATTRIB_ARRAY_SIZE	$16 * \times Z$	GetVertexAttrib	4	Vertex attrib array size	2.8	vertex-array
VERTEX_ATTRIB_ARRAY_STRIDE	$16 * \times Z^+$	GetVertexAttrib	0	Vertex attrib array stride	2.8	vertex-array
VERTEX_ATTRIB_ARRAY_TYPE	$16 * \times Z_4$	GetVertexAttrib	FLOAT	Vertex attrib array type	2.8	vertex-array
VERTEX_ATTRIB_ARRAY_NORMALIZED	$16 * \times B$	GetVertexAttrib	FALSE	Vertex attrib array normalized	2.8	vertex-array
VERTEX_ATTRIB_ARRAY_INTEGER	$16 * \times B$	GetVertexAttrib	FALSE	Vertex attrib array has unconverted integers	2.8	vertex-array
VERTEX_ATTRIB_ARRAY_POINTER	$16 * \times Y$	GetVertexAttribPointer	NULL	Vertex attrib array pointer	2.8	vertex-array
EDGE_FLAG_ARRAY	B	IsEnabled	FALSE	Edge flag array enable	2.8	vertex-array
EDGE_FLAG_ARRAY_STRIDE	Z^+	GetIntegerv	0	Stride between edge flags	2.8	vertex-array
EDGE_FLAG_ARRAY_POINTER	Y	GetPointerv	0	Pointer to the edge flag array	2.8	vertex-array

Table 6.8. Vertex Array Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_ARRAY_BUFFER_BINDING	Z^+	GetIntegerv	0	Vertex array buffer binding	2.9	vertex-array
NORMAL_ARRAY_BUFFER_BINDING	Z^+	GetIntegerv	0	Normal array buffer binding	2.9	vertex-array
COLOR_ARRAY_BUFFER_BINDING	Z^+	GetIntegerv	0	Color array buffer binding	2.9	vertex-array
INDEX_ARRAY_BUFFER_BINDING	Z^+	GetIntegerv	0	Index array buffer binding	2.9	vertex-array
TEXTURE_COORD_ARRAY_BUFFER_BINDING	$2 * \times Z^+$	GetIntegerv	0	Texcoord array buffer binding	2.9	vertex-array
EDGE_FLAG_ARRAY_BUFFER_BINDING	Z^+	GetIntegerv	0	Edge flag array buffer binding	2.9	vertex-array
SECONDARY_COLOR_ARRAY_BUFFER_BINDING	Z^+	GetIntegerv	0	Secondary color array buffer binding	2.9	vertex-array
FOG_COORD_ARRAY_BUFFER_BINDING	Z^+	GetIntegerv	0	Fog coordinate array buffer binding	2.9	vertex-array
ELEMENT_ARRAY_BUFFER_BINDING	Z^+	GetIntegerv	0	Element array buffer binding	2.9.2	vertex-array
VERTEX_ATTRIB_ARRAY_BUFFER_BINDING	$16 * \times Z^+$	GetVertexAttribiv	0	Attribute array buffer binding	2.9	vertex-array
VERTEX_ARRAY_BINDING	Z^+	GetIntegerv	0	Current vertex array object binding	2.10	vertex-array

Table 6.9. Vertex Array Object State (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CLIENT_ACTIVE_TEXTURE	Z_{2^*}	GetIntegerv	TEXTURE0	Client active texture unit selector	2.7	vertex-array
ARRAY_BUFFER_BINDING	Z^+	GetIntegerv	0	Current buffer binding	2.9	vertex-array

Table 6.10. Vertex Array Data (not in Vertex Array objects)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
-	$n \times BMU$	GetBufferSubData	-	buffer data	2.9	-
BUFFER_SIZE	$n \times Z^+$	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_MAPPED	$n \times B$	GetBufferParameteriv	FALSE	Buffer map flag	2.9	-
BUFFER_MAP_POINTER	$n \times Y$	GetBufferPointeriv	NULL	Mapped buffer pointer	2.9	-

Table 6.11. Buffer Object State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
COLOR_MATRIX (TRANPOSE.COLOR_MATRIX)	$2 * \times M^4$	GetFloatv	Identity	Color matrix stack	3.7.3	–
MODELVIEW_MATRIX (TRANPOSE.MODELVIEW_MATRIX)	$32 * \times M^4$	GetFloatv	Identity	Model-view matrix stack	2.12.2	–
PROJECTION_MATRIX (TRANPOSE.PROJECTION_MATRIX)	$2 * \times M^4$	GetFloatv	Identity	Projection matrix stack	2.12.2	–
TEXTURE_MATRIX (TRANPOSE.TEXTURE_MATRIX)	$2 * \times 2 * \times M^4$	GetFloatv	Identity	Texture matrix stack	2.12.2	–
VIEWPORT	$4 \times Z$	GetIntegerv	see 2.12.1	Viewport origin & extent	2.12.1	viewport
DEPTH_RANGE	$2 \times R^+$	GetFloatv	0,1	Depth range near & far	2.12.1	viewport
COLOR_MATRIX_STACK_DEPTH	Z^+	GetIntegerv	1	Color matrix stack pointer	3.7.3	–
MODELVIEW_STACK_DEPTH	Z^+	GetIntegerv	1	Model-view matrix stack pointer	2.12.2	–
PROJECTION_STACK_DEPTH	Z^+	GetIntegerv	1	Projection matrix stack pointer	2.12.2	–
TEXTURE_STACK_DEPTH	$2 * \times Z^+$	GetIntegerv	1	Texture matrix stack pointer	2.12.2	–
MATRIX_MODE	Z_4	GetIntegerv	MODELVIEW	Current matrix mode	2.12.2	transform
NORMALIZE	B	IsEnabled	FALSE	Current normal normalization on/off	2.12.3	transform/enable
RESCALE_NORMAL	B	IsEnabled	FALSE	Current normal rescaling on/off	2.12.3	transform/enable
CLIP_PLANE _{<i>i</i>}	$6 * \times R^4$	GetClipPlane	0,0,0,0	User clipping plane coefficients	2.17	transform
CLIP_PLANE _{<i>i</i>}	$6 * \times B$	IsEnabled	FALSE	<i>i</i> th user clipping plane enabled	2.17	transform/enable

Table 6.12, Transformation state
Version 3.0 - August 11, 2008

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
FOG_COLOR	C	GetFloatv	0.0,0.0	Fog color	3.11	fog
FOG_INDEX	CI	GetFloatv	0	Fog index	3.11	fog
FOG_DENSITY	R	GetFloatv	1.0	Exponential fog density	3.11	fog
FOG_START	R	GetFloatv	0.0	Linear fog start	3.11	fog
FOG_END	R	GetFloatv	1.0	Linear fog end	3.11	fog
FOG_MODE	Z_3	GetIntegerv	EXP	Fog mode	3.11	fog
FOG	B	IsEnabled	FALSE	True if fog enabled	3.11	fog/enable
FOG_COORD_SRC	Z_2	GetIntegerv	FRAGMENT_DEPTH	Source of coordinate for fog calculation	3.11	fog
COLOR_SUM	B	IsEnabled	FALSE	True if color sum enabled	3.10	fog/enable
SHADE_MODEL	Z^+	GetIntegerv	SMOOTH	ShadeModel setting	2.19.7	lighting
CLAMP_VERTEX_COLOR	Z_3	GetIntegerv	TRUE	Vertex color clamping	2.19.6	lighting/enable
CLAMP_FRAGMENT_COLOR	Z_3	GetIntegerv	FIXED_ONLY	Fragment color clamping	3.8	color-buffer/enable
CLAMP_READ_COLOR	Z_3	GetIntegerv	FIXED_ONLY	Read color clamping	4.2	color-buffer/enable

Table 6.13. Coloring

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
LIGHTING	B	IsEnabled	FALSE	True if lighting is enabled	2.19.1	lighting/enable
COLOR_MATERIAL	B	IsEnabled	FALSE	True if color tracking is enabled	2.19.3	lighting/enable
COLOR_MATERIAL_PARAMETER	Z_5	GetIntegerv	AMBIENT_AND_DIFFUSE	Material properties tracking current color	2.19.3	lighting
COLOR_MATERIAL_FACE	Z_3	GetIntegerv	FRONT_AND_BACK	Face(s) affected by color tracking	2.19.3	lighting
AMBIENT	$2 \times C$	GetMaterialfv	(0.2,0.2,0.2,1.0)	Ambient material color	2.19.1	lighting
DIFFUSE	$2 \times C$	GetMaterialfv	(0.8,0.8,0.8,1.0)	Diffuse material color	2.19.1	lighting
SPECULAR	$2 \times C$	GetMaterialfv	(0.0,0.0,0.0,1.0)	Specular material color	2.19.1	lighting
EMISSION	$2 \times C$	GetMaterialfv	(0.0,0.0,0.0,1.0)	Emissive mat. color	2.19.1	lighting
SHININESS	$2 \times R$	GetMaterialfv	0.0	Specular exponent of material	2.19.1	lighting
LIGHT_MODEL_AMBIENT	C	GetFloatv	(0.2,0.2,0.2,1.0)	Ambient scene color	2.19.1	lighting
LIGHT_MODEL_LOCAL_VIEWER	B	GetBooleanv	FALSE	Viewer is local	2.19.1	lighting
LIGHT_MODEL_TWO_SIDE	B	GetBooleanv	FALSE	Use two-sided lighting	2.19.1	lighting
LIGHT_MODEL_COLOR_CONTROL	Z_2	GetIntegerv	SINGLE_COLOR	Color control	2.19.1	lighting

Table 6.14. Lighting (see also table 2.11 for defaults)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
AMBIENT	$8 * \times C$	GetLightfv	(0.0,0.0,0.0,1.0)	Ambient intensity of light i	2.19.1	lighting
DIFFUSE	$8 * \times C$	GetLightfv	see table 2.11	Diffuse intensity of light i	2.19.1	lighting
SPECULAR	$8 * \times C$	GetLightfv	see table 2.11	Specular intensity of light i	2.19.1	lighting
POSITION	$8 * \times P$	GetLightfv	(0.0,0.0,1.0,0.0)	Position of light i	2.19.1	lighting
CONSTANT_ATTENUATION	$8 * \times R^+$	GetLightfv	1.0	Constant atten. factor	2.19.1	lighting
LINEAR_ATTENUATION	$8 * \times R^+$	GetLightfv	0.0	Linear atten. factor	2.19.1	lighting
QUADRATIC_ATTENUATION	$8 * \times R^+$	GetLightfv	0.0	Quadratic atten. factor	2.19.1	lighting
SPOT_DIRECTION	$8 * \times D$	GetLightfv	(0.0,0.0,-1.0)	Spotlight direction of light i	2.19.1	lighting
SPOT_EXPONENT	$8 * \times R^+$	GetLightfv	0.0	Spotlight exponent of light i	2.19.1	lighting
SPOT_CUTOFF	$8 * \times R^+$	GetLightfv	180.0	Spot. angle of light i	2.19.1	lighting
LIGHT $_i$	$8 * \times B$	IsEnabled	FALSE	True if light i enabled	2.19.1	lighting/enable
COLORINDEXES	$2 \times 3 \times R$	GetMaterialfv	0,1,1	a_m , d_m , and s_m for color index lighting	2.19.1	lighting

Table 6.15. Lighting (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
POINT_SIZE	R^+	GetFloatv	1.0	Point size	3.4	point
POINT_SMOOTH	B	IsEnabled	FALSE	Point antialiasing on	3.4	point/enable
POINT_SPRITE	B	IsEnabled	FALSE	Point sprite enable	3.4	point/enable
POINT_SIZE_MIN	R^+	GetFloatv	0.0	Attenuated minimum point size	3.4	point
POINT_SIZE_MAX	R^+	GetFloatv	1	Attenuated maximum point size. ¹ Max. of the impl. dependent max. aliased and smooth point sizes.	3.4	point
POINT_FADE_THRESHOLD_SIZE	R^+	GetFloatv	1.0	Threshold for alpha attenuation	3.4	point
POINT_DISTANCE_ATTENUATION	$3 \times R^+$	GetFloatv	1,0,0	Attenuation coefficients	3.4	point
POINT_SPRITE_COORD_ORIGIN	Z_2	GetIntegerv	UPPER_LEFT	Origin orientation for point sprites	3.4	point
LINE_WIDTH	R^+	GetFloatv	1.0	Line width	3.5	line
LINE_SMOOTH	B	IsEnabled	FALSE	Line antialiasing on	3.5	line/enable
LINE_STIPPLE_PATTERN	Z^+	GetIntegerv	1's	Line stipple	3.5.2	line
LINE_STIPPLE_REPEAT	Z^+	GetIntegerv	1	Line stipple repeat	3.5.2	line
LINE_STIPPLE	B	IsEnabled	FALSE	Line stipple enable	3.5.2	line/enable

Table 6.16. Rasterization

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CULL_FACE	B	IsEnabled	FALSE	Polygon culling enabled	3.6.1	polygon/enable
CULL_FACE_MODE	Z_3	GetIntegerv	BACK	Cull front/back facing polygons	3.6.1	polygon
FRONT_FACE	Z_2	GetIntegerv	CCW	Polygon frontface CW/CCW indicator	3.6.1	polygon
POLYGON_SMOOTH	B	IsEnabled	FALSE	Polygon antialiasing on	3.6	polygon/enable
POLYGON_MODE	$2 \times Z_3$	GetIntegerv	FILL	Polygon rasterization mode (front & back)	3.6.4	polygon
POLYGON_OFFSET_FACTOR	R	GetFloatv	0	Polygon offset factor	3.6.5	polygon
POLYGON_OFFSET_UNITS	R	GetFloatv	0	Polygon offset units	3.6.5	polygon
POLYGON_OFFSET_POINT	B	IsEnabled	FALSE	Polygon offset enable for POINT mode rasterization	3.6.5	polygon/enable
POLYGON_OFFSET_LINE	B	IsEnabled	FALSE	Polygon offset enable for LINE mode rasterization	3.6.5	polygon/enable
POLYGON_OFFSET_FILL	B	IsEnabled	FALSE	Polygon offset enable for FILL mode rasterization	3.6.5	polygon/enable
-	I	GetPolygonStipple	1's	Polygon stipple	3.6	polygon-stipple
POLYGON_STIPPLE	B	IsEnabled	FALSE	Polygon stipple enable	3.6.2	polygon/enable

Table 6.17. Rasterization (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
MULTISAMPLE	B	IsEnabled	TRUE	Multisample rasterization	3.3.1	multisample/enable
SAMPLE.ALPHA.TO.COVERAGE	B	IsEnabled	FALSE	Modify coverage from alpha	4.1.3	multisample/enable
SAMPLE.ALPHA.TO.ONE	B	IsEnabled	FALSE	Set alpha to maximum	4.1.3	multisample/enable
SAMPLE.COVERAGE	B	IsEnabled	FALSE	Mask to modify coverage	4.1.3	multisample/enable
SAMPLE.COVERAGE.VALUE	R^+	GetFloatv	1	Coverage mask value	4.1.3	multisample
SAMPLE.COVERAGE.INVERT	B	GetBooleanv	FALSE	Invert coverage mask value	4.1.3	multisample

Table 6.18. Multisampling

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE.xD	$2 * \times 3 \times B$	IsEnabled	FALSE	True if xD texturing is enabled; x is 1, 2, or 3	3.9.17	texture/enable
TEXTURE_CUBE_MAP	$2 * \times B$	IsEnabled	FALSE	True if cube map texturing is enabled	3.9.13	texture/enable
TEXTURE_BINDING.xD	$2 * \times 3 \times Z^+$	GetInteger	0	Texture object bound to TEXTURE.xD	3.9.12	texture
TEXTURE_BINDING_ID_ARRAY	$2 * \times 3 \times Z^+$	GetInteger	0	Texture object bound to TEXTURE_1D_ARRAY	3.9.12	texture
TEXTURE_BINDING_2D_ARRAY	$2 * \times 3 \times Z^+$	GetInteger	0	Texture object bound to TEXTURE_2D_ARRAY	3.9.12	texture
TEXTURE_BINDING_CUBE_MAP	$2 * \times Z^+$	GetInteger	0	Texture object bound to TEXTURE_CUBE_MAP	3.9.11	texture
TEXTURE.xD	$n \times I$	GetTexImage	see 3.9	xD texture image at l.o.d. i	3.9	–
TEXTURE_CUBE_MAP_POSITIVE_X	$n \times I$	GetTexImage	see 3.9.1	+x face cube map texture image at l.o.d. i	3.9.1	–
TEXTURE_CUBE_MAP_NEGATIVE_X	$n \times I$	GetTexImage	see 3.9.1	–x face cube map texture image at l.o.d. i	3.9.1	–
TEXTURE_CUBE_MAP_POSITIVE_Y	$n \times I$	GetTexImage	see 3.9.1	+y face cube map texture image at l.o.d. i	3.9.1	–
TEXTURE_CUBE_MAP_NEGATIVE_Y	$n \times I$	GetTexImage	see 3.9.1	–y face cube map texture image at l.o.d. i	3.9.1	–
TEXTURE_CUBE_MAP_POSITIVE_Z	$n \times I$	GetTexImage	see 3.9.1	+z face cube map texture image at l.o.d. i	3.9.1	–
TEXTURE_CUBE_MAP_NEGATIVE_Z	$n \times I$	GetTexImage	see 3.9.1	–z face cube map texture image at l.o.d. i	3.9.1	–

Table 6.19. Textures (state per texture unit and binding point)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE_BORDER_COLOR	$n \times C$	GetTexParameter	0,0,0,0	Border color	3.9	texture
TEXTURE_MIN_FILTER	$n \times Z_6$	GetTexParameter	see 3.9	Minification function	3.9.7	texture
TEXTURE_MAG_FILTER	$n \times Z_2$	GetTexParameter	see 3.9	Magnification function	3.9.8	texture
TEXTURE_WRAP_S	$n \times Z_5$	GetTexParameter	REPEAT	Texture coord s wrap mode	3.9.7	texture
TEXTURE_WRAP_T	$n \times Z_5$	GetTexParameter	REPEAT	Texture coord t wrap mode (2D, 3D, cube map textures only)	3.9.7	texture
TEXTURE_WRAP_R	$n \times Z_5$	GetTexParameter	REPEAT	Texture coord r wrap mode (3D textures only)	3.9.7	texture
TEXTURE_PRIORITY	$n \times R^{[0,1]}$	GetTexParameterfv	1	Texture object priority	3.9.12	texture
TEXTURE_RESIDENT	$n \times B$	GetTexParameteriv	see 3.9.12	Texture residency	3.9.12	texture
TEXTURE_MIN_LOD	$n \times R$	GetTexParameterfv	-1000	Minimum level of detail	3.9	texture
TEXTURE_MAX_LOD	$n \times R$	GetTexParameterfv	1000	Maximum level of detail	3.9	texture
TEXTURE_BASE_LEVEL	$n \times Z^+$	GetTexParameterfv	0	Base texture array	3.9	texture
TEXTURE_MAX_LEVEL	$n \times Z^+$	GetTexParameterfv	1000	Max. texture array level	3.9	texture
TEXTURE_LOD_BIAS	$n \times R$	GetTexParameterfv	0.0	Texture level of detail bias ($bias_{texobj}$)	3.9.7	texture
DEPTH_TEXTURE_MODE	$n \times Z_3$	GetTexParameteriv	LUMINANCE	Depth texture mode	3.9.5	texture
TEXTURE_COMPARE_MODE	$n \times Z_2$	GetTexParameteriv	NONE	Comparison mode	3.9.14	texture
TEXTURE_COMPARE_FUNC	$n \times Z_8$	GetTexParameteriv	LEQUAL	Comparison function	3.9.14	texture
GENERATE_MIPMAP	$n \times B$	GetTexParameter	FALSE	Automatic mipmap generation enabled	3.9.7	texture

Table 6.20. Textures (state per texture object)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
TEXTURE.WIDTH	$n \times Z^+$	GetTexLevelParameter	0	Specified width	3.9	–
TEXTURE.HEIGHT	$n \times Z^+$	GetTexLevelParameter	0	Specified height (2D/3D)	3.9	–
TEXTURE.DEPTH	$n \times Z^+$	GetTexLevelParameter	0	Specified depth (3D)	3.9	–
TEXTURE.BORDER	$n \times Z^+$	GetTexLevelParameter	0	Specified border width	3.9	–
TEXTURE.INTERNAL_FORMAT (TEXTURE.COMPONENTS)	$n \times Z_{68*}$	GetTexLevelParameter	1	Internal format	3.9	–
TEXTURE.x.SIZE	$n \times 8 \times Z^+$	GetTexLevelParameter	0	Component resolution (x is RED, GREEN, BLUE, ALPHA, LUMINANCE, INTENSITY, DEPTH, or STENCIL)	3.9	–
TEXTURE.SHARED_SIZE	$n \times Z^+$	GetTexLevelParameter	0	Shared exponent field resolution	3.9	–
TEXTURE.x.TYPE	$n \times Z_5$	GetTexLevelParameter	NONE	Component type (x is RED, GREEN, BLUE, ALPHA, LUMINANCE, INTENSITY, or DEPTH)	6.1.3	–
TEXTURE.COMPRESSED	$n \times B$	GetTexLevelParameter	FALSE	True if image has a compressed internal format	3.9.3	–
TEXTURE.COMPRESSED.IMAGE.SIZE	$n \times Z^+$	GetTexLevelParameter	0	size (in bytes) of compressed image	3.9.3	–

Table 6.21. Textures (state per texture image)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
COORD.REPLACE	$2 * \times B$	GetTexEnviv	FALSE	Coordinate replacement enable	3.4	point
ACTIVE.TEXTURE	Z_{2*}	GetIntegerv	TEXTURE0	Active texture unit selector	2.7	texture
TEXTURE.ENV.MODE	$2 * \times Z_6$	GetTexEnviv	MODULATE	Texture application function	3.9.13	texture
TEXTURE.ENV.COLOR	$2 * \times C$	GetTexEnvfv	0,0,0,0	Texture environment color	3.9.13	texture
TEXTURE.LOD.BIAS	$2 * \times R$	GetTexEnvfv	0.0	Texture level of detail bias <i>bias_{texunit}</i>	3.9.7	texture
TEXTURE.GEN. <i>x</i>	$2 * \times 4 \times B$	IsEnabled	FALSE	Texgen enabled (<i>x</i> is S, T, R, or Q)	2.12.4	texture/enable
EYE.PLANE	$2 * \times 4 \times R^4$	GetTexGenfv	see 2.12.4	Texgen plane equation coefficients (for S, T, R, and Q)	2.12.4	texture
OBJECT.PLANE	$2 * \times 4 \times R^4$	GetTexGenfv	see 2.12.4	Texgen object linear coefficients (for S, T, R, and Q)	2.12.4	texture
TEXTURE.GEN.MODE	$2 * \times 4 \times Z_5$	GetTexGeniv	EYE.LINEAR	Function used for texgen (for S, T, R, and Q)	2.12.4	texture
COMBINE.RGB	$2 * \times Z_8$	GetTexEnviv	MODULATE	RGB combiner function	3.9.13	texture
COMBINE.ALPHA	$2 * \times Z_6$	GetTexEnviv	MODULATE	Alpha combiner function	3.9.13	texture

Table 6.22. Texture Environment and Generation

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SRC0.RGB	$2 * \times Z_3$	GetTexEnviv	TEXTURE	RGB source 0	3.9.13	texture
SRC1.RGB	$2 * \times Z_3$	GetTexEnviv	PREVIOUS	RGB source 1	3.9.13	texture
SRC2.RGB	$2 * \times Z_3$	GetTexEnviv	CONSTANT	RGB source 2	3.9.13	texture
SRC0.ALPHA	$2 * \times Z_3$	GetTexEnviv	TEXTURE	Alpha source 0	3.9.13	texture
SRC1.ALPHA	$2 * \times Z_3$	GetTexEnviv	PREVIOUS	Alpha source 1	3.9.13	texture
SRC2.ALPHA	$2 * \times Z_3$	GetTexEnviv	CONSTANT	Alpha source 2	3.9.13	texture
OPERAND0.RGB	$2 * \times Z_4$	GetTexEnviv	SRC_COLOR	RGB operand 0	3.9.13	texture
OPERAND1.RGB	$2 * \times Z_4$	GetTexEnviv	SRC_COLOR	RGB operand 1	3.9.13	texture
OPERAND2.RGB	$2 * \times Z_4$	GetTexEnviv	SRC_ALPHA	RGB operand 2	3.9.13	texture
OPERAND0.ALPHA	$2 * \times Z_2$	GetTexEnviv	SRC_ALPHA	Alpha operand 0	3.9.13	texture
OPERAND1.ALPHA	$2 * \times Z_2$	GetTexEnviv	SRC_ALPHA	Alpha operand 1	3.9.13	texture
OPERAND2.ALPHA	$2 * \times Z_2$	GetTexEnviv	SRC_ALPHA	Alpha operand 2	3.9.13	texture
RGB.SCALE	$2 * \times R^3$	GetTexEnvfv	1.0	RGB post-combiner scaling	3.9.13	texture
ALPHA.SCALE	$2 * \times R^3$	GetTexEnvfv	1.0	Alpha post-combiner scaling	3.9.13	texture

Table 6.23. Texture Environment and Generation (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SCISSOR.TEST	B	IsEnabled	FALSE	Scissoring enabled	4.1.2	scissor/enable
SCISSOR.BOX	$4 \times Z$	GetIntegerv	see 4.1.2	Scissor box	4.1.2	scissor
ALPHA.TEST	B	IsEnabled	FALSE	Alpha test enabled	4.1.4	color-buffer/enable
ALPHA.TEST.FUNC	Z_8	GetIntegerv	ALWAYS	Alpha test function	4.1.4	color-buffer
ALPHA.TEST.REF	R^+	GetIntegerv	0	Alpha test reference value	4.1.4	color-buffer
STENCIL.TEST	B	IsEnabled	FALSE	Stenciling enabled	4.1.5	stencil-buffer/enable
STENCIL.FUNC	Z_8	GetIntegerv	ALWAYS	Front stencil function	4.1.5	stencil-buffer
STENCIL.VALUE.MASK	Z^+	GetIntegerv	1's	Front stencil mask	4.1.5	stencil-buffer
STENCIL.REF	Z^+	GetIntegerv	0	Front stencil reference value	4.1.5	stencil-buffer
STENCIL.FAIL	Z_8	GetIntegerv	KEEP	Front stencil fail action	4.1.5	stencil-buffer
STENCIL.PASS.DEPTH.FAIL	Z_8	GetIntegerv	KEEP	Front stencil depth buffer fail action	4.1.5	stencil-buffer
STENCIL.PASS.DEPTH.PASS	Z_8	GetIntegerv	KEEP	Front stencil depth buffer pass action	4.1.5	stencil-buffer
STENCIL.BACK.FUNC	Z_8	GetIntegerv	ALWAYS	Back stencil function	4.1.5	stencil-buffer
STENCIL.BACK.VALUE.MASK	Z^+	GetIntegerv	1's	Back stencil mask	4.1.5	stencil-buffer
STENCIL.BACK.REF	Z^+	GetIntegerv	0	Back stencil reference value	4.1.5	stencil-buffer
STENCIL.BACK.FAIL	Z_8	GetIntegerv	KEEP	Back stencil fail action	4.1.5	stencil-buffer
STENCIL.BACK.PASS.DEPTH.FAIL	Z_8	GetIntegerv	KEEP	Back stencil depth buffer fail action	4.1.5	stencil-buffer
STENCIL.BACK.PASS.DEPTH.PASS	Z_8	GetIntegerv	KEEP	Back stencil depth buffer pass action	4.1.5	stencil-buffer

Table 6.24. Pixel Operations

Version 3.0 - August 11, 2008

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
DEPTH.TEST	B	IsEnabled	FALSE	Depth buffer enabled	4.1.6	depth-buffer/enable
DEPTH.FUNC	Z_8	GetIntegerv	LESS	Depth buffer test function	4.1.6	depth-buffer
BLEND	$1 * \times B$	IsEnabledi	FALSE	Blending enabled for draw buffer i	4.1.8	color-buffer/enable
BLEND_SRC.RGB (v1.3.BLEND_SRC)	Z_{15}	GetIntegerv	ONE	Blending source RGB function	4.1.8	color-buffer
BLEND_SRC.ALPHA	Z_{15}	GetIntegerv	ONE	Blending source A function	4.1.8	color-buffer
BLEND_DST.RGB (v1.3.BLEND_DST)	Z_{14}	GetIntegerv	ZERO	Blending dest. RGB function	4.1.8	color-buffer
BLEND_DST.ALPHA	Z_{14}	GetIntegerv	ZERO	Blending dest. A function	4.1.8	color-buffer
BLEND.EQUATION.RGB (v1.5: BLEND.EQUATION)	Z_5	GetIntegerv	FUNC.ADD	RGB blending equation	4.1.8	color-buffer
BLEND.EQUATION.ALPHA	Z_5	GetIntegerv	FUNC.ADD	Alpha blending equation	4.1.8	color-buffer
BLEND.COLOR	C	GetFloatv	0,0,0,0	Constant blend color	4.1.8	color-buffer
FRAMEBUFFER.SRGB	B	IsEnabled	FALSE	sRGB update and blending enable	4.1.8	color-buffer/enable
DITHER	B	IsEnabled	TRUE	Dithering enabled	4.1.10	color-buffer/enable
INDEX.LOGIC.OP (v1.0.LOGIC.OP)	B	IsEnabled	FALSE	Index logic op enabled	4.1.11	color-buffer/enable
COLOR.LOGIC.OP	B	IsEnabled	FALSE	Color logic op enabled	4.1.11	color-buffer/enable
LOGIC.OP.MODE	Z_{16}	GetIntegerv	COPY	Logic op function	4.1.11	color-buffer

Table 6.25. Pixel Operations (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
INDEX.WRITEMASK	Z^+	GetIntegerv	1's	Color index writemask	4.2.2	color-buffer
COLOR.WRITEMASK	$1 * 4 \times 4 \times B$	GetBooleanv	(TRUE,TRUE,TRUE,TRUE)	Color write enables (R,G,B,A) for draw buffer i	4.2.2	color-buffer
DEPTH.WRITEMASK	B	GetBooleanv	TRUE	Depth buffer enabled for writing	4.2.2	depth-buffer
STENCIL.WRITEMASK	Z^+	GetIntegerv	1's	Front stencil buffer writemask	4.2.2	stencil-buffer
STENCIL.BACK.WRITEMASK	Z^+	GetIntegerv	1's	Back stencil buffer writemask	4.2.2	stencil-buffer
COLOR.CLEAR.VALUE	C	GetFloatv	0,0,0,0	Color buffer clear value (RGBA mode)	4.2.3	color-buffer
INDEX.CLEAR.VALUE	CI	GetFloatv	0	Color buffer clear value (color index mode)	4.2.3	color-buffer
DEPTH.CLEAR.VALUE	R^+	GetIntegerv	1	Depth buffer clear value	4.2.3	depth-buffer
STENCIL.CLEAR.VALUE	Z^+	GetIntegerv	0	Stencil clear value	4.2.3	stencil-buffer
ACCUM.CLEAR.VALUE	$4 \times R^+$	GetFloatv	0	Accumulation buffer clear value	4.2.3	accum-buffer

Table 6.26. Framebuffer Control

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
DRAW_FRAMEBUFFER_BINDING	Z ⁺	GetIntegerv	0	framebuffer object bound to DRAW_FRAMEBUFFER	4.4.1	–
READ_FRAMEBUFFER_BINDING	Z ⁺	GetIntegerv	0	framebuffer object bound to READ_FRAMEBUFFER	4.4.1	–

Table 6.27. Framebuffer (state per target binding point)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
DRAW_BUFFER _i	$1 * \times Z_{11}^*$	GetIntegerv	see 4.2.1	Draw buffer selected for color output i	4.2.1	color-buffer
READ_BUFFER	Z_{11}^*	GetIntegerv	see 4.3.2	Read source buffer	4.3.2	pixel

Table 6.28. Framebuffer (state per framebuffer object)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
FRAMEBUFFER_ATTACHMENT- _OBJECT_TYPE	Z	GetFramebufferAttachment- Parameteriv	NONE	Type of image attached to framebuffer attachment point	4.4.2	–
FRAMEBUFFER_ATTACHMENT- _OBJECT_NAME	Z	GetFramebufferAttachment- Parameteriv	0	Name of object attached to framebuffer attachment point	4.4.2	–
FRAMEBUFFER_ATTACHMENT- _TEXTURE_LEVEL	Z	GetFramebufferAttachment- Parameteriv	0	Mipmap level of texture image attached, if object attached is texture	4.4.2	–
FRAMEBUFFER_ATTACHMENT- _TEXTURE_CUBE_MAP_FACE	Z^+	GetFramebufferAttachment- Parameteriv	TEXTURE_CUBE- MAP_POSITIVE_X	Cubemap face of texture image attached, if object attached is cubemap texture	4.4.2	–
FRAMEBUFFER_ATTACHMENT- _TEXTURE_LAYER	Z	GetFramebufferAttachment- Parameteriv	0	Layer of texture image attached, if object attached is 3D texture	4.4.2	–
FRAMEBUFFER_ATTACHMENT- _COLOR_ENCODING	Z_2	GetFramebufferAttachment- Parameteriv	-	Encoding of components in the attached image	6.1.3	–
FRAMEBUFFER_ATTACHMENT- _COMPONENT_TYPE	Z_4	GetFramebufferAttachment- Parameteriv	-	Data type of components in the attached image	6.1.3	–
FRAMEBUFFER_ATTACHMENT- _x_SIZE	Z^+	GetFramebufferAttachment- Parameteriv	-	Size in bits of attached image's x component; x is RED, GREEN, BLUE, ALPHA, DEPTH, or STENCIL	6.1.3	–

Table 6.29. Framebuffer (state per attachment point)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
RENDERBUFFER_BINDING	Z	GetInteger	0	renderbuffer object bound to RENDERBUFFER	4.4.2	–

Table 6.30. Renderbuffer (state per target and binding point)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
RENDERBUFFER_WIDTH	Z ⁺	GetRenderbufferParameteriv	0	Width of renderbuffer	4.4.2	–
RENDERBUFFER_HEIGHT	Z ⁺	GetRenderbufferParameteriv	0	Height of renderbuffer	4.4.2	–
RENDERBUFFER_INTERNAL_FORMAT	Z ⁺	GetRenderbufferParameteriv	RGBA	Internal format of renderbuffer	4.4.2	–
RENDERBUFFER_RED_SIZE	Z ⁺	GetRenderbufferParameteriv	0	Size in bits of renderbuffer image's red component	4.4.2	–
RENDERBUFFER_GREEN_SIZE	Z ⁺	GetRenderbufferParameteriv	0	Size in bits of renderbuffer image's green component	4.4.2	–
RENDERBUFFER_BLUE_SIZE	Z ⁺	GetRenderbufferParameteriv	0	Size in bits of renderbuffer image's blue component	4.4.2	–
RENDERBUFFER_ALPHA_SIZE	Z ⁺	GetRenderbufferParameteriv	0	Size in bits of renderbuffer image's alpha component	4.4.2	–
RENDERBUFFER_DEPTH_SIZE	Z ⁺	GetRenderbufferParameteriv	0	Size in bits of renderbuffer image's depth component	4.4.2	–
RENDERBUFFER_STENCIL_SIZE	Z ⁺	GetRenderbufferParameteriv	0	Size in bits of renderbuffer image's stencil component	4.4.2	–
RENDERBUFFER_SAMPLES	Z ⁺	GetRenderbufferParameteriv	0	Number of samples	4.4.2	–

Table 6.31. Renderbuffer (state per renderbuffer object)
Version 3.0 - August 11, 2008

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
UNPACK_SWAP_BYTES	B	GetBooleanv	FALSE	Value of UNPACK_SWAP_BYTES	3.7.1	pixel-store
UNPACK_LSB_FIRST	B	GetBooleanv	FALSE	Value of UNPACK_LSB_FIRST	3.7.1	pixel-store
UNPACK_IMAGE_HEIGHT	Z^+	GetIntegerv	0	Value of UNPACK_IMAGE_HEIGHT	3.7.1	pixel-store
UNPACK_SKIP_IMAGES	Z^+	GetIntegerv	0	Value of UNPACK_SKIP_IMAGES	3.7.1	pixel-store
UNPACK_ROW_LENGTH	Z^+	GetIntegerv	0	Value of UNPACK_ROW_LENGTH	3.7.1	pixel-store
UNPACK_SKIP_ROWS	Z^+	GetIntegerv	0	Value of UNPACK_SKIP_ROWS	3.7.1	pixel-store
UNPACK_SKIP_PIXELS	Z^+	GetIntegerv	0	Value of UNPACK_SKIP_PIXELS	3.7.1	pixel-store
UNPACK_ALIGNMENT	Z^+	GetIntegerv	4	Value of UNPACK_ALIGNMENT	3.7.1	pixel-store
PACK_SWAP_BYTES	B	GetBooleanv	FALSE	Value of PACK_SWAP_BYTES	4.3.2	pixel-store
PACK_LSB_FIRST	B	GetBooleanv	FALSE	Value of PACK_LSB_FIRST	4.3.2	pixel-store
PACK_IMAGE_HEIGHT	Z^+	GetIntegerv	0	Value of PACK_IMAGE_HEIGHT	4.3.2	pixel-store
PACK_SKIP_IMAGES	Z^+	GetIntegerv	0	Value of PACK_SKIP_IMAGES	4.3.2	pixel-store
PACK_ROW_LENGTH	Z^+	GetIntegerv	0	Value of PACK_ROW_LENGTH	4.3.2	pixel-store
PACK_SKIP_ROWS	Z^+	GetIntegerv	0	Value of PACK_SKIP_ROWS	4.3.2	pixel-store
PACK_SKIP_PIXELS	Z^+	GetIntegerv	0	Value of PACK_SKIP_PIXELS	4.3.2	pixel-store
PACK_ALIGNMENT	Z^+	GetIntegerv	4	Value of PACK_ALIGNMENT	4.3.2	pixel-store
PIXEL_PACK_BUFFER_BINDING	Z^+	GetIntegerv	0	Pixel pack buffer binding	4.3.2	pixel-store
PIXEL_UNPACK_BUFFER_BINDING	Z^+	GetIntegerv	0	Pixel unpack buffer binding	6.1.13	pixel-store

Table 6.32. Pixels

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
MAP.COLOR	<i>B</i>	GetBooleanv	FALSE	True if colors are mapped	3.7.3	pixel
MAP.STENCIL	<i>B</i>	GetBooleanv	FALSE	True if stencil values are mapped	3.7.3	pixel
INDEX.SHIFT	<i>Z</i>	GetIntegerv	0	Value of INDEX_SHIFT	3.7.3	pixel
INDEX.OFFSET	<i>Z</i>	GetIntegerv	0	Value of INDEX_OFFSET	3.7.3	pixel
x _SCALE	<i>R</i>	GetFloatv	1	Value of x _SCALE; x is RED, GREEN, BLUE, ALPHA, or DEPTH	3.7.3	pixel
x _BIAS	<i>R</i>	GetFloatv	0	Value of x _BIAS	3.7.3	pixel

Table 6.33. Pixels (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
COLOR.TABLE	B	IsEnabled	FALSE	True if color table lookup is done	3.7.3	pixel/enable
POST_CONVOLUTION.COLOR.TABLE	B	IsEnabled	FALSE	True if post convolution color table lookup is done	3.7.3	pixel/enable
POST_COLOR_MATRIX.COLOR.TABLE	B	IsEnabled	FALSE	True if post color matrix color table lookup is done	3.7.3	pixel/enable
COLOR.TABLE	I	GetColorTable	<i>empty</i>	Color table	3.7.3	–
POST_CONVOLUTION.COLOR.TABLE	I	GetColorTable	<i>empty</i>	Post convolution color table	3.7.3	–
POST_COLOR_MATRIX.COLOR.TABLE	I	GetColorTable	<i>empty</i>	Post color matrix color table	3.7.3	–
COLOR.TABLE.FORMAT	$2 \times 3 \times Z_{42}$	GetColorTable-Parameteriv	RGBA	Color tables' internal image format	3.7.3	–
COLOR.TABLE.WIDTH	$2 \times 3 \times Z^+$	GetColorTable-Parameteriv	0	Color tables' specified width	3.7.3	–
COLOR.TABLE.x.SIZE	$6 \times 2 \times 3 \times Z^+$	GetColorTable-Parameteriv	0	Color table component resolution; x is RED, GREEN, BLUE, ALPHA, LUMINANCE, or INTENSITY	3.7.3	–
COLOR.TABLE.SCALE	$3 \times R^4$	GetColorTable-Parameterfv	1,1,1,1	Scale factors applied to color table entries	3.7.3	pixel
COLOR.TABLE.BIAS	$3 \times R^4$	GetColorTable-Parameterfv	0,0,0,0	Bias factors applied to color table entries	3.7.3	pixel

Table 6.34. Pixels (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CONVOLUTION_1D	B	IsEnabled	FALSE	True if 1D convolution is done	3.7.3	pixel/enable
CONVOLUTION_2D	B	IsEnabled	FALSE	True if 2D convolution is done	3.7.3	pixel/enable
SEPARABLE_2D	B	IsEnabled	FALSE	True if separable 2D convolution is done	3.7.3	pixel/enable
CONVOLUTION_2D	$2 \times I$	GetConvolution-Filter	<i>empty</i>	Convolution filters; x is 1 or 2	3.7.3	–
SEPARABLE_2D	$2 \times I$	GetSeparable-Filter	<i>empty</i>	Separable convolution filter	3.7.3	–
CONVOLUTION_BORDER_COLOR	$3 \times C$	GetConvolution-Parameterfv	0,0,0,0	Convolution border color	3.7.5	pixel
CONVOLUTION_BORDER_MODE	$3 \times Z_4$	GetConvolution-Parameteriv	REDUCE	Convolution border mode	3.7.5	pixel
CONVOLUTION_FILTER_SCALE	$3 \times R^4$	GetConvolution-Parameterfv	1,1,1,1	Scale factors applied to convolution filter entries	3.7.3	pixel
CONVOLUTION_FILTER_BIAS	$3 \times R^4$	GetConvolution-Parameterfv	0,0,0,0	Bias factors applied to convolution filter entries	3.7.3	pixel
CONVOLUTION_FORMAT	$3 \times Z_{42}$	GetConvolution-Parameteriv	RGEA	Convolution filter internal format	3.7.5	–
CONVOLUTION_WIDTH	$3 \times Z^+$	GetConvolution-Parameteriv	0	Convolution filter width	3.7.5	–
CONVOLUTION_HEIGHT	$2 \times Z^+$	GetConvolution-Parameteriv	0	Convolution filter height	3.7.5	–

Table 6.35. Pixels (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
POST_CONVOLUTION. x .SCALE	R	GetFloatv	1	Component scale factors after convolution; x is RED, GREEN, BLUE, or ALPHA	3.7.3	pixel
POST_CONVOLUTION. x .BIAS	R	GetFloatv	0	Component bias factors after convolution	3.7.3	pixel
POST_COLOR_MATRIX. x .SCALE	R	GetFloatv	1	Component scale factors after color matrix	3.7.3	pixel
POST_COLOR_MATRIX. x .BIAS	R	GetFloatv	0	Component bias factors after color matrix	3.7.3	pixel
HISTOGRAM	B	IsEnabled	FALSE	True if histogramming is enabled	3.7.3	pixel/enable
HISTOGRAM	I	GetHistogram	<i>empty</i>	Histogram table	3.7.3	–
HISTOGRAM.WIDTH	$2 \times Z^+$	GetHistogram-Parameteriv	0	Histogram table width	3.7.3	–
HISTOGRAM.FORMAT	$2 \times Z_{42}$	GetHistogram-Parameteriv	RGBA	Histogram table internal format	3.7.3	–
HISTOGRAM. x .SIZE	$5 \times 2 \times Z^+$	GetHistogram-Parameteriv	0	Histogram table component resolution; x is RED, GREEN, BLUE, ALPHA, or LUMINANCE	3.7.3	–
HISTOGRAM.SINK	B	GetHistogram-Parameteriv	FALSE	True if histogramming consumes pixel groups	3.7.3	–

Table 6.36. Pixels (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
MINMAX	B	IsEnabled	FALSE	True if minmax is enabled	3.7.3	pixel/enable
MINMAX	R^n	GetMinmax	(M,M,M,M),(m,m,m,m)	Minmax table	3.7.3	–
MINMAX_FORMAT	Z_{42}	GetMinmax-Parameteriv	RGBA	Minmax table internal format	3.7.3	–
MINMAX_SINK	B	GetMinmax-Parameteriv	FALSE	True if minmax consumes pixel groups	3.7.3	–
ZOOM.X	R	GetFloatv	1.0	x zoom factor	3.7.4	pixel
ZOOM.Y	R	GetFloatv	1.0	y zoom factor	3.7.4	pixel
x	$8 \times 32 * \times R$	GetPixelFormat	0's	RGBA PixelFormat translation tables; x is a map name from table 3.3	3.7.3	–
x	$2 \times 32 * \times Z$	GetPixelFormat	0's	Index PixelFormat translation tables; x is a map name from table 3.3	3.7.3	–
x .SIZE	Z^+	GetIntegerv	1	Size of table x	3.7.3	–

Table 6.37. Pixels (cont.)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
ORDER	$9 \times Z_{8*}$	GetMapiv	1	1d map order	5.1	–
ORDER	$9 \times 2 \times Z_{8*}$	GetMapiv	1,1	2d map orders	5.1	–
COEFF	$9 \times 8 * \times R^n$	GetMapfv	see 5.1	1d control points	5.1	–
COEFF	$9 \times 8 * \times 8 * \times R^n$	GetMapfv	see 5.1	2d control points	5.1	–
DOMAIN	$9 \times 2 \times R$	GetMapfv	see 5.1	1d domain endpoints	5.1	–
DOMAIN	$9 \times 4 \times R$	GetMapfv	see 5.1	2d domain endpoints	5.1	–
MAP1. <i>x</i>	$9 \times B$	IsEnabled	FALSE	1d map enables: <i>x</i> is map type	5.1	eval/enable
MAP2. <i>x</i>	$9 \times B$	IsEnabled	FALSE	2d map enables: <i>x</i> is map type	5.1	eval/enable
MAP1.GRID.DOMAIN	$2 \times R$	GetFloatv	0,1	1d grid endpoints	5.1	eval
MAP2.GRID.DOMAIN	$4 \times R$	GetFloatv	0,1;0,1	2d grid endpoints	5.1	eval
MAP1.GRID.SEGMENTS	Z^+	GetFloatv	1	1d grid divisions	5.1	eval
MAP2.GRID.SEGMENTS	$2 \times Z^+$	GetFloatv	1,1	2d grid divisions	5.1	eval
AUTO.NORMAL	B	IsEnabled	FALSE	True if automatic normal generation enabled	5.1	eval/enable

Table 6.38. Evaluators (**GetMap** takes a map name)

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
SHADER.TYPE	Z_2	GetShaderiv	-	Type of shader (vertex or fragment)	2.20.1	-
DELETE.STATUS	B	GetShaderiv	FALSE	Shader flagged for deletion	2.20.1	-
COMPILE.STATUS	B	GetShaderiv	FALSE	Last compile succeeded	2.20.1	-
-	S	GetShaderInfoLog	empty string	Info log for shader objects	6.1.15	-
INFO.LOG.LENGTH	Z^+	GetShaderiv	0	Length of info log	6.1.15	-
-	S	GetShaderSource	empty string	Source code for a shader	2.20.1	-
SHADER.SOURCE.LENGTH	Z^+	GetShaderiv	0	Length of source code	6.1.15	-

Table 6.39. Shader Object State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
CURRENT_PROGRAM	Z^+	GetIntegerv	0	Name of current program object	2.20.2	–
DELETE_STATUS	B	GetProgramiv	FALSE	Program object deleted	2.20.2	–
LINK_STATUS	B	GetProgramiv	FALSE	Last link attempt succeeded	2.20.2	–
VALIDATE_STATUS	B	GetProgramiv	FALSE	Last validate attempt succeeded	2.20.2	–
ATTACHED_SHADERS	Z^+	GetProgramiv	0	Number of attached shader objects	6.1.15	–
-	$0 * \times Z^+$	GetAttachedShaders	empty	Shader objects attached	6.1.15	–
-	S	GetProgramInfoLog	empty	Info log for program object	6.1.15	–
INFO_LOG_LENGTH	Z^+	GetProgramiv	0	Length of info log	2.20.3	–
ACTIVE_UNIFORMS	Z^+	GetProgramiv	0	Number of active uniforms	2.20.3	–
-	$0 * \times Z$	GetUniformLocation	–	Location of active uniforms	6.1.15	–
-	$0 * \times Z^+$	GetActiveUniform	–	Size of active uniform	2.20.3	–
-	$0 * \times Z^+$	GetActiveUniform	–	Type of active uniform	2.20.3	–
-	$0 * \times \text{char}$	GetActiveUniform	empty	Name of active uniform	2.20.3	–
ACTIVE_UNIFORM_MAX_LENGTH	Z^+	GetProgramiv	0	Maximum active uniform name length	6.1.15	–
-	$512 * \times R$	GetUniform	0	Uniform value	2.20.3	–
ACTIVE_ATTRIBUTES	Z^+	GetProgramiv	0	Number of active attributes	2.20.3	–

Table 6.40. Program Object State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
-	$0 * \times Z$	GetAttribLocation	-	Location of active generic attribute	2.20.3	-
-	$0 * \times Z^+$	GetActiveAttrib	-	Size of active attribute	2.20.3	-
-	$0 * \times Z^+$	GetActiveAttrib	-	Type of active attribute	2.20.3	-
-	$0 * \times \text{char}$	GetActiveAttrib	empty	Name of active attribute	2.20.3	-
ACTIVE_ATTRIBUTE_MAX_LENGTH	Z^+	GetProgramiv	0	Maximum active attribute name length	6.1.15	-
TRANSFORM_FEEDBACK_BUFFER_MODE	Z_2	GetProgramiv	INTERLEAVED-ATTRIBS	Transform feedback mode for the program	6.1.15	-
TRANSFORM_FEEDBACK_VARYINGS	Z^+	GetProgramiv	0	Number of varyings to stream to buffer object(s)	6.1.15	-
TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH	Z^+	GetProgramiv	0	Maximum transform feedback varying name length	6.1.15	-
-	Z^+	GetTransform-FeedbackVarying	-	Size of each transform feedback varying variable	2.20.3	-
-	Z^+	GetTransform-FeedbackVarying	-	Type of each transform feedback varying variable	2.20.3	-
-	$0^+ \times \text{char}$	GetTransform-FeedbackVarying	-	Name of each transform feedback varying variable	2.20.3	-

Table 6.41. Program Object State (cont.)
Version 3.0 - August 11, 2008

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
VERTEX_PROGRAM_TWO_SIDE	B	IsEnabled	FALSE	Two-sided color mode	2.19.1	enable
CURRENT_VERTEX_ATTRIB	$16 * R4$	GetVertexAttrib	0,0,0,1	Generic vertex attribute	2.7	current
VERTEX_PROGRAM_POINT_SIZE	B	IsEnabled	FALSE	Point size mode	3.4	enable

Table 6.42. Vertex Shader State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
QUERY_RESULT	Z^+	GetQueryObjectiv	0	Query object result	6.1.12	–
QUERY_RESULT_AVAILABLE	B	GetQueryObjectiv	FALSE	Is the query object result available?	6.1.12	–

Table 6.43. Query Object State

Get value	Get Command		Initial Value	Description	Sec.	Attribute
	Type	Get Command				
TRANSFORM_FEEDBACK.BUFFER.BINDING	Z^+	GetInteger_v	0	Buffer object bound to generic bind point for transform feedback	6.1.13	–
TRANSFORM_FEEDBACK.BUFFER.BINDING	nxZ^+	GetIntegeri_v	0	Buffer object bound to each transform feedback attribute stream	6.1.13	–
TRANSFORM_FEEDBACK.BUFFER.START	nxZ^+	GetIntegeri_v	0	Start offset of binding range for each transform feedback attrib. stream	6.1.13	–
TRANSFORM_FEEDBACK.BUFFER.SIZE	$n \times Z^+$	GetIntegeri_v	0	Size of binding range for each transform feedback attrib. stream	6.1.13	–
MAX.TRANSFORM_FEEDBACK.INTERLEAVED.COMPONENTS	Z^+	GetInteger_v	64	Max number of components to write to a single buffer in interleaved mode	2.15	–
MAX.TRANSFORM_FEEDBACK.SEPARATE.ATTRIBS	Z^+	GetInteger_v	4	Max number of separate attributes or vayings that can be captured in transform feedback	2.15	–
MAX.TRANSFORM_FEEDBACK.SEPARATE.COMPONENTS	Z^+	GetInteger_v	4	Max number of components per attribute or varying in separate mode	2.15	–

Table 6.44. Transform Feedback State

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
PERSPECTIVE_CORRECTION_HINT	Z_3	GetInteger	DONT_CARE	Perspective correction hint	5.6	hint
POINT_SMOOTH_HINT	Z_3	GetInteger	DONT_CARE	Point smooth hint	5.6	hint
LINE_SMOOTH_HINT	Z_3	GetInteger	DONT_CARE	Line smooth hint	5.6	hint
POLYGON_SMOOTH_HINT	Z_3	GetInteger	DONT_CARE	Polygon smooth hint	5.6	hint
FOG_HINT	Z_3	GetInteger	DONT_CARE	Fog hint	5.6	hint
GENERATE_MIPMAP_HINT	Z_3	GetInteger	DONT_CARE	Mipmap generation hint	5.6	hint
TEXTURE_COMPRESSION_HINT	Z_3	GetInteger	DONT_CARE	Texture compression quality hint	5.6	hint
FRAGMENT_SHADER_DERIVATIVE_HINT	Z_3	GetInteger	DONT_CARE	Fragment shader derivative accuracy hint	5.6	hint

Table 6.45. Hints

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_LIGHTS	Z^+	GetIntegerv	8	Maximum number of lights	2.19.1	—
MAX_CLIP_PLANES	Z^+	GetIntegerv	6	Maximum number of user clipping planes	2.17	—
MAX_COLOR_MATRIX_STACK_DEPTH	Z^+	GetIntegerv	2	Maximum color matrix stack depth	3.7.3	—
MAX_MODELVIEW_STACK_DEPTH	Z^+	GetIntegerv	32	Maximum model-view stack depth	2.12.2	—
MAX_PROJECTION_STACK_DEPTH	Z^+	GetIntegerv	2	Maximum projection matrix stack depth	2.12.2	—
MAX_TEXTURE_STACK_DEPTH	Z^+	GetIntegerv	2	Maximum number depth of texture matrix stack	2.12.2	—
SUBPIXEL_BITS	Z^+	GetIntegerv	4	Number of bits of subpixel precision in screen x_w and y_w	3	—
MAX_3D_TEXTURE_SIZE	Z^+	GetIntegerv	256	Maximum 3D texture image dimension	3.9.1	—
MAX_TEXTURE_SIZE	Z^+	GetIntegerv	1024	Maximum 2D/1D texture image dimension	3.9.1	—
MAX_ARRAY_TEXTURE_LAYERS	Z^+	GetIntegerv	256	Maximum number of layers for texture arrays	3.9.1	—
MAX_TEXTURE_LOD_BIAS	R^+	GetFloatv	2.0	Maximum absolute texture level of detail bias	3.9.7	—
MAX_CUBE_MAP_TEXTURE_SIZE	Z^+	GetIntegerv	1024	Maximum cube map texture image dimension	3.9.1	—
MAX_RENDERBUFFER_SIZE	Z^+	GetIntegerv	1024	Maximum width and height of renderbuffers	4.4.2	—

Table 6.46. Implementation Dependent Values

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_PIXEL_MAP_TABLE	Z^+	GetIntegerv	32	Maximum size of a Pixmap translation table	3.7.3	–
MAX_NAME_STACK_DEPTH	Z^+	GetIntegerv	64	Maximum selection name stack depth	5.2	–
MAX_LIST_NESTING	Z^+	GetIntegerv	64	Maximum display list call nesting	5.4	–
MAX_EVAL_ORDER	Z^+	GetIntegerv	8	Maximum evaluator polynomial order	5.1	–
MAX_VIEWPORT_DIMS	$2 \times Z^+$	GetIntegerv	see 2.12.1	Maximum viewport dimensions	2.12.1	–
MAX_ATTRIB_STACK_DEPTH	Z^+	GetIntegerv	16	Maximum depth of the server attribute stack	6	–
MAX_CLIENT_ATTRIB_STACK_DEPTH	Z^+	GetIntegerv	16	Maximum depth of the client attribute stack	6	–
–	$3 \times Z^+$	–	32	Max. size of a color table	3.7.3	–
–	Z^+	–	32	Max. size of the histogram table	3.7.3	–
ALIASED_POINT_SIZE_RANGE	$2 \times R^+$	GetFloatv	1,1	Range (lo to hi) of aliased point sizes	3.4	–
SMOOTH_POINT_SIZE_RANGE (v1.1: POINT_SIZE_RANGE)	$2 \times R^+$	GetFloatv	1,1	Range (lo to hi) of antialiased point sizes	3.4	–
SMOOTH_POINT_SIZE_GRANULARITY (v1.1: POINT_SIZE_GRANULARITY)	R^+	GetFloatv	–	Antialiased point size granularity	3.4	–

Table 6.47. Implementation Dependent Values (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
ALIASED.LINE.WIDTH.RANGE	$2 \times R^+$	GetFloatv	1,1	Range (lo to hi) of aliased line widths	3.5	–
SMOOTH.LINE.WIDTH.RANGE (v1.1: LINE.WIDTH.RANGE)	$2 \times R^+$	GetFloatv	1,1	Range (lo to hi) of antialiased line widths	3.5	–
SMOOTH.LINE.WIDTH.GRANULARITY (v1.1: LINE.WIDTH.GRANULARITY)	R^+	GetFloatv	–	Antialiased line width granularity	3.5	–
MAX.CONVOLUTION.WIDTH	$3 \times Z^+$	GetConvolution-Parameteriv	3	Maximum width of convolution filter	4.3	–
MAX.CONVOLUTION.HEIGHT	$2 \times Z^+$	GetConvolution-Parameteriv	3	Maximum height of convolution filter	4.3	–
MAX.ELEMENTS.INDICES	Z^+	GetIntegerv	–	Recommended max. number of DrawRangeElements indices	2.8	–
MAX.ELEMENTS.VERTICES	Z^+	GetIntegerv	–	Recommended max. number of DrawRangeElements vertices	2.8	–
COMPRESSED.TEXTURE.FORMATS	$0 \times Z$	GetIntegerv	–	Enumerated compressed texture formats	3.9.3	–
NUM.COMPRESSED.TEXTURE.FORMATS	Z	GetIntegerv	0	Number of compressed texture formats	3.9.3	–

Table 6.48. Implementation Dependent Values (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
QUERY_COUNTER_BITS	$2 \times Z^+$	GetQueryiv	see 6.1.12	Asynchronous query counter bits	6.1.12	–
EXTENSIONS	$0 * \times S$	GetStringi	–	Supported individual extension names	6.1.11	–
NUM_EXTENSIONS	Z^+	GetIntegerv	–	Number of individual extension names	6.1.11	–
MAJOR_VERSION	Z^+	GetIntegerv	–	Major version number supported	6.1.11	–
MINOR_VERSION	Z^+	GetIntegerv	–	Minor version number supported	6.1.11	–
CONTEXT_FLAGS	Z^+	GetIntegerv	–	Context full/forward-compatible flag	6.1.11	–
EXTENSIONS	S	GetString	–	Supported extension names	6.1.11	–
RENDERER	S	GetString	–	Renderer string	6.1.11	–
SHADING_LANGUAGE_VERSION	S	GetString	–	Shading Language version supported	6.1.11	–
VENDOR	S	GetString	–	Vendor string	6.1.11	–
VERSION	S	GetString	–	OpenGL version supported	6.1.11	–

Table 6.49. Implementation Dependent Values (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
MAX_TEXTURE_UNITS	Z ⁺	GetIntegerv	2	Number of fixed-function texture units	2.6	–
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	2.20.3	–
MAX_VARYING_COMPONENTS	Z ⁺	GetIntegerv	64	Number of components for varying variables	2.20.3	–
MAX_COMBINED_TEXTURE_IMAGE_UNITS	Z ⁺	GetIntegerv	16	Total number of texture units accessible by the GL	2.20.4	–
MAX_VERTEX_TEXTURE_IMAGE_UNITS	Z ⁺	GetIntegerv	16	Number of texture image units accessible by a vertex shader	2.20.4	–
MAX_TEXTURE_IMAGE_UNITS	Z ⁺	GetIntegerv	16	Number of texture image units accessible by fragment processing	2.20.4	–
MAX_TEXTURE_COORDS	Z ⁺	GetIntegerv	8	Number of texture coordinate sets	2.7	–
MAX_FRAGMENT_UNIFORM_COMPONENTS	Z ⁺	GetIntegerv	1024	Number of components for frag. shader uniform variables	3.12.1	–
MIN_PROGRAM_TEXEL_OFFSET	Z	GetIntegerv	-8	Minimum texel offset allowed in lookup	2.20.4	–
MAX_PROGRAM_TEXEL_OFFSET	Z	GetIntegerv	7	Maximum texel offset allowed in lookup	2.20.4	–

Table 6.50. Implementation Dependent Values (cont.)

Get value	Type	Get Command	Minimum Value	Description	Sec.	Attribute
AUX_BUFFERS	Z^+	GetIntegerv	0	Number of auxiliary buffers	4.2.1	—
MAX_DRAW_BUFFERS	Z^+	GetIntegerv	8*	Maximum number of active draw buffers	4.2.1	—
RGBA_MODE	B	GetBooleany	—	True if color buffers store RGBA	2.7	—
INDEX_MODE	B	GetBooleany	—	True if color buffers store indexes	2.7	—
DOUBLEBUFFER	B	GetBooleany	—	True if front & back buffers exist	4.2.1	—
STEREO	B	GetBooleany	—	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	—
x_BITS	Z^+	GetIntegerv	—	Number of bits in x color buffer component. x is one of RED, GREEN, BLUE, ALPHA, or INDEX	4	—
DEPTH_BITS	Z^+	GetIntegerv	—	Number of depth buffer planes	4	—
STENCIL_BITS	Z^+	GetIntegerv	—	Number of stencil planes	4	—
ACCUM_ x_BITS	Z^+	GetIntegerv	—	Number of bits in x accumulation buffer component (x is RED, GREEN, BLUE, or ALPHA)	4	—

Table 6.51. Framebuffer Dependent Values

Get value	Type	Get Command	Initial Value	Description	Sec.	Attribute
LIST_BASE	Z^+	GetIntegerv	0	Setting of ListBase	5.4	list
LIST_INDEX	Z^+	GetIntegerv	0	Number of display list under construction; 0 if none	5.4	–
LIST_MODE	Z^+	GetIntegerv	0	Mode of display list under construction; undefined if none	5.4	–
–	$16 * \times A$	–	empty	Server attribute stack	6	–
ATTRIB_STACK_DEPTH	Z^+	GetIntegerv	0	Server attribute stack pointer	6	–
–	$16 * \times A$	–	empty	Client attribute stack	6	–
CLIENT_ATTRIB_STACK_DEPTH	Z^+	GetIntegerv	0	Client attribute stack pointer	6	–
NAME_STACK_DEPTH	Z^+	GetIntegerv	0	Name stack depth	5.2	–
RENDER_MODE	Z_3	GetIntegerv	RENDER	RenderMode setting	5.2	–
SELECTION_BUFFER_POINTER	Y	GetPointerv	0	Selection buffer pointer	5.2	select
SELECTION_BUFFER_SIZE	Z^+	GetIntegerv	0	Selection buffer size	5.2	select
FEEDBACK_BUFFER_POINTER	Y	GetPointerv	0	Feedback buffer pointer	5.3	feedback
FEEDBACK_BUFFER_SIZE	Z^+	GetIntegerv	0	Feedback buffer size	5.3	feedback
FEEDBACK_BUFFER_TYPE	Z_5	GetIntegerv	2D	Feedback type	5.3	feedback
–	$n \times Z_8$	GetError	0	Current error code(s)	2.5	–
–	$n \times B$	–	FALSE	True if there is a corresponding error	2.5	–
–	B	–	FALSE	Occlusion query active	4.1.7	–
CURRENT_QUERY	$3 \times Z^+$	GetQueryiv	0	Active query object names	6.1.12	–

Table 6.52. Miscellaneous

Appendix A

Invariance

The OpenGL specification is not pixel exact. It therefore does not guarantee an exact match between images produced by different GL implementations. However, the specification does specify exact matches, in some cases, for images produced by the same implementation. The purpose of this appendix is to identify and provide justification for those cases that require exact matches.

A.1 Repeatability

The obvious and most fundamental case is repeated issuance of a series of GL commands. For any given GL and framebuffer state *vector*, and for any GL command, the resulting GL and framebuffer state must be identical whenever the command is executed on that initial GL and framebuffer state.

One purpose of repeatability is avoidance of visual artifacts when a double-buffered scene is redrawn. If rendering is not repeatable, swapping between two buffers rendered with the same command sequence may result in visible changes in the image. Such false motion is distracting to the viewer. Another reason for repeatability is testability.

Repeatability, while important, is a weak requirement. Given only repeatability as a requirement, two scenes rendered with one (small) polygon changed in position might differ at every pixel. Such a difference, while within the law of repeatability, is certainly not within its spirit. Additional invariance rules are desirable to ensure useful operation.

A.2 Multi-pass Algorithms

Invariance is necessary for a whole set of useful multi-pass algorithms. Such algorithms render multiple times, each time with a different GL mode vector, to eventually produce a result in the framebuffer. Examples of these algorithms include:

- “Erasing” a primitive from the framebuffer by redrawing it, either in a different color or using the XOR logical operation.
- Using stencil operations to compute capping planes.

On the other hand, invariance rules can greatly increase the complexity of high-performance implementations of the GL. Even the weak repeatability requirement significantly constrains a parallel implementation of the GL. Because GL implementations are required to implement ALL GL capabilities, not just a convenient subset, those that utilize hardware acceleration are expected to alternate between hardware and software modules based on the current GL mode vector. A strong invariance requirement forces the behavior of the hardware and software modules to be identical, something that may be very difficult to achieve (for example, if the hardware does floating-point operations with different precision than the software).

What is desired is a compromise that results in many compliant, high-performance implementations, and in many software vendors choosing to port to OpenGL.

A.3 Invariance Rules

For a given instantiation of an OpenGL rendering context:

Rule 1 *For any given GL and framebuffer state vector, and for any given GL command, the resulting GL and framebuffer state must be identical each time the command is executed on that initial GL and framebuffer state.*

Rule 2 *Changes to the following state values have no side effects (the use of any other state value is not affected by the change):*

Required:

- *Framebuffer contents (all bitplanes)*
- *The color buffers enabled for writing*
- *The values of matrices other than the top-of-stack matrices*

- *Scissor parameters (other than enable)*
- *Writemasks (color, index, depth, stencil)*
- *Clear values (color, index, depth, stencil, accumulation)*
 - *Current values (color, index, normal, texture coords, edgeflag)*
 - *Current raster color, index and texture coordinates.*
 - *Material properties (ambient, diffuse, specular, emission, shininess)*

Strongly suggested:

- *Matrix mode*
- *Matrix stack depths*
- *Alpha test parameters (other than enable)*
- *Stencil parameters (other than enable)*
- *Depth test parameters (other than enable)*
- *Blend parameters (other than enable)*
- *Logical operation parameters (other than enable)*
- *Pixel storage and transfer state*
- *Evaluator state (except as it affects the vertex data generated by the evaluators)*
- *Polygon offset parameters (other than enables, and except as they affect the depth values of fragments)*

Corollary 1 *Fragment generation is invariant with respect to the state values marked with • in Rule 2.*

Corollary 2 *The window coordinates (x, y, and z) of generated fragments are also invariant with respect to*

Required:

- *Current values (color, color index, normal, texture coords, edgeflag)*
- *Current raster color, color index, and texture coordinates*
- *Material properties (ambient, diffuse, specular, emission, shininess)*

Rule 3 *The arithmetic of each per-fragment operation is invariant except with respect to parameters that directly control it (the parameters that control the alpha test, for instance, are the alpha test enable, the alpha test function, and the alpha test reference value).*

Corollary 3 *Images rendered into different color buffers sharing the same frame-buffer, either simultaneously or separately using the same command sequence, are pixel identical.*

Rule 4 *The same vertex or fragment shader will produce the same result when run multiple times with the same input. The wording 'the same shader' means a program object that is populated with the same source strings, which are compiled and then linked, possibly multiple times, and which program object is then executed using the same GL state vector.*

Rule 5 *All fragment shaders that either conditionally or unconditionally assign `gl_FragCoord.z` to `gl_FragDepth` are depth-invariant with respect to each other, for those fragments where the assignment to `gl_FragDepth` actually is done.*

A.4 What All This Means

Hardware accelerated GL implementations are expected to default to software operation when some GL state vectors are encountered. Even the weak repeatability requirement means, for example, that OpenGL implementations cannot apply hysteresis to this swap, but must instead guarantee that a given mode vector implies that a subsequent command *always* is executed in either the hardware or the software machine.

The stronger invariance rules constrain when the switch from hardware to software rendering can occur, given that the software and hardware renderers are not pixel identical. For example, the switch can be made when blending is enabled or disabled, but it should not be made when a change is made to the blending parameters.

Because floating point values may be represented using different formats in different renderers (hardware and software), many OpenGL state values may change subtly when renderers are swapped. This is the type of state value change that Rule 1 seeks to avoid.

Appendix B

Corollaries

The following observations are derived from the body and the other appendixes of the specification. Absence of an observation from this list in no way impugns its veracity.

1. The `CURRENT_RASTER_TEXTURE_COORDS` must be maintained correctly at all times, including periods while texture mapping is not enabled, and when the GL is in color index mode.
2. When requested, texture coordinates returned in feedback mode are always valid, including periods while texture mapping is not enabled, and when the GL is in color index mode.
3. The error semantics of upward compatible OpenGL revisions may change. Otherwise, only additions can be made to upward compatible revisions.
4. GL query commands are not required to satisfy the semantics of the **Flush** or the **Finish** commands. All that is required is that the queried state be consistent with complete execution of all previously executed GL commands.
5. Application specified point size and line width must be returned as specified when queried. Implementation dependent clamping affects the values only while they are in use.
6. Bitmaps and pixel transfers do not cause selection hits.
7. The mask specified as the third argument to **StencilFunc** affects the operands of the stencil comparison function, but has no direct effect on the update of the stencil buffer. The mask specified by **StencilMask** has no effect on the stencil comparison function; it limits the effect of the update of the stencil buffer.

8. Polygon shading is completed before the polygon mode is interpreted. If the shade model is `FLAT`, all of the points or lines generated by a single polygon will have the same color.
9. A display list is just a group of commands and arguments, so errors generated by commands in a display list must be generated when the list is executed. If the list is created in `COMPILE` mode, errors should not be generated while the list is being created.
10. **RasterPos** does not change the current raster index from its default value in an RGBA mode GL context. Likewise, **RasterPos** does not change the current raster color from its default value in a color index GL context. Both the current raster index and the current raster color can be queried, however, regardless of the color mode of the GL context.
11. A material property that is attached to the current color via **ColorMaterial** always takes the value of the current color. Attempts to change that material property via **Material** calls have no effect.
12. **Material** and **ColorMaterial** can be used to modify the RGBA material properties, even in a color index context. Likewise, **Material** can be used to modify the color index material properties, even in an RGBA context.
13. There is no atomicity requirement for OpenGL rendering commands, even at the fragment level.
14. Because rasterization of non-antialiased polygons is point sampled, polygons that have no area generate no fragments when they are rasterized in `FILL` mode, and the fragments generated by the rasterization of “narrow” polygons may not form a continuous array.
15. OpenGL does not force left- or right-handedness on any of its coordinates systems. Consider, however, the following conditions: (1) the object coordinate system is right-handed; (2) the only commands used to manipulate the model-view matrix are **Scale** (with positive scaling values only), **Rotate**, and **Translate**; (3) exactly one of either **Frustum** or **Ortho** is used to set the projection matrix; (4) the near value is less than the far value for **DepthRange**. If these conditions are all satisfied, then the eye coordinate system is right-handed and the clip, normalized device, and window coordinate systems are left-handed.
16. **ColorMaterial** has no effect on color index lighting.

17. (No pixel dropouts or duplicates.) Let two polygons share an identical edge (that is, there exist vertices A and B of an edge of one polygon, and vertices C and D of an edge of the other polygon, and the coordinates of vertex A (resp. B) are identical to those of vertex C (resp. D), and the state of the coordinate transformations is identical when A, B, C, and D are specified). Then, when the fragments produced by rasterization of both polygons are taken together, each fragment intersecting the interior of the shared edge is produced exactly once.
18. OpenGL state continues to be modified in `FEEDBACK` mode and in `SELECT` mode. The contents of the framebuffer are not modified.
19. The current raster position, the user defined clip planes, the spot directions and the light positions for `LIGHTi`, and the eye planes for texgen are transformed when they are specified. They are not transformed during a **PopAttrib**, or when copying a context.
20. Dithering algorithms may be different for different components. In particular, alpha may be dithered differently from red, green, or blue, and an implementation may choose to not dither alpha at all.
21. For any GL and framebuffer state, and for any group of GL commands and arguments, the resulting GL and framebuffer state is identical whether the GL commands and arguments are executed normally or from a display list.

Appendix C

Compressed Texture Image Formats

C.1 RGTC Compressed Texture Image Formats

Compressed texture images stored using the RGTC compressed image encodings are represented as a collection of 4×4 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 less than four, the data corresponding to texels outside the image are irrelevant and undefined.

When an RGTC image with a width of w , height of h , and block size of *blocksize* (8 or 16 bytes) is decoded, the corresponding image size (in bytes) is:

$$\lceil \frac{w}{4} \rceil \times \lceil \frac{h}{4} \rceil \times \text{blocksize}.$$

When decoding an RGTC image, the block containing the texel at offset (x, y) begins at an offset (in bytes) relative to the base of the image of:

$$\text{blocksize} \times \left(\lceil \frac{w}{4} \rceil \times \lfloor \frac{y}{4} \rfloor + \lfloor \frac{x}{4} \rfloor \right).$$

The data corresponding to a specific texel (x, y) are extracted from a 4×4 texel block using a relative (x, y) value of

$$(x \bmod 4, y \bmod 4).$$

There are four distinct RGTC image formats:

C.1.1 Format COMPRESSED_RED_RGTC1

Each 4×4 block of texels consists of 64 bits of unsigned red image data.

Each red image data block is encoded as a sequence of 8 bytes, called (in order of increasing address):

$$red_0, red_1, bits_0, bits_1, bits_2, bits_3, bits_4, bits_5$$

The 6 $bits_*$ bytes of the block are decoded into a 48-bit bit vector:

$$bits = bits_0 + 256 \times (bits_1 + 256 \times (bits_2 + 256 \times (bits_3 + 256 \times (bits_4 + 256 \times bits_5))))$$

red_0 and red_1 are 8-bit unsigned integers that are unpacked to red values RED_0 and RED_1 as though they were pixels with a *format* of LUMINANCE and a *type* of UNSIGNED_BYTE.

$bits$ is a 48-bit unsigned integer, from which a three-bit control code is extracted for a texel at location (x, y) in the block using:

$$code(x, y) = bits[3 \times (4 \times y + x) + 2 \dots 3 \times (4 \times y + x) + 0]$$

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 display lists, pixel and vertex buffer objects, program and shader objects, and texture objects (except for the texture objects named zero).

Framebuffer and vertex array objects are not shared.

D.1 Object Deletion Behavior

After an object is deleted, its name is immediately marked unused. Caution should be taken when deleting a shared object bound in multiple contexts. Following its deletion, the object's name can be used by any context to create a new object, or be returned by **Gen*** commands, even though the underlying object state and data may still be in use by contexts other than the one in which the object was deleted. Such another context may continue using the object, and may still contain state identifying its name as being currently bound, until such time as another attempt to bind the name is made in that context. Since the name is marked unused, rather than rebinding the underlying object, a new object with the same name will be created.

Appendix E

The Deprecation Model

OpenGL 3.0 introduces a deprecation model in which certain features may be marked as *deprecated*. Deprecated features are expected to be completely removed from a future version of OpenGL. Deprecated features are summarized in section [E.1](#).

To aid developers in writing applications which will run on such future versions, it is possible to create an OpenGL 3.0 context which does not support deprecated features. Such a context is called a *forward compatible* context, while a context supporting all OpenGL 3.0 features is called a *full* context. Forward compatible contexts cannot restore deprecated functionality through extensions, but they may support additional, non-deprecated functionality through extensions.

Profiles allow defining subsets of OpenGL functionality targeted to specific application domains. While OpenGL 3.0 only defines a single profile, future versions may introduce profiles addressing domains such as workstation, gaming, and embedded. Implementations are not required to support all defined profiles, but must support at least one profile.

To enable application control of deprecation and profiles, new *context creation APIs* have been defined as extensions to GLX and WGL. These APIs allow specifying a particular version, profile, and full or forward compatible status, and will either create a context compatible with the request, or fail (if, for example, requesting an OpenGL version or profile not supported by the implementation),

Only the ARB may define OpenGL profiles and deprecated features.

E.1 Profiles and Deprecated Features of OpenGL 3.0

OpenGL 3.0 defines a single profile, and all OpenGL 3.0 implementations must support that profile.

The features deprecated in OpenGL 3.0 are summarized below, together with the sections of the specification in which they are defined. Functions which are completely deprecated will generate an `INVALID_OPERATION` error if called in a forward-compatible context. Functions which are partially deprecated (e.g. no longer accept some parameter values) will generate the errors appropriate for any other unrecognized value of that parameter when a deprecated value is passed in a forward-compatible context.

- Application-generated object names - the names of all object types, such as buffer, query, and texture objects, must be generated using the corresponding **Gen*** commands. Trying to bind an object name not returned by a **Gen*** command will result in an `INVALID_OPERATION` error. This behavior is already the case for framebuffer, renderbuffer, and vertex array objects.
- Color index mode - No color index visuals are supplied by the window system-binding APIs such as GLX and WGL, so the default framebuffer is always in RGBA mode. All language and state related to color index mode vertex, rasterization, and fragment processing behavior is removed. `COLOR_INDEX` formats are also deprecated.
- OpenGL Shading Language versions 1.10 and 1.20. These versions of the shading language depend on many API features that have also been deprecated.
- **Begin / End** primitive specification - **Begin**, **End**, and **EdgeFlag*** (section 2.6.1); **Color***, **FogCoord***, **Index***, **Normal3***, **SecondaryColor3***, **TexCoord***, **Vertex*** (section 2.7); and all associated state in tables 6.4 and 6.5. Vertex arrays and array drawing commands must be used to draw primitives. However, **VertexAttrib*** and the current vertex attribute state are retained in order to provide default attribute values for disabled attribute arrays.
- Edge flags and fixed-function vertex processing - **ColorPointer**, **EdgeFlagPointer**, **FogCoordPointer**, **IndexPointer**, **NormalPointer**, **SecondaryColorPointer**, **TexCoordPointer**, **VertexPointer**, **EnableClientState**, **DisableClientState**, and **ClientActiveTexture** (section 2.8); **Frustum**, **LoadIdentity**, **LoadMatrix**, **LoadTransposeMatrix**, **MatrixMode**, **MultMatrix**, **MultTransposeMatrix**, **Ortho**, **PopMatrix**, **PushMatrix**, **Rotate**, **Scale**, and **Translate** (section 2.12.2; **Enable/Disable** targets `RESCALE_NORMAL` and `NORMALIZE` (section 2.12.3); **TexGen*** and **Enable/Disable** targets `TEXTURE_GEN_*` (section 2.12.4, **Material***

Light*, **LightModel***, and **ColorMaterial**, and **Enable/Disable** targets `LIGHTING`, `VERTEX_PROGRAM_TWO_SIDE`, `LIGHTi`, and `COLOR_MATERIAL` (sections 2.19.2 and 2.19.3; and all associated fixed-function vertex array, multitexture, matrix and matrix stack, normal and texture coordinate, and lighting state. A vertex shader must be defined in order to draw primitives.

Language referring to edge flags in the current specification is modified as though all edge flags are `TRUE`.

Note that the **FrontFace**, **ClampColor**, and **ShadeModel** commands in section 2.19 are **not** deprecated, as they still affect other non-deprecated functionality;

- Client vertex arrays - all vertex array attribute pointers must refer to buffer objects (section 2.9.1). The default vertex array object (the name zero) is also deprecated. Calling **VertexAttribPointer** when no buffer object or no vertex array object is bound will generate an `INVALID_OPERATION` error, as will calling any array drawing command when no vertex array object is bound.
- Rectangles - **Rect*** (section 2.11).
- Current raster position - **RasterPos*** and **WindowPos*** (section 2.18), and all associated state.
- Two-sided color selection - **to be done**
- Non-sprite points (section 3.4) - **Enable/Disable** targets `POINT_SMOOTH` and `POINT_SPRITE`, and all associated state. Point rasterization is always performed as though `POINT_SPRITE` were enabled.
- Wide lines and line stipple - **LineWidth** is not deprecated, but values greater than 1.0 will generate an `INVALID_VALUE` error; **LineStipple** and **Enable/Disable** target `LINE_STIPPLE` (section 3.5.2, and all associated state.
- Quadrilateral and polygon primitives - vertex array drawing modes `POLYGON`, `QUADS`, and `QUAD_STRIP` (section 2.6.1, related descriptions of rasterization of non-triangle polygons in section 3.6, and all associated state.
- Separate polygon draw mode - **PolygonMode** *face* values of `FRONT` and `BACK`; polygons are always drawn in the same mode, no matter which face is being rasterized.
- Polygon Stipple - **PolygonStipple** and **Enable/Disable** target `POLYGON_STIPPLE` (section 3.6.2, and all associated state.

- Pixel transfer modes and operations - all pixel transfer modes, including pixel maps, shift and bias, color table lookup, color matrix, and convolution commands and state (sections 3.7.2, 3.7.3, and 3.7.5), and all associated state and commands defining that state.
- Pixel drawing - **DrawPixels** and **PixelZoom** (section 3.7.4). However, the language describing pixel rectangles in section 3.7 is retained as it is required for **TexImage*** and **ReadPixels**.
- Bitmaps - **Bitmap** (section 3.8).
- Legacy OpenGL 1.0 pixel formats - the values 1, 2, 3, and 4 are no longer accepted as internal formats by **TexImage*** or any other command taking an internal format argument.
- Legacy pixel formats - all ALPHA_LUMINANCE, LUMINANCE_ALPHA, and INTENSITY external and internal formats, including compressed, floating-point, and integer variants (see tables 3.6, 3.15, 3.17, 3.19, 3.23, and 6.1); all references to luminance and intensity formats elsewhere in the specification, including conversion to and from those formats; and all associated state, including state describing the allocation or format of luminance and intensity texture or framebuffer components.
- Depth texture mode - DEPTH_TEXTURE_MODE. Section 3.9.14 is to be changed so that *r* is returned to texture samplers directly, and the OpenGL Shading Language 1.30 Specification is to be changed so that (*r*, *r*, *r*, 1) is always returned from depth texture samplers in this case.
- Texture wrap mode CLAMP - CLAMP is no longer accepted as a value of texture parameters TEXTURE_WRAP_S, TEXTURE_WRAP_T, or TEXTURE_WRAP_R.
- Texture borders - the *border* value to **TexImage*** must always be zero, or an INVALID_VALUE error is generated (section 3.9.1); all language in section 3.9 referring to nonzero border widths during texture image specification and texture sampling; **TexParameter*** *target* TEXTURE_BORDER_COLOR and texture wrap mode CLAMP_TO_BORDER (section 3.9.4); and all associated state.
- Automatic mipmap generation - **TexParameter*** *target* GENERATE_MIPMAP (section 3.9.7), and all associated state
- Fixed-function fragment processing - **AreTexturesResident**, **PrioritizeTextures**, and **TexParameter** *target* TEXTURE_PRIORITY; **TexEnv** *target*

TEXTURE_ENV and all associated parameters; **Enable** *targets* of all dimensionalities (TEXTURE_1D, TEXTURE_2D, TEXTURE_3D, TEXTURE_1D_ARRAY, TEXTURE_2D_ARRAY, and TEXTURE_CUBE_MAP); **Enable** target COLOR_SUM; **Enable** *target* FOG, **Fog**, and all associated parameters; and all associated state.

- Alpha test - **AlphaFunc** and **Enable/Disable** target ALPHA_TEST (section 4.1.4), and all associated state.
- Accumulation buffers - **ClearAccum**, and ACCUM_BUFFER_BIT is not valid as a bit in the argument to **Clear** (section 4.2.3); **Accum** (section 4.2.4); the ACCUM_*_BITS framebuffer state describing the size of accumulation buffer components (table 6.51); and all associated state.

Window system-binding APIs such as GLX and WGL may choose to either not expose window configs containing accumulation buffers, or to ignore accumulation buffers when the default framebuffer bound to a GL context contains them.

- Context framebuffer size queries - RED_BITS, GREEN_BITS, BLUE_BITS, ALPHA_BITS, DEPTH_BITS, and STENCIL_BITS.
- Evaluators - **Map***, **EvalCoord***, **MapGrid***, **EvalMesh***, **EvalPoint***, and all evaluator map enables in table 5.1 (section 5.1, and all associated state.
- Selection and feedback modes - **RenderMode**, **InitNames**, **PopName**, **PushName**, **LoadName**, and **SelectBuffer** (section 5.2); **FeedbackBuffer** and **PassThrough** (section 5.3); and all associated state.
- Display lists - **NewList**, **EndList**, **CallList**, **CallLists**, **ListBase**, **GenLists**, **IsList**, and **DeleteLists** (section 5.4); all references to display lists and behavior when compiling commands into display lists elsewhere in the specification; and all associated state.
- Hints - the PERSPECTIVE_CORRECTION_HINT, FOG_HINT, and GENERATE_MIPMAP_HINT targets to **Hint** (section 5.6).
- Attribute stacks - **PushAttrib**, **PushClientAttrib**, **PopAttrib**, **PopClientAttrib**, the MAX_ATTRIB_STACK_DEPTH, MAX_CLIENT_ATTRIB_STACK_DEPTH state, and the values ALL_ATTRIB_BITS and CLIENT_ALL_ATTRIB_BITS (section 6.1.18).
- Unified extension string - EXTENSIONS target to **GetString** (section 6.1.11).

- Token names and queries - all token names and queries not otherwise mentioned above for deprecated state, as well as all query entry points where all valid targets of that query are deprecated state (chapter 6 and the state tables)

Appendix F

Version 1.1

OpenGL version 1.1 is the first revision since the original version 1.0 was released on 1 July 1992. Version 1.1 is upward compatible with version 1.0, meaning that any program that runs with a 1.0 GL implementation will also run unchanged with a 1.1 GL implementation. Several additions were made to the GL, especially to the texture mapping capabilities, but also to the geometry and fragment operations. Following are brief descriptions of each addition.

F.1 Vertex Array

Arrays of vertex data may be transferred to the GL with many fewer commands than were previously necessary. Six arrays are defined, one each storing vertex positions, normal coordinates, colors, color indices, texture coordinates, and edge flags. The arrays may be specified and enabled independently, or one of the pre-defined configurations may be selected with a single command.

The primary goal was to decrease the number of subroutine calls required to transfer non-display listed geometry data to the GL. A secondary goal was to improve the efficiency of the transfer; especially to allow direct memory access (DMA) hardware to be used to effect the transfer. The additions match those of the `GL_EXT_vertex_array` extension, except that static array data are not supported (because they complicated the interface, and were not being used), and the pre-defined configurations are added (both to reduce subroutine count even further, and to allow for efficient transfer of array data).

F.2 Polygon Offset

Depth values of fragments generated by the rasterization of a polygon may be shifted toward or away from the origin, as an affine function of the window coordinate depth slope of the polygon. Shifted depth values allow coplanar geometry, especially facet outlines, to be rendered without depth buffer artifacts. They may also be used by future shadow generation algorithms.

The additions match those of the `GL_EXT_polygon_offset` extension, with two exceptions. First, the offset is enabled separately for `POINT`, `LINE`, and `FILL` rasterization modes, all sharing a single affine function definition. (Shifting the depth values of the outline fragments, instead of the fill fragments, allows the contents of the depth buffer to be maintained correctly.) Second, the offset bias is specified in units of depth buffer resolution, rather than in the $[0,1]$ depth range.

F.3 Logical Operation

Fragments generated by RGBA rendering may be merged into the framebuffer using a logical operation, just as color index fragments are in GL version 1.0. Blending is disabled during such operation because it is rarely desired, because many systems could not support it, and to match the semantics of the `GL_EXT_blend_logic_op` extension, on which this addition is loosely based.

F.4 Texture Image Formats

Stored texture arrays have a format, known as the *internal format*, rather than a simple count of components. The internal format is represented as a single enumerated value, indicating both the organization of the image data (`LUMINANCE`, `RGB`, etc.) and the number of bits of storage for each image component. Clients can use the internal format specification to suggest the desired storage precision of texture images. New *base internal formats*, `ALPHA` and `INTENSITY`, provide new texture environment operations. These additions match those of a subset of the `GL_EXT_texture` extension.

F.5 Texture Replace Environment

A common use of texture mapping is to replace the color values of generated fragments with texture color data. This could be specified only indirectly in GL version 1.0, which required that client specified “white” geometry be modulated

by a texture. GL version 1.1 allows such replacement to be specified explicitly, possibly improving performance. These additions match those of a subset of the `GL_EXT_texture` extension.

F.6 Texture Proxies

Texture proxies allow a GL implementation to advertise different maximum texture image sizes as a function of some other texture parameters, especially of the internal image format. Clients may use the proxy query mechanism to tailor their use of texture resources at run time. The proxy interface is designed to allow such queries without adding new routines to the GL interface. These additions match those of a subset of the `GL_EXT_texture` extension, except that implementations return allocation information consistent with support for complete mipmap arrays.

F.7 Copy Texture and Subtexture

Texture array data can be specified from framebuffer memory, as well as from client memory, and rectangular subregions of texture arrays can be redefined either from client or framebuffer memory. These additions match those defined by the `GL_EXT_copy_texture` and `GL_EXT_subtexture` extensions.

F.8 Texture Objects

A set of texture arrays and their related texture state can be treated as a single object. Such treatment allows for greater implementation efficiency when multiple arrays are used. In conjunction with the subtexture capability, it also allows clients to make gradual changes to existing texture arrays, rather than completely redefining them. These additions match those of the `GL_EXT_texture_object` extension, with slight additions to the texture residency semantics.

F.9 Other Changes

1. Color indices may now be specified as unsigned bytes.
2. Texture coordinates s , t , and r are divided by q during the rasterization of points, pixel rectangles, and bitmaps. This division was documented only for lines and polygons in the 1.0 version.

3. The line rasterization algorithm was changed so that vertical lines on pixel borders rasterize correctly.
4. Separate pixel transfer discussions in chapter 3 and chapter 4 were combined into a single discussion in chapter 3.
5. Texture alpha values are returned as 1.0 if there is no alpha channel in the texture array. This behavior was unspecified in the 1.0 version, and was incorrectly documented in the reference manual.
6. Fog start and end values may now be negative.
7. Evaluated color values direct the evaluation of the lighting equation if **ColorMaterial** is enabled.

F.10 Acknowledgements

OpenGL 1.1 is the result of the contributions of many people, representing a cross section of the computer industry. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:

Kurt Akeley, Silicon Graphics
Bill Armstrong, Evans & Sutherland
Andy Bigos, 3Dlabs
Pat Brown, IBM
Jim Cobb, Evans & Sutherland
Dick Coulter, Digital Equipment
Bruce D'Amora, GE Medical Systems
John Dennis, Digital Equipment
Fred Fisher, Accel Graphics
Chris Frazier, Silicon Graphics
Todd Frazier, Evans & Sutherland
Tim Freese, NCD
Ken Garnett, NCD
Mike Heck, Template Graphics Software
Dave Higgins, IBM
Phil Huxley, 3Dlabs
Dale Kirkland, Intergraph
Hock San Lee, Microsoft
Kevin LeFebvre, Hewlett Packard
Jim Miller, IBM
Tim Misner, SunSoft

Jeremy Morris, 3Dlabs
Israel Pinkas, Intel
Bimal Poddar, IBM
Lyle Ramshaw, Digital Equipment
Randi Rost, Hewlett Packard
John Schimpf, Silicon Graphics
Mark Segal, Silicon Graphics
Igor Sinyak, Intel
Jeff Stevenson, Hewlett Packard
Bill Sweeney, SunSoft
Kelvin Thompson, Portable Graphics
Neil Trevett, 3Dlabs
Linus Vepstas, IBM
Andy Vesper, Digital Equipment
Henri Warren, Megatek
Paula Womack, Silicon Graphics
Mason Woo, Silicon Graphics
Steve Wright, Microsoft

Appendix G

Version 1.2

OpenGL version 1.2, released on March 16, 1998, is the second revision since the original version 1.0. Version 1.2 is upward compatible with version 1.1, meaning that any program that runs with a 1.1 GL implementation will also run unchanged with a 1.2 GL implementation.

Several additions were made to the GL, especially to texture mapping capabilities and the pixel processing pipeline. Following are brief descriptions of each addition.

G.1 Three-Dimensional Texturing

Three-dimensional textures can be defined and used. In-memory formats for three-dimensional images, and pixel storage modes to support them, are also defined. The additions match those of the `GL_EXT_texture3D` extension.

One important application of three-dimensional textures is rendering volumes of image data.

G.2 BGRA Pixel Formats

`BGRA` extends the list of client memory color formats. Specifically, it provides a component order matching file and framebuffer formats common on Windows platforms. The additions match those of the `GL_EXT_bgra` extension.

G.3 Packed Pixel Formats

Packed pixels in client memory are represented entirely by one unsigned byte, one unsigned short, or one unsigned integer. The fields with the packed pixel are not proper machine types, but the pixel as a whole is. Thus the pixel storage modes and their unpacking counterparts all work correctly with packed pixels.

The additions match those of the `GL_EXT_packed_pixels` extension, with the further addition of reversed component order packed formats.

G.4 Normal Rescaling

Normals may be rescaled by a constant factor derived from the model-view matrix. Rescaling can operate faster than renormalization in many cases, while resulting in the same unit normals.

The additions are based on the `GL_EXT_rescale_normal` extension.

G.5 Separate Specular Color

Lighting calculations are modified to produce a primary color consisting of emissive, ambient and diffuse terms of the usual GL lighting equation, and a secondary color consisting of the specular term. Only the primary color is modified by the texture environment; the secondary color is added to the result of texturing to produce a single post-texturing color. This allows highlights whose color is based on the light source creating them, rather than surface properties.

The additions match those of the `GL_EXT_separate_specular_color` extension.

G.6 Texture Coordinate Edge Clamping

GL normally clamps such that the texture coordinates are limited to exactly the range $[0, 1]$. When a texture coordinate is clamped using this algorithm, the texture sampling filter straddles the edge of the texture image, taking half its sample values from within the texture image, and the other half from the texture border. It is sometimes desirable to clamp a texture without requiring a border, and without using the constant border color.

A new texture clamping algorithm, `CLAMP_TO_EDGE`, clamps texture coordinates at all mipmap levels such that the texture filter never samples a border texel. The color returned when clamping is derived only from texels at the edge of the texture image.

The additions match those of the `GL_SGIS_texture_edge_clamp` extension.

G.7 Texture Level of Detail Control

Two constraints related to the texture level of detail parameter λ are added. One constraint clamps λ to a specified floating point range. The other limits the selection of mipmap image arrays to a subset of the arrays that would otherwise be considered.

Together these constraints allow a large texture to be loaded and used initially at low resolution, and to have its resolution raised gradually as more resolution is desired or available. Image array specification is necessarily integral, rather than continuous. By providing separate, continuous clamping of the λ parameter, it is possible to avoid "popping" artifacts when higher resolution images are provided.

The additions match those of the `GL_SGIS_texture_lod` extension.

G.8 Vertex Array Draw Element Range

A new form of **DrawElements** that provides explicit information on the range of vertices referred to by the index set is added. Implementations can take advantage of this additional information to process vertex data without having to scan the index data to determine which vertices are referenced.

The additions match those of the `GL_EXT_draw_range_elements` extension.

G.9 Imaging Subset

The remaining new features are primarily intended for advanced image processing applications, and may not be present in all GL implementations. They are collectively referred to as the *imaging subset*.

G.9.1 Color Tables

A new RGBA-format color lookup mechanism is defined in the pixel transfer process, providing additional lookup capabilities beyond the existing lookup. The key difference is that the new lookup tables are treated as one-dimensional images with internal formats, like texture images and convolution filter images. Thus the new tables can operate on a subset of the components of passing pixel groups. For example, a table with internal format `ALPHA` modifies only the A component of each pixel group, leaving the R, G, and B components unmodified.

Three independent lookups may be performed: prior to convolution; after convolution and prior to color matrix transformation; after color matrix transformation and prior to gathering pipeline statistics.

Methods to initialize the color lookup tables from the framebuffer, in addition to the standard memory source mechanisms, are provided.

Portions of a color lookup table may be redefined without reinitializing the entire table. The affected portions may be specified either from client memory or from the framebuffer.

The additions match those of the `GL_EXT_color_table` and `GL_EXT_color_subtable` extensions.

G.9.2 Convolution

One- or two-dimensional convolution operations are executed following the first color table lookup in the pixel transfer process. The convolution kernels are themselves treated as one- and two-dimensional images, which can be loaded from application memory or from the framebuffer.

The convolution framework is designed to accommodate three-dimensional convolution, but that API is left for a future extension.

The additions match those of the `GL_EXT_convolution` and `GL_HP_convolution_border_modes` extensions.

G.9.3 Color Matrix

A 4x4 matrix transformation and associated matrix stack are added to the pixel transfer path. The matrix operates on RGBA pixel groups, using the equation

$$C' = MC,$$

where

$$C = \begin{pmatrix} R \\ G \\ B \\ A \end{pmatrix}$$

and M is the 4×4 matrix on the top of the color matrix stack. After the matrix multiplication, each resulting color component is scaled and biased by a programmed amount. Color matrix multiplication follows convolution.

The color matrix can be used to reassign and duplicate color components. It can also be used to implement simple color space conversions.

The additions match those of the `GL_SGI_color_matrix` extension.

G.9.4 Pixel Pipeline Statistics

Pixel operations that count occurrences of specific color component values (histogram) and that track the minimum and maximum color component values (min-max) are performed at the end of the pixel transfer pipeline. An optional mode allows pixel data to be discarded after the histogram and/or minmax operations are completed. Otherwise the pixel data continues on to the next operation unaffected.

The additions match those of the `GL_EXT_histogram` extension.

G.9.5 Constant Blend Color

A constant color that can be used to define blend weighting factors may be defined. A typical usage is blending two RGB images. Without the constant blend factor, one image must have an alpha channel with each pixel set to the desired blend factor.

The additions match those of the `GL_EXT_blend_color` extension.

G.9.6 New Blending Equations

Blending equations other than the normal weighted sum of source and destination components may be used.

Two of the new equations produce the minimum (or maximum) color components of the source and destination colors. Taking the maximum is useful for applications such as maximum projection in medical imaging.

The other two equations are similar to the default blending equation, but produce the difference of its left and right hand sides, rather than the sum. Image differences are useful in many image processing applications.

The additions match those of the `GL_EXT_blend_minmax` and `GL_EXT_blend_subtract` extensions.

G.10 Acknowledgements

OpenGL 1.2 is the result of the contributions of many people, representing a cross section of the computer industry. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:

- Kurt Akeley, Silicon Graphics
- Bill Armstrong, Evans & Sutherland
- Otto Berkes, Microsoft
- Pierre-Luc Bisailon, Matrox Graphics
- Drew Bliss, Microsoft

David Blythe, Silicon Graphics
Jon Brewster, Hewlett Packard
Dan Brokenshire, IBM
Pat Brown, IBM
Newton Cheung, S3
Bill Clifford, Digital
Jim Cobb, Parametric Technology
Bruce D'Amora, IBM
Kevin Dallas, Microsoft
Mahesh Dandapani, Rendition
Daniel Daum, AccelGraphics
Suzy Deffeyes, IBM
Peter Doyle, Intel
Jay Duluk, Raycer
Craig Dunwoody, Silicon Graphics
Dave Erb, IBM
Fred Fisher, AccelGraphics / Dynamic Pictures
Celeste Fowler, Silicon Graphics
Allen Gallotta, ATI
Ken Garnett, NCD
Michael Gold, Nvidia / Silicon Graphics
Craig Groeschel, Metro Link
Jan Hardenbergh, Mitsubishi Electric
Mike Heck, Template Graphics Software
Dick Hessel, Raycer Graphics
Paul Ho, Silicon Graphics
Shawn Hopwood, Silicon Graphics
Jim Hurley, Intel
Phil Huxley, 3Dlabs
Dick Jay, Template Graphics Software
Paul Jensen, 3Dfx
Brett Johnson, Hewlett Packard
Michael Jones, Silicon Graphics
Tim Kelley, Real3D
Jon Khazam, Intel
Louis Khouw, Sun
Dale Kirkland, Intergraph
Chris Kitrick, Raycer
Don Kuo, S3
Herb Kuta, Quantum 3D

Phil Lacroute, Silicon Graphics
Prakash Ladia, S3
Jon Leech, Silicon Graphics
Kevin Lefebvre, Hewlett Packard
David Ligon, Raycer Graphics
Kent Lin, S3
Dan McCabe, S3
Jack Middleton, Sun
Tim Misner, Intel
Bill Mitchell, National Institute of Standards
Jeremy Morris, 3Dlabs
Gene Munce, Intel
William Newhall, Real3D
Matthew Papakipos, Nvidia / Raycer
Garry Paxinos, Metro Link
Hanspeter Pfister, Mitsubishi Electric
Richard Pimentel, Parametric Technology
Bimal Poddar, IBM / Intel
Rob Putney, IBM
Mike Quinlan, Real3D
Nate Robins, University of Utah
Detlef Roettger, Elsa
Randi Rost, Hewlett Packard
Kevin Rushforth, Sun
Richard S. Wright, Real3D
Hock San Lee, Microsoft
John Schimpf, Silicon Graphics
Stefan Seeboth, ELSA
Mark Segal, Silicon Graphics
Bob Seitsinger, S3
Min-Zhi Shao, S3
Colin Sharp, Rendition
Igor Sinyak, Intel
Bill Sweeney, Sun
William Sweeney, Sun
Nathan Tuck, Raycer
Doug Twillenger, Sun
John Tynefeld, 3dfx
Kartik Venkataraman, Intel
Andy Vesper, Digital Equipment

Henri Warren, Digital Equipment / Megatek
Paula Womack, Silicon Graphics
Steve Wright, Microsoft
David Yu, Silicon Graphics
Randy Zhao, S3

Appendix H

Version 1.2.1

OpenGL version 1.2.1, released on October 14, 1998, introduced ARB extensions (see Appendix [O](#)). The only ARB extension defined in this version is multitexture, allowing application of multiple textures to a fragment in one rendering pass. Multitexture is based on the `GL_SGIS_multitexture` extension, simplified by removing the ability to route texture coordinate sets to arbitrary texture units.

A new corollary discussing display list and immediate mode invariance was added to Appendix [B](#) on April 1, 1999.

Appendix I

Version 1.3

OpenGL version 1.3, released on August 14, 2001, is the third revision since the original version 1.0. Version 1.3 is upward compatible with earlier versions, meaning that any program that runs with a 1.2, 1.1, or 1.0 GL implementation will also run unchanged with a 1.3 GL implementation.

Several additions were made to the GL, especially texture mapping capabilities previously defined by ARB extensions. Following are brief descriptions of each addition.

I.1 Compressed Textures

Compressing texture images can reduce texture memory utilization and improve performance when rendering textured primitives. The GL provides a framework upon which extensions providing specific compressed image formats can be built, and a set of generic compressed internal formats that allow applications to specify that texture images should be stored in compressed form without needing to code for specific compression formats (specific compressed formats, such as S3TC or FXT1, are supported by extensions).

Texture compression was promoted from the `GL_ARB_texture_compression` extension.

I.2 Cube Map Textures

Cube map textures provide a new texture generation scheme for looking up textures from a set of six two-dimensional images representing the faces of a cube. The (*str*) texture coordinates are treated as a direction vector emanating from the center of a cube. At texture generation time, the interpolated per-fragment (*str*) selects

one cube face two-dimensional image based on the largest magnitude coordinate (the major axis). A new (st) is calculated by dividing the two other coordinates (the minor axes values) by the major axis value, and the new (st) is used to lookup into the selected two-dimensional texture image face of the cube map.

Two new texture coordinate generation modes are provided for use in conjunction with cube map texturing. The `REFLECTION_MAP` mode generates texture coordinates (str) matching the vertex's eye-space reflection vector, useful for environment mapping without the singularity inherent in `SPHERE_MAP` mapping. The `NORMAL_MAP` mode generates texture coordinates matching the vertex's transformed eye-space normal, useful for texture-based diffuse lighting models.

Cube mapping was promoted from the `GL_ARB_texture_cube_map` extension.

I.3 Multisample

Multisampling provides a antialiasing mechanism which samples all primitives multiple times at each pixel. The color sample values are resolved to a single, displayable color each time a pixel is updated, so antialiasing appears to be automatic at the application level. Because each sample includes depth and stencil information, the depth and stencil functions perform equivalently to the single-sample mode.

When multisampling is supported, an additional buffer, called the multisample buffer, is added to the framebuffer. Pixel sample values, including color, depth, and stencil values, are stored in this buffer.

Multisampling is usually an expensive operation, so it is usually not supported on all contexts. Applications must obtain a multisample-capable context using the new interfaces provided by GLX 1.4 or by the `WGL_ARB_multisample` extension.

Multisampling was promoted from the `GL_ARB_multisample` extension; The definition of the extension was changed slightly to support both multisampling and supersampling implementations.

I.4 Multitexture

Multitexture adds support for multiple texture units. The capabilities of the multiple texture units are identical, except that evaluation and feedback are supported only for texture unit 0. Each texture unit has its own state vector which includes texture vertex array specification, texture image and filtering parameters, and texture environment application.

The texture environments of the texture units are applied in a pipelined fashion whereby the output of one texture environment is used as the input fragment color

for the next texture environment. Changes to texture client state and texture server state are each routed through one of two selectors which control which instance of texture state is affected.

Multitexture was promoted from the `GL_ARB_multitexture` extension.

I.5 Texture Add Environment Mode

The `TEXTURE_ENV_MODE` texture environment function `ADD` provides a texture function to add incoming fragment and texture source colors.

Texture add mode was promoted from the `GL_ARB_texture_env_add` extension.

I.6 Texture Combine Environment Mode

The `TEXTURE_ENV_MODE` texture environment function `COMBINE` provides a wide range of programmable combiner functions using the incoming fragment color, texture source color, texture constant color, and the result of the previous texture environment stage as possible parameters.

Combiner operations include passthrough, multiplication, addition and biased addition, subtraction, and linear interpolation of specified parameters. Different combiner operations may be selected for RGB and A components, and the final result may be scaled by 1, 2, or 4.

Texture combine was promoted from the `GL_ARB_texture_env_combine` extension.

I.7 Texture Dot3 Environment Mode

The `TEXTURE_ENV_MODE` `COMBINE` operations also provide three-component dot products of specified parameters, with the resulting scalar value replicated into the RGB or RGBA components of the output color. The dot product is performed using pseudo-signed arithmetic to enable per-pixel lighting computations.

Texture DOT3 mode was promoted from the `GL_ARB_texture_env_dot3` extension.

I.8 Texture Border Clamp

The texture wrap parameter `CLAMP_TO_BORDER` mode clamps texture coordinates at all mipmap levels such that when the texture filter straddles an edge of the texture

image, the color returned is derived only from border texels. This behavior mirrors the behavior of the texture edge clamp mode introduced by OpenGL 1.2.

Texture border clamp was promoted from the `GL_ARB_texture_border_clamp` extension.

I.9 Transpose Matrix

New functions and tokens are added allowing application matrices stored in row major order rather than column major order to be transferred to the implementation. This allows an application to use standard C-language 2-dimensional arrays and have the array indices match the expected matrix row and column indexes. These arrays are referred to as transpose matrices since they are the transpose of the standard matrices passed to OpenGL.

Transpose matrix adds an interface for transferring data to and from the OpenGL pipeline. It does not change any OpenGL processing or imply any changes in state representation.

Transpose matrix was promoted from the `GL_ARB_transpose_matrix` extension.

I.10 Acknowledgements

OpenGL 1.3 is the result of the contributions of many people. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:

- Adrian Muntianu, ATI
- Al Reyes, 3dfx
- Alain Bouchard, Matrox
- Alan Commike, SGI
- Alan Heirich, Compaq
- Alex Herrera, SP3D
- Allen Akin, VA Linux
- Allen Gallotta, ATI
- Alligator Descartes, Arcane
- Andy Vesper, MERL
- Andy Wolf, Diamond Multimedia
- Axel Schildan, S3
- Barthold Lichtenbelt, 3Dlabs
- Benj Lipchak, Compaq
- Bill Armstrong, Evans & Sutherland

Bill Clifford, Intel
Bill Mannel, SGI
Bimal Poddar, Intel
Bob Beretta, Apple
Brent Insko, NVIDIA
Brian Goldiez, UCF
Brian Greenstone, Apple
Brian Paul, VA Linux
Brian Sharp, GLSetup
Bruce D'Amora, IBM
Bruce Stockwell, Compaq
Chris Brady, Alt.software
Chris Frazier, Raycer
Chris Hall, 3dlabs
Chris Hecker, GLSetup
Chris Lane, Intel
Chris Thornborrow, PixelFusion
Christopher Fraser, IMG
Chuck Smith, Intellgraphics
Craig Dunwoody, SGI
Dairsie Latimer, PixelFusion
Dale Kirkland, 3Dlabs / Intergraph
Dan Brokenshire, IBM
Dan Ginsburg, ATI
Dan McCabe, S3
Dave Aronson, Microsoft
Dave Gosselin, ATI
Dave Shreiner, SGI
Dave Zenz, Dell
David Aronson, Microsoft
David Blythe, SGI
David Kirk, NVIDIA
David Story, SGI
David Yu, SGI
Deanna Hohn, 3dfx
Dick Coulter, Silicon Magic
Don Mullis, 3dfx
Eamon O Dea, PixelFusion
Edward (Chip) Hill, Pixelfusion
Eiji Obata, NEC

Elio Del Giudice, Matrox
Eric Young, S3
Evan Hart, ATI
Fred Fisher, 3dLabs
Garry Paxinos, Metro Link
Gary Tarolli, 3dfx
George Kyriazis, NVIDIA
Graham Connor, IMG
Herb Kuta, Quantum3D
Howard Miller, Apple
Igor Sinyak, Intel
Jack Middleton, Sun
James Bowman, 3dfx
Jan C. Hardenbergh, MERL
Jason Mitchell, ATI
Jeff Weyman, ATI
Jeffrey Newquist, 3dfx
Jens Owen, Precision Insight
Jeremy Morris, 3Dlabs
Jim Bushnell, Pyramid Peak
John Dennis, Sharp Eye
John Metcalfe, IMG
John Stauffer, Apple
John Tynan, PixelFusion
John W. Polick, NEC
Jon Khazam, Intel
Jon Leech, SGI
Jon Paul Schelter, Matrox
Karl Hilleslad, NVIDIA
Kelvin Thompson
Ken Cameron, Pixelfusion
Ken Dyke, Apple
Ken Nicholson, SGI
Kent Lin, Intel
Kevin Lefebvre, HP
Kevin Martin, VA Linux
Kurt Akeley, SGI
Les Silvern, NEC
Mahesh Dandipani, Rendition
Mark Kilgard, NVIDIA

Martin Amon, 3dfx
Martina Sourada, ATI
Matt Lavoie, Pixelfusion
Matt Russo, Matrox
Matthew Papakipos, NVIDIA
Michael Gold, NVIDIA
Miriam Geller, SGI
Morgan Von Essen, Metro Link
Naruki Aruga, PFU
Nathan Tuck, Raycer Graphics
Neil Trevett, 3Dlabs
Newton Cheung, S3
Nick Triantos, NVIDIA
Patrick Brown, Intel
Paul Jensen, 3dfx
Paul Keller, NVIDIA
Paul Martz, HP
Paula Womack, 3dfx
Peter Doenges, Evans & Sutherland
Peter Graffagnino, Apple
Phil Huxley, 3Dlabs
Ralf Biermann, Elsa AG
Randi Rost, 3Dlabs
Renee Rashid, Micron
Rich Johnson, HP
Richard Pimentel, PTC
Richard Schlein, Apple
Rick Hammerstone, ATI
Rik Faith, VA Linux
Rob Glidden, Sun
Rob Wheeler, 3dfx
Shari Petersen, Rendition
Shawn Hopwood, SGI
Steve Glickman, Silicon Magic
Steve McGuigan, SGI
Steve Wright, Microsoft
Stuart Anderson, Metro Link
T. C. Zhao, MERL
Teri Morrison, HP
Thomas Fox, IBM

Tim Kelley, Real 3D
Tom Frisinger, ATI
Victor Vedovato, Micron
Vikram Simha, MERL
Yanjun Zhang, Sun
Zahid Hussain, TI

Appendix J

Version 1.4

OpenGL version 1.4, released on July 24, 2002, is the fourth revision since the original version 1.0. Version 1.4 is upward compatible with earlier versions, meaning that any program that runs with a 1.3, 1.2, 1.1, or 1.0 GL implementation will also run unchanged with a 1.4 GL implementation.

In addition to numerous additions to the classical fixed-function GL pipeline in OpenGL 1.4, the OpenGL ARB also approved the `GL_ARB_vertex_program` extension, which supports programmable vertex processing. Following are brief descriptions of each addition to OpenGL 1.4; see Chapter [O](#) for a description of `GL_ARB_vertex_program`.

J.1 Automatic Mipmap Generation

Setting the texture parameter `GENERATE_MIPMAP` to `TRUE` introduces a side effect to any modification of the $level_{base}$ of a mipmap array, wherein all higher levels of the mipmap pyramid are recomputed automatically by successive filtering of the base level array.

Automatic mipmap generation was promoted from the `GL_SGIS_generate_mipmap` extension.

J.2 Blend Squaring

Blend squaring extends the set of supported source and destination blend functions to permit squaring RGB and alpha values during blending. Functions `SRC_COLOR` and `ONE_MINUS_SRC_COLOR` are added to the allowed source blending functions,

and `DST_COLOR` and `ONE_MINUS_DST_COLOR` are added to the allowed destination blending functions.

Blend squaring was promoted from the `GL_NV_blend_square` extension.

J.3 Changes to the Imaging Subset

The subset of blending features described by **BlendEquation**, **BlendColor**, and the **BlendFunc** *modes* `CONSTANT_COLOR`, `ONE_MINUS_CONSTANT_COLOR`, `CONSTANT_ALPHA`, and `ONE_MINUS_CONSTANT_ALPHA` are now supported. These feature were available only in the optional imaging subset in versions 1.2 and 1.3 of the GL.

J.4 Depth Textures and Shadows

Depth textures define a new texture internal format, `DEPTH`, normally used to represent depth values. Applications include image-based shadow casting, displacement mapping, and image-based rendering.

Image-based shadowing is enabled with a new texture application mode defined by the parameter `TEXTURE_COMPARE_MODE`. This mode enables comparing texture *r* coordinates to depth texture values to generate a boolean result.

Depth textures and shadows were promoted from the `GL_ARB_depth_texture` and `GL_ARB_shadow` extensions.

J.5 Fog Coordinate

A new associated vertex and fragment datum, the *fog coordinate* may be used in computing fog for a fragment, instead of using eye distance to the fragment, by specifying the coordinate with the **FogCoord** commands and setting the `FOG_COORDINATE_SOURCE` fog parameter. Fog coordinates are particularly useful in computing more complex fog models.

Fog coordinate was promoted from the `GL_EXT_fog_coord` extension.

J.6 Multiple Draw Arrays

Multiple primitives may be drawn in a single call using the **MultiDrawArrays** and **MultiDrawElements** commands.

Multiple draw arrays was promoted from the `GL_EXT_multi_draw_arrays` extension.

J.7 Point Parameters

Point parameters defined by the **PointParameter** commands support additional geometric characteristics of points, allowing the size of a point to be affected by linear or quadratic distance attenuation, and increasing control of the mapping from point size to raster point area and point transparency. This effect may be used for distance attenuation in rendering particles or light points.

Point parameters was promoted from the `GL_ARB_point_parameters` extension.

J.8 Secondary Color

The secondary color may be varied even when lighting is disabled by specifying it as a vertex parameter with the **SecondaryColor** commands.

Secondary color was promoted from the `GL_EXT_secondary_color` extension.

J.9 Separate Blend Functions

Blending capability is extended with **BlendFuncSeparate** to allow independent setting of the RGB and alpha blend functions for blend operations that require source and destination blend factors.

Separate blend functions was promoted from the `GL_EXT_blend_func_separate` extension.

J.10 Stencil Wrap

New stencil operations `INCR_WRAP` and `DECR_WRAP` allow the stencil value to wrap around the range of stencil values instead of saturating to the minimum or maximum values on decrement or increment. Stencil wrapping is needed for algorithms that use the stencil buffer for per-fragment inside-outside primitive computations.

Stencil wrap was promoted from the `GL_EXT_stencil_wrap` extension.

J.11 Texture Crossbar Environment Mode

Texture crossbar extends the texture combine environment mode `COMBINE` by allowing use of the texture color from different texture units as sources to the texture combine function.

Texture environment crossbar was promoted from the `GL_ARB_texture_env_crossbar` extension.

J.12 Texture LOD Bias

The texture filter control parameter `TEXTURE_LOD_BIAS` may be set to bias the computed λ parameter used in texturing for mipmap level of detail selection, providing a means to blur or sharpen textures. LOD bias may be used for depth of field and other special visual effects, as well as for some types of image processing.

Texture LOD bias was based on the `GL_EXT_texture_lod_bias` extension, with the addition of a second per-texture object bias term.

J.13 Texture Mirrored Repeat

Texture mirrored repeat extends the set of texture wrap modes with the mode `MIRRORED_REPEAT`. This effectively defines a texture map twice as large as the original texture image in which the additional half, for each mirrored texture coordinate, is a mirror image of the original texture. Mirrored repeat can be used seamless tiling of a surface.

Texture mirrored repeat was promoted from the `GL_ARB_texture_mirrored_repeat` extension.

J.14 Window Raster Position

The raster position may be set directly to specified window coordinates with the **WindowPos** commands, bypassing the transformation applied to **RasterPos**. Window raster position is particularly useful for imaging and other 2D operations.

Window raster position was promoted from the `GL_ARB_window_pos` extension.

J.15 Acknowledgements

OpenGL 1.4 is the result of the contributions of many people. Following is a partial list of the contributors, including the company that they represented at the time of their contribution. The editor especially thanks Bob Beretta and Pat Brown for their sustained efforts in leading the `GL_ARB_vertex_program` working group,

without which this critical extension could not have been defined and approved in conjunction with OpenGL 1.4.

Kurt Akeley, NVIDIA
Allen Akin
Bill Armstrong, Evans & Sutherland
Ben Ashbaugh, Intel
Chris Bentley, ATI
Bob Beretta, Apple
Daniel Brokenshire, IBM
Pat Brown, NVIDIA
Bill Clifford, Intel
Graham Connor, Videologic
Matt Craighead, NVIDIA
Suzy Deffeyes, IBM
Jean-Luc Dery, Discreet
Kenneth Dyke, Apple
Cass Everitt, NVIDIA
Allen Gallotta, ATI
Lee Gross, IBM
Evan Hart, ATI
Chris Hecker, Definition 6
Alan Heirich, Compaq / HP
Gareth Hughes, VA Linux
Michael I Gold, NVIDIA
Rich Johnson, HP
Mark Kilgard, NVIDIA
Dale Kirkland, 3Dlabs
David Kirk, NVIDIA
Christian Laforte, Alias—Wavefront
Luc Leblanc, Discreet
Jon Leech, SGI
Bill Licea-Kane, ATI
Barthold Lichtenbelt, 3Dlabs
Jack Middleton, Sun
Howard Miller, Apple
Jeremy Morris, 3Dlabs
Jon Paul Schelter, Matrox
Brian Paul, VA Linux / Tungsten Graphics
Bimal Poddar, Intel
Thomas Roell, Xi Graphics

Randi Rost, 3Dlabs
Jeremy Sandmel, ATI
John Stauffer, Apple
Nick Triantos, NVIDIA
Daniel Vogel, Epic Games
Mason Woo, World Wide Woo
Dave Zenz, Dell

Appendix K

Version 1.5

OpenGL version 1.5, released on July 29, 2003, is the fifth revision since the original version 1.0. Version 1.5 is upward compatible with earlier versions, meaning that any program that runs with a 1.4, 1.3, 1.2, 1.1, or 1.0 GL implementation will also run unchanged with a 1.5 GL implementation.

In addition to additions to the classical fixed-function GL pipeline in OpenGL 1.5, the OpenGL ARB also approved a related set of ARB extensions including the OpenGL Shading Language specification and the `GL_ARB_shader_objects`, `GL_ARB_vertex_shader`, and `GL_ARB_fragment_shader` extensions through which high-level shading language programs can be loaded and used in place of the fixed-function pipeline.

Following are brief descriptions of each addition to OpenGL 1.5. The low-level and high-level shading languages are important adjuncts to the OpenGL core. They are described in more detail in appendix O, and their corresponding ARB extension specifications are available online as described in that appendix.

K.1 Buffer Objects

Buffer objects allow various types of data (especially vertex array data) to be cached in high-performance graphics memory on the server, thereby increasing the rate of data transfers to the GL.

Buffer objects were promoted from the `GL_ARB_vertex_buffer_object` extension.

K.2 Occlusion Queries

An occlusion query is a mechanism whereby an application can query the number of pixels (or, more precisely, samples) drawn by a primitive or group of primitives. The primary purpose of occlusion queries is to determine the visibility of an object.

Occlusion query was promoted from the `GL_ARB_occlusion_query` extension.

K.3 Shadow Functions

Texture comparison functions are generalized to support all eight binary functions rather than just `LEQUAL` and `GEQUAL`.

Texture comparison functions were promoted from the `GL_EXT_shadow_funcs` extension.

K.4 Changed Tokens

To achieve consistency with the syntax guidelines for OpenGL function and token names, new token names are introduced to be used in place of old, inconsistent names. However, the old token names continue to be supported, for backwards compatibility with code written for previous versions of OpenGL. The new names, and the old names they replace, are shown in table [K.1](#).

K.5 Acknowledgements

OpenGL 1.5 is the result of the contributions of many people. The editor especially thanks the following individuals for their sustained efforts in leading ARB working groups essential to the success of OpenGL 1.5 and of ARB extensions approved in conjunction with OpenGL 1.5:

Matt Craighead led the working group which created the `GL_ARB_vertex_buffer_object` extension and OpenGL 1.5 core feature. Kurt Akeley wrote the initial specification for the group.

Daniel Ginsburg and Matt Craighead led the working group which created the `GL_ARB_occlusion_query` extension and OpenGL 1.5 core feature.

Benjamin Lipchak led the fragment program working group which created the `GL_ARB_fragment_program` extension, completing the low-level programmable shading interface.

Bill Licea-Kane led the GL2 working group which created the high-level programmable shading interface, including the `GL_ARB_fragment_shader`,

New Token Name	Old Token Name
FOG_COORD_SRC	FOG_COORDINATE_SOURCE
FOG_COORD	FOG_COORDINATE
CURRENT_FOG_COORD	CURRENT_FOG_COORDINATE
FOG_COORD_ARRAY_TYPE	FOG_COORDINATE_ARRAY_TYPE
FOG_COORD_ARRAY_STRIDE	FOG_COORDINATE_ARRAY_STRIDE
FOG_COORD_ARRAY_POINTER	FOG_COORDINATE_ARRAY_POINTER
FOG_COORD_ARRAY	FOG_COORDINATE_ARRAY
FOG_COORD_ARRAY_BUFFER_BINDING	FOG_COORDINATE_ARRAY_BUFFER_BINDING
SRC0_RGB	SOURCE0_RGB
SRC1_RGB	SOURCE1_RGB
SRC2_RGB	SOURCE2_RGB
SRC0_ALPHA	SOURCE0_ALPHA
SRC1_ALPHA	SOURCE1_ALPHA
SRC2_ALPHA	SOURCE2_ALPHA

Table K.1: New token names and the old names they replace.

`GL_ARB_shader_objects`, and `GL_ARB_vertex_shader` extensions and the OpenGL Shading Language.

John Kessenich was the principal editor of the OpenGL Shading Language specification for the GL2 working group, starting from the initial glslang proposal written by John, Dave Baldwin, and Randi Rost.

A partial list of other contributors, including the company that they represented at the time of their contribution, follows:

- Kurt Akeley, NVIDIA
- Allen Akin
- Chad Anson, Dell Computer
- Bill Armstrong, Evans & Sutherland
- Ben Ashbaugh, Intel
- Dave Baldwin, 3Dlabs
- Chris Bentley, ATI
- Bob Beretta, Apple
- David Blythe
- Alain Bouchard, Matrox
- Daniel Brokenshire, IBM
- Pat Brown, NVIDIA
- John Carmack, Id Software

Paul Carmichael, NVIDIA
Bob Carwell, IBM
Paul Clarke, IBM
Bill Clifford, Intel
Roger Cloud, SGI
Graham Connor, Power VR
Matt Craighead, NVIDIA
Doug Crisman, SGI
Matt Cruikshank, Vital Images
Deron Dann Johnson, Sun
Suzy Deffeyes, IBM
Steve Demlow, Vital Images
Joe Deng, SiS
Jean-Luc Dery, Discreet
Kenneth Dyke, Apple
Brian Emberling, Sun
Cass Everitt, NVIDIA
Brandon Fliflet, Intel
Allen Gallotta, ATI
Daniel Ginsburg, ATI
Steve Glanville, NVIDIA
Peter Graffagnino, Apple
Lee Gross, IBM
Rick Hammerstone, ATI
Evan Hart, ATI
Chris Hecker, Definition 6
Alan Heirich, HP
Gareth Hughes, NVIDIA
Michael I Gold, NVIDIA
John Jarvis, Alt.software
Rich Johnson, HP
John Kessenich, 3Dlabs
Mark Kilgard, NVIDIA
Dale Kirkland, 3Dlabs
Raymond Klassen, Intel
Jason Knipe, Bioware
Jayant Kolhe, NVIDIA
Steve Koren, 3Dlabs
Bob Kuehne, SGI
Christian Laforte, Alias

Luc Leblanc, Discreet
Jon Leech, SGI
Kevin Lefebvre, HP
Bill Licea-Kane, ATI
Barthold Lichtenbelt, 3Dlabs
Kent Lin, Intel
Benjamin Lipchak, ATI
Rob Mace, ATI
Bill Mark, NVIDIA
Michael McCool, U. Waterloo
Jack Middleton, Sun
Howard Miller, Apple
Teri Morrison, HP / 3Dlabs
Marc Olano, SGI / U. Maryland
Jean-Francois Panisset, Discreet
Jon Paul Schelter, Matrox
Brian Paul, Tungsten Graphics
Scott Peterson, HP
Bimal Poddar, Intel
Thomas Roell, Xi Graphics
Phil Rogers, ATI
Ian Romanick, IBM
John Rosasco, Apple
Randi Rost, 3Dlabs
Matt Russo, Matrox
Jeremy Sandmel, ATI
Paul Sargent, 3Dlabs
Folker Schamel, Spinor GMBH
Michael Schulman, Sun
John Scott, Raven Software
Avinash Seetharamaiah, Intel
John Spitzer, NVIDIA
Vlad Stamate, Power VR
Michelle Stamnes, Intel
John Stauffer, Apple
Eskil Steenberg, Obsession
Bruce Stockwell, HP
Christopher Tan, IBM
Ray Tice, Avid
Pierre P. Tremblay, Discreet

Neil Trevett, 3Dlabs
Nick Triantos, NVIDIA
Douglas Twilleager, Sun
Shawn Underwood, SGI
Steve Urquhart, Intellgraphics
Victor Vedovato, ATI
Daniel Vogel, Epic Games
Mik Wells, Softimage
Helene Workman, Apple
Dave Zenz, Dell
Karel Zuiderveld, Vital Images

Appendix L

Version 2.0

OpenGL version 2.0, released on September 7, 2004, is the sixth revision since the original version 1.0. Despite incrementing the major version number (to indicate support for high-level programmable shaders), version 2.0 is upward compatible with earlier versions, meaning that any program that runs with a 1.5, 1.4, 1.3, 1.2, 1.1, or 1.0 GL implementation will also run unchanged with a 2.0 GL implementation.

Following are brief descriptions of each addition to OpenGL 2.0.

L.1 Programmable Shading

The OpenGL Shading Language, and the related APIs to create, manage, and use programmable shaders written in the Shading Language, were promoted to core features in OpenGL 2.0. The complete list of features related to programmable shading includes:

L.1.1 Shader Objects

Shader objects provides mechanisms necessary to manage shader and program objects. Shader objects were promoted from the `GL_ARB_shader_objects` extension.

L.1.2 Shader Programs

Vertex and fragment shader programs may be written in the high-level OpenGL Shading Language, replacing fixed-functionality vertex and fragment processing respectively. Vertex and fragment shader programs were promoted from the `GL_ARB_vertex_shader` and `GL_ARB_fragment_shader` extensions.

L.1.3 OpenGL Shading Language

The OpenGL Shading Language is a high-level, C-like language used to program the vertex and fragment pipelines. The Shading Language Specification defines the language proper, while OpenGL API features control how vertex and fragment programs interact with the fixed-function OpenGL pipeline and how applications manage those programs.

OpenGL 2.0 implementations must support at least revision 1.10 of the OpenGL Shading Language. Implementations may query the `SHADING_LANGUAGE_VERSION` string to determine the exact version of the language supported. The OpenGL Shading Language was promoted from the `GL_ARB_shading_language_100` extension (the shading language itself is specified in a companion document; due to the way it's written, that document did not need to be changed as a consequence of promoting programmable shading to the OpenGL core).

L.1.4 Changes To Shader APIs

Small changes to the APIs for managing shader and program objects were made in the process of promoting the shader extensions to the OpenGL 2.0 core. These changes do not affect the functionality of the shader APIs, but include use of the existing `uint` core GL type rather than the new `handleARB` type introduced by the extensions, and changes in some function names, for example mapping the extension function `CreateShaderObjectARB` into the core function `CreateShader`.

L.2 Multiple Render Targets

Programmable shaders may write different colors to multiple output color buffers in a single pass. Multiple render targets was promoted from the `GL_ARB_draw_buffers` extension.

L.3 Non-Power-Of-Two Textures

The restriction of textures to power-of-two dimensions has been relaxed for all texture targets, so that non-power-of-two textures may be specified without generating errors. Non-power-of-two textures was promoted from the `GL_ARB_texture_non_power_of_two` extension.

L.4 Point Sprites

Point sprites replace point texture coordinates with texture coordinates interpolated across the point. This allows drawing points as customized textures, useful for particle systems.

Point sprites were promoted from the `GL_ARB_point_sprite` extension, with the further addition of the `POINT_SPRITE_COORD_ORIGIN` parameter controlling the direction in which the t texture coordinate increases.

L.5 Separate Blend Equation

Blending capability is extended with **BlendEquationSeparate** to allow independent setting of the RGB and alpha blend equations for blend operations.

Separate blend functions was promoted from the `GL_EXT_blend_equation_separate` extension. Note that blend equation `LOGIC_OP` is not supported unless the `GL_EXT_blend_logic_op` extension is supported; `LOGIC_OP` was inadvertently included in the initial release of the OpenGL 2.0 Specification.

L.6 Separate Stencil

Separate stencil functionality may be defined for the front and back faces of primitives, improving performance of shadow volume and Constructive Solid Geometry rendering algorithms.

Separate stencil was based on the the API of the `GL_ATI_separate_stencil` extension, with additional state defined by the similar `GL_EXT_stencil_two_side` extension.

L.7 Other Changes

Several minor revisions and corrections to the OpenGL 1.5 specification were made:

- In section 2.7, **SecondaryColor3** was changed to set A to 1.0 (previously 0.0), so the initial GL state can be restored.
- In section 2.18, transformation was added to the list of steps not performed by **WindowPos**.

- Section 3.9.1 was clarified to mandate that selection of texture internal format must allocate a non-zero number of bits for all components named by the internal format, and zero bits for all other components.
- Tables 3.24 and 3.25 were generalized to multiple textures by replacing C_f with C_p .
- In section 6.1.9, **GetHistogram** was clarified to note that the Final Conversion pixel storage mode is not applied when storing histogram counts.
- The `FOG_COORD_ARRAY_BUFFER_BINDING` enumerant alias was added to table K.1.

After the initial version of the OpenGL 2.0 was released, several more minor corrections were made in the specification revision approved on October 22, 2004:

- Corrected name of the fog source from `FOG_COORD_SRC` to `FOG_COORD` in section 2.18.
- Corrected last parameter type in the declaration of the **UniformMatrix*** commands to `const float *value`, in section 2.20.3.
- Changed the end of the second paragraph of the **Conversion to Fragments** subsection of section 3.7.4, to more clearly describe the set of generated fragments.
- Changed from the older `FOG_COORDINATE` to the newer `FOG_COORD` notation in section 3.11.
- Added `POINT_SPRITE_COORD_ORIGIN` state to table 6.16.
- Changed the description of `MAX_TEXTURE_UNITS` in table 6.48 to reflect its legacy status (referring to the number of fixed-function texture units), and moved it into table 6.49.
- Removed duplicated table entries for `MAX_TEXTURE_IMAGE_UNITS` and `MAX_TEXTURE_COORDS` from table 6.49.
- Added Victor Vedovato to the OpenGL 2.0 Acknowledgements section.
- Miscellaneous typographical corrections.

Additional minor corrections were made in the specification revision approved on February 9, 2005:

- Restored missing language from the depth texture extension in section 6.1.4, allowing `DEPTH_COMPONENT` as a format for texture readbacks.
- Added separate blend equation to the feature list in appendix L. The feature has been in the actual OpenGL 2.0 specification all along, but was omitted from the feature list in the initial specification release.
- Removed `LOGIC_OP` from the allowed blend equations in section 4.1.8 and table 4.1, and adjusted the type of the blend equation state in table 6.25 accordingly.
- Restored missing `VERTEX_ATTRIB_ARRAY_BUFFER_BINDING` state from table 6.8.
- Miscellaneous typographical corrections.

L.8 Acknowledgements

OpenGL 2.0 is the result of the contributions of many people. The editors especially thank the ongoing work of the ARB GL2 working group, led by Bill Licea-Kane and with specifications edited by John Kessenich and Barthold Lichtenbelt, in performing work necessary to promote the OpenGL Shading Language to a core OpenGL feature.

A partial list of other contributors, including the company that they represented at the time of their contribution, follows:

Kurt Akeley, NVIDIA
Allen Akin
Dave Baldwin, 3Dlabs
Bob Beretta, Apple
Pat Brown, NVIDIA
Matt Craighead, NVIDIA
Suzy Deffeyes, IBM
Ken Dyke, Apple
Cass Everitt, NVIDIA
Steve Glanville, NVIDIA
Michael I. Gold, NVIDIA
Evan Hart, ATI
Phil Huxley, 3Dlabs
Deron Dann Johnson, Sun
John Kessenich, 3Dlabs
Mark Kilgard, NVIDIA

Dale Kirkland, 3Dlabs
Steve Koren, 3Dlabs
Jon Leech, SGI
Bill Licea-Kane, ATI
Barthold Lichtenbelt, 3Dlabs
Kent Lin, Intel
Benjamin Lipchak, ATI
Rob Mace, ATI
Michael McCool, U. Waterloo
Jack Middleton, Sun
Jeremy Morris, 3Dlabs
Teri Morrison, 3Dlabs
Marc Olano, SGI / U. Maryland
Glenn Ortner, ATI
Brian Paul, Tungsten Graphics
Bimal Poddar, Intel
Phil Rogers, ATI
Ian Romanick, IBM
Randi Rost, 3Dlabs
Jeremy Sandmel, ATI
Folker Schamel, Spinor GMBH
Geoff Stahl, Apple
Eskil Steenberg, Obsession
Neil Trevett, 3Dlabs
Victor Vedovato, ATI
Mik Wells, Softimage
Esen Yilmaz, Intel
Dave Zenz, Dell

Appendix M

Version 2.1

OpenGL version 2.1, released on August 2, 2006, is the seventh revision since the original version 1.0. Despite incrementing the major version number (to indicate support for high-level programmable shaders), version 2.1 is upward compatible with earlier versions, meaning that any program that runs with a 2.0, 1.5, 1.4, 1.3, 1.2, 1.1, or 1.0 GL implementation will also run unchanged with a 2.0 GL implementation.

Following are brief descriptions of each addition to OpenGL 2.1.

M.1 OpenGL Shading Language

OpenGL 2.1 implementations must support at least revision 1.20 of the OpenGL Shading Language. Implementations may query the `SHADING_LANGUAGE_VERSION` string to determine the exact version of the language supported. Refer to the OpenGL Shading Language Specification for details of the changes between revision 1.10 and 1.20.

M.2 Non-Square Matrices

Added the `UniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}fv` commands in section 2.20.3, allowing specification of non-square uniform matrices.

M.3 Pixel Buffer Objects

Pixel buffer objects expand on the interface provided by the vertex buffer objects, allowing buffer objects to be used with both vertex array and pixel data. This allows

more acceleration opportunities for OpenGL pixel commands.

When a buffer object is bound to the `PIXEL_PACK_BUFFER` target, commands such as **ReadPixels** write their data into a buffer object. When a buffer object is bound to the `PIXEL_UNPACK_BUFFER` target, commands such as **DrawPixels** and **TexImage2D** read their data from a buffer object.

Pixel buffer objects was promoted from the `GL_ARB_pixel_buffer_object` extension. The specification was tightened to consistently require error be generated when read or write operations to a pixel buffer object would run past the end of the buffer, or would be misaligned with respect to the data type being read or written.

M.4 sRGB Textures

New uncompressed and compressed color texture formats with sRGB color components are defined. The sRGB color space is based on typical (non-linear) monitor characteristics expected in a dimly lit office. It has been standardized by the International Electrotechnical Commission (IEC) as IEC 61966-2-1. The sRGB color space roughly corresponds to 2.2 gamma correction.

sRGB textures was promoted from the `GL_EXT_texture_sRGB` extension. Specific compressed sRGB internal formats defined by the extension were not included in OpenGL 2.1, while the generic uncompressed and compressed formats were retained.

M.5 Other Changes

Several minor revisions and corrections to the OpenGL 2.0 specification were made:

- Note that the information log for program objects can include both link and validation information, in section 2.20.2.
- Noted in section 3.7.4 that there is a current raster secondary color, and added the `CURRENT_RASTER_SECONDARY_COLOR` query.
- Required perspective-correct interpolation for all fragment attributes except depth in sections 3.5.1 and 3.6.1, effectively making `GL_PERSPECTIVE_CORRECT_HINT` a no-op.
- Merged specific and generic compressed internal texture format tables into the single table 3.19.

- Changed the type of texture wrap mode and min/mag filter parameters from integer to enum in table 3.20.
- Removed mention of compressed texture depth components from section 3.9.1, since no compressed depth formats are currently defined.
- Added forward reference from section 3.9.5 to section 3.9.14, which defines how depth textures are actually used.
- Remove c notation in table 4.1, and fixed blend equations for FUNC_REVERSE_SUBTRACT.
- Noted in section 6.1.18 that **all** texture object parameters are pushed and popped by **PushAttrib** and **PopAttrib** when TEXTURE_BIT is set in the attribute mask.
- Miscellaneous typographical corrections.

Additional minor corrections were made in a specification revision approved on August 10, 2006, with a few additional fixes and omissions corrected on December 1, 2006:

- Noted in section 2.5 that INVALID_VALUE is generated for negative sizeiptr values.
- Noted in section 2.7 that **VertexAttrib*** can be used to load attributes for any supported matrix type, not just square matrices.
- Removed the description of generating multiple connected components when clipping vertices with w_c values of different signs, at the end of section 2.17. Implementations should not render in the $w_c < 0$ region.
- Added FLOAT_MAT2x3, FLOAT_MAT2x4, FLOAT_MAT3x2, FLOAT_MAT3x4, FLOAT_MAT4x2, and FLOAT_MAT4x3 tokens for non-square matrix types to **GetActiveAttrib** and **GetActiveUniform** in section 2.20.3, and expanded the description of how attribute components are mapped to matrix elements to match.
- Clarified in section 3.9.12 that the texture object passed to **BindTexture** must match the specified *target*, not just the dimensionality of *target*.
- Added missing **TexEnv** targets RGB_SCALE, ALPHA_SCALE, SRCn_RGB, SRCn_ALPHA, OPERANDn_RGB, and OPERANDn_ALPHA in section 3.9.13.

- Noted that `POINT_SPRITE` is a possible *env* parameter to **GetTexEnv** in section 6.1.3.
- Miscellaneous typographical corrections.

Additional minor corrections (in process) were made after December 1, 2006:

- Corrected spelling of `ACTIVE_ATTRIBUTE_MAX_LENGTH` in table 6.40.
- Clarified in section 4.1.12 that logical operations are also performed for each sample when multisampling is active (Khronos internal bug 1022, Bill Licea-Kane).
- Added missing `QUERY_RESULT` and `QUERY_RESULT_AVAILABLE` to state table 6.52.
- Specified in section 3.7.3 that floating-point **PixelMap** entries are clamped to the range $[0, 1]$ when specified (bug 3379, Mark Kilgard).
- Corrected description of type of `TEXTURE_MIN_LOD` and `TEXTURE_MAX_LOD` in section 3.9.11 (bug 2861, Bill Licea-Kane).
- Changed number of sets of texture state from two to four in section 3.9.11 (bug 2862, Bill Licea-Kane).
- Specified in section 5.4 that floating-point offsets to **CallLists** are truncated towards negative infinity (bug 3379, Mark Kilgard).
- Specified that the data conversion rules for integer queries of floating-point state defined in section 6.1.2 also apply to the enumerated queries in section 6.1.3 (bug 3379, Mark Kilgard).
- Removed `DRAW_BUFFER` state from table 6.26 and added descriptions of `DRAW_BUFFERi` and `DRAW_BUFFER` in section 4.2.1 (bug 1029, Jeff Juliano).

M.6 Acknowledgements

OpenGL 2.1 is the result of the contributions of many people. The editor especially thanks the ongoing work of the ARB GLSL working group, led by Bill Licea-Kane and with specifications edited by John Kessenich and Barthold Lichtenbelt, in updating the OpenGL Shading Language to revision 1.20. Ralf Biermann, Derek Cornish, Matt Craighead, and Mark Kilgard edited the `EXT_pixel_buffer_object`

proposal later adopted and developed by the ARB Pixel Buffer Object working group, and Mark Kilgard edited the `EXT_texture_sRGB` extension.

A partial list of other contributors, including the company that they represented at the time of their contribution, follows:

Aaftab Munshi, ATI
Avi Shapira, Graphic Remedy
Barthold Lichtenbelt, 3Dlabs / NVIDIA
Benjamin Lipchak, ATI
Benji Bowman, Imagination Technologies
Bill Armstrong, Evans and Sutherland
Bill Licea-Kane, ATI
Bimal Poddar, Intel
Bob Beretta, Apple
Brian Paul, Tungsten Graphics
Cass Everitt, NVIDIA
Chris Dodd, NVIDIA
Chris Starkey, 3Dlabs
Dale Kirkland, 3Dlabs
Daniel Vogel, Epic Games
Dave Shreiner, SGI
Derek Cornish, NVIDIA
Eskil Steenberg, Obsession
Evan Hart, ATI
Folker Schamel, Spinor GMBH
Geoff Stahl, Apple
Howard Miller, Apple
Ian Romanick, IBM
James A. McCombe, Apple
Jeff Juliano, NVIDIA
Jeff Weyman, ATI
Jeremy Sandmel, Apple / ATI
John Kessenich, 3Dlabs / Intel
John Rosasco, Apple
John Scott
Jon Leech, SGI / Independent
Jon Trulson, Xi Graphics
Ken Severson, NVIDIA
Kenneth Dyke, Apple
Kenneth Russell, Sun
Kent Lin, Intel

Marc Olano, U. Maryland
Mark Kilgard, NVIDIA
Michael Gold, NVIDIA
Neeraj Srivastava, Dell
Neil Trevett, 3Dlabs / NVIDIA
Nick Burns, Apple
Pat Brown, NVIDIA
Paul Martz, SimAuthor
Paul Ramsey, Sun
Pierre Boudier, ATI
Ralf Biermann, NVIDIA
Randi Rost, 3Dlabs
Rob Mace, ATI
Robert Simpson, Bitboys / ATI
Saifuddin Fakhruddin, Intel
Shawn Underwood, SGI
Steve Demlow, Vital Images
Steve Koren, 3Dlabs
Steven Zhu, Intel
Thomas Roell, NVIDIA
Tom Lanzoni, Dell
Travis Bryson, Sun
Yaki Tebeka, Graphic Remedy

Appendix N

Version 3.0

OpenGL version 3.0, released on August 11, 2008, is the eighth revision since the original version 1.0. When using a *full* 3.0 context, OpenGL 3.0 is upward compatible with earlier versions, meaning that any program that runs with a 2.1 or earlier GL implementation will also run unchanged with a 3.0 GL implementation. OpenGL 3.0 context creation is done using a window system binding API, and on most platforms a new command, defined by extensions introduced along with OpenGL 3.0, must be called to create a 3.0 context. Calling the older context creation commands will return an OpenGL 2.1 context. When using a *forward compatible* context, many OpenGL 2.1 features are not supported.

Following are brief descriptions of changes and additions to OpenGL 3.0.

N.1 New Features

New features in OpenGL 3.0, including the extension or extensions if any on which they were based, include:

- API support for the new texture lookup, texture format, and integer and unsigned integer capabilities of the OpenGL Shading Language 1.30 specification (GL_EXT_gpu_shader4).
- Conditional rendering (GL_NV_conditional_render).
- Fine control over mapping buffer subranges into client space and flushing modified data.
- 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`)

N.2 Deprecation Model

OpenGL 3.0 introduces a *deprecation model* in which certain features may be marked as *deprecated*. The deprecation model is described in detail in appendix E, together with a summary of features deprecated in OpenGL 3.0.

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 N.1: New token names and the old names they replace.

N.3 Changed Tokens

New token names are introduced to be used in place of old, inconsistent names. However, the old token names continue to be supported, for backwards compatibility with code written for previous versions of OpenGL. The new names, and the old names they replace, are shown in table [N.1](#).

N.4 Credits and Acknowledgements

OpenGL 3.0 is the result of the contributions of many people and companies. Members of the Khronos OpenGL ARB Working Group during the development of OpenGL 3.0, including the company that they represented at the time of their contributions, follow. Some major contributions made by individuals are listed together with their name, including specific functionality developed in the form of new ARB extensions together with OpenGL 3.0. In addition, many people participated in developing earlier vendor and EXT extensions on which the OpenGL 3.0 functionality is based in part; those individuals are listed in the respective extension specifications in the OpenGL Extension Registry.

Aaftab Munshi, Apple
 Alain Bouchard, Matrox
 Alexis Mather, AMD (Chair, ARB Marketing TSG)
 Andreas Wolf, AMD
 Avi Shapira, Graphic Remedy
 Barthold Lichtenbelt, NVIDIA (Chair, Khronos OpenGL ARB Working Group)
 Benjamin Lipchak, AMD
 Benji Bowman, Imagination Technologies
 Bill Licea-Kane, AMD (Chair, ARB Shading Language TSG)
 Bob Beretta, Apple
 Brent Insko, Intel

Brian Paul, Tungsten Graphics
Bruce Merry, ARM (Detailed specification review)
Cass Everitt, NVIDIA
Chris Dodd, NVIDIA
Daniel Horowitz, NVIDIA
Daniel Koch, Transgaming (Framebuffer objects, half float vertex formats, and instanced rendering)
Daniel Omachi, Apple
Dave Shreiner, ARM
David Garcia, Imagination Technologies
Eric Boumaour, AMD
Eskil Steenberg, Obsession
Evan Hart, NVIDIA
Folker Schamel, Spinor GMBH
Gavriel State, Transgaming
Geoff Stahl, Apple
Georg Kolling, Imagination Technologies
Gregory Prisament, NVIDIA
Guillaume Portier, HI Corp
Ian Romanick, IBM / Intel (Vertex array objects; GLX protocol)
James Helferty, Transgaming (Instanced rendering)
James Jones, NVIDIA
Jamie Gennis, NVIDIA
Jason Green, Transgaming
Jeff Bolz, NVIDIA
Jeff Juliano, NVIDIA
Jeremy Sandmel, Apple (Chair, ARB Nextgen (OpenGL 3.0) TSG)
John Kessenich, Intel (OpenGL Shading Language Specification Editor; 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 O

ARB Extensions

OpenGL extensions that have been approved by the OpenGL Architectural Review Board (ARB) are described in this chapter. These extensions are not required to be supported by a conformant OpenGL implementation, but are expected to be widely available; they define functionality that is likely to move into the required feature set in a future revision of the specification.

In order not to compromise the readability of the core specification, ARB extensions are not integrated into the core language; instead, they are made available online in the *OpenGL Extension Registry* (as are a much larger number of vendor-specific extensions, as well as extensions to window system binding APIs, such as GLX and WGL). Extensions are documented as changes to the Specification. The Registry is available on the World Wide Web at URL

<http://www.opengl.org/registry/>

Brief descriptions of ARB extensions are provided below.

O.1 Naming Conventions

To distinguish ARB extensions from core OpenGL features and from vendor-specific extensions, the following naming conventions are used:

- A unique *name string* of the form "GL_ARB_name" is associated with each extension. If the extension is supported by an implementation, this string will be present in the EXTENSIONS string described in section 6.1.11.
- All functions defined by the extension will have names of the form ***FunctionARB***

- All enumerants defined by the extension will have names of the form `NAME_ARB`.
- In addition to OpenGL extensions, there are also ARB extensions to the related GLX and WGL APIs. Such extensions have name strings prefixed by "GLX_" and "WGL_" respectively. Not all GLX and WGL ARB extensions are described here, but all such extensions are included in the registry.

O.2 Promoting Extensions to Core Features

ARB extensions can be *promoted* to required core features in later revisions of OpenGL. When this occurs, the extension specifications are merged into the core specification. Functions and enumerants that are part of such promoted extensions will have the **ARB** affix removed.

GL implementations of such later revisions should continue to export the name strings of promoted extensions in the `EXTENSIONS` string, and continue to support the **ARB**-affixed versions of functions and enumerants as a transition aid.

For descriptions of extensions promoted to core features in OpenGL 1.3 and beyond, see appendices [I](#), [J](#), [K](#), and [L](#) respectively.

O.3 Multitexture

The name string for multitexture is `GL_ARB_multitexture`. It was promoted to a core feature in OpenGL 1.3.

O.4 Transpose Matrix

The name string for transpose matrix is `GL_ARB_transpose_matrix`. It was promoted to a core feature in OpenGL 1.3.

O.5 Multisample

The name string for multisample is `GL_ARB_multisample`. It was promoted to a core feature in OpenGL 1.3.

O.6 Texture Add Environment Mode

The name string for texture add mode is `GL_ARB_texture_env_add`. It was promoted to a core feature in OpenGL 1.3.

O.7 Cube Map Textures

The name string for cube mapping is `GL_ARB_texture_cube_map`. It was promoted to a core feature in OpenGL 1.3.

O.8 Compressed Textures

The name string for compressed textures is `GL_ARB_texture_compression`. It was promoted to a core feature in OpenGL 1.3.

O.9 Texture Border Clamp

The name string for texture border clamp is `GL_ARB_texture_border_clamp`. It was promoted to a core feature in OpenGL 1.3.

O.10 Point Parameters

The name string for point parameters is `GL_ARB_point_parameters`. It was promoted to a core features in OpenGL 1.4.

O.11 Vertex Blend

Vertex blending replaces the single model-view transformation with multiple vertex units. Each unit has its own transform matrix and an associated current weight. Vertices are transformed by all the enabled units, scaled by their respective weights, and summed to create the eye-space vertex. Normals are similarly transformed by the inverse transpose of the model-view matrices.

The name string for vertex blend is `GL_ARB_vertex_blend`.

O.12 Matrix Palette

Matrix palette extends vertex blending to include a palette of model-view matrices. Each vertex may be transformed by a different set of matrices chosen from the palette.

The name string for matrix palette is `GL_ARB_matrix_palette`.

O.13 Texture Combine Environment Mode

The name string for texture combine mode is `GL_ARB_texture_env_combine`. It was promoted to a core feature in OpenGL 1.3.

O.14 Texture Crossbar Environment Mode

The name string for texture crossbar is `GL_ARB_texture_env_crossbar`. It was promoted to a core features in OpenGL 1.4.

O.15 Texture Dot3 Environment Mode

The name string for DOT3 is `GL_ARB_texture_env_dot3`. It was promoted to a core feature in OpenGL 1.3.

O.16 Texture Mirrored Repeat

The name string for texture mirrored repeat is `GL_ARB_texture_mirrored_repeat`. It was promoted to a core feature in OpenGL 1.4.

O.17 Depth Texture

The name string for depth texture is `GL_ARB_depth_texture`. It was promoted to a core feature in OpenGL 1.4.

O.18 Shadow

The name string for shadow is `GL_ARB_shadow`. It was promoted to a core feature in OpenGL 1.4.

O.19 Shadow Ambient

Shadow ambient extends the basic image-based shadow functionality by allowing a texture value specified by the `TEXTURE_COMPARE_FAIL_VALUE_ARB` texture parameter to be returned when the texture comparison fails. This may be used for ambient lighting of shadowed fragments and other advanced lighting effects.

The name string for shadow ambient is `GL_ARB_shadow_ambient`.

O.20 Window Raster Position

The name string for window raster position is `GL_ARB_window_pos`. It was promoted to a core feature in OpenGL 1.4.

O.21 Low-Level Vertex Programming

Application-defined *vertex programs* may be specified in a new low-level programming language, replacing the standard fixed-function vertex transformation, lighting, and texture coordinate generation pipeline. Vertex programs enable many new effects and are an important first step towards future graphics pipelines that will be fully programmable in an unrestricted, high-level shading language.

The name string for low-level vertex programming is `GL_ARB_vertex_program`.

O.22 Low-Level Fragment Programming

Application-defined *fragment programs* may be specified in the same low-level language as `GL_ARB_vertex_program`, replacing the standard fixed-function vertex texturing, fog, and color sum operations.

The name string for low-level fragment programming is `GL_ARB_fragment_program`.

O.23 Buffer Objects

The name string for buffer objects is `GL_ARB_vertex_buffer_object`. It was promoted to a core feature in OpenGL 1.5.

O.24 Occlusion Queries

The name string for occlusion queries is `GL_ARB_occlusion_query`. It was promoted to a core feature in OpenGL 1.5.

O.25 Shader Objects

The name string for shader objects is `GL_ARB_shader_objects`. It was promoted to a core feature in OpenGL 2.0.

O.26 High-Level Vertex Programming

The name string for high-level vertex programming is `GL_ARB_vertex_shader`. It was promoted to a core feature in OpenGL 2.0.

O.27 High-Level Fragment Programming

The name string for high-level fragment programming is `GL_ARB_fragment_shader`. It was promoted to a core feature in OpenGL 2.0.

O.28 OpenGL Shading Language

The name string for the OpenGL Shading Language is `GL_ARB_shading_language_100`. The presence of this extension string indicates that programs written in version 1 of the Shading Language are accepted by OpenGL.

It was promoted to a core feature in OpenGL 2.0.

O.29 Non-Power-Of-Two Textures

The name string for non-power-of-two textures is `GL_ARB_texture_non_power_of_two`. It was promoted to a core feature in OpenGL 2.0.

O.30 Point Sprites

The name string for point sprites is `GL_ARB_point_sprite`. It was promoted to a core feature in OpenGL 2.0.

O.31 Fragment Program Shadow

Fragment program shadow extends low-level fragment programs defined with `GL_ARB_fragment_program` to add shadow 1D, 2D, and 3D texture targets, and remove the interaction with `GL_ARB_shadow`.

The name string for fragment program shadow is `GL_ARB_fragment_program_shadow`.

O.32 Multiple Render Targets

The name string for multiple render targets is `GL_ARB_draw_buffers`. It was promoted to a core feature in OpenGL 2.0.

O.33 Rectangular Textures

Rectangular textures define a new texture target `TEXTURE_RECTANGLE_ARB` that supports 2D textures without requiring power-of-two dimensions. Rectangular textures are useful for storing video images that do not have power-of-two sizes (POTS). Resampling artifacts are avoided and less texture memory may be required. They are also useful for shadow maps and window-space texturing. These textures are accessed by dimension-dependent (aka non-normalized) texture coordinates.

Rectangular textures are a restricted version of non-power-of-two textures. The differences are that rectangular textures are supported only for 2D; they require a new texture target; and the new target uses non-normalized texture coordinates.

The name string for texture rectangles is `GL_ARB_texture_rectangle`.

O.34 Floating-Point Color Buffers

Floating-point color buffers can represent values outside the normal $[0, 1]$ range of colors in the fixed-function OpenGL pipeline. This group of related extensions enables controlling clamping of vertex colors, fragment colors throughout the pipeline, and pixel data read back to client memory, and also includes WGL and

GLX extensions for creating frame buffers with floating-point color components (referred to in GLX as *framebuffer configurations*, and in WGL as *pixel formats*).

The name strings for floating-point color buffers are `GL_ARB_color_buffer_float`, `GLX_ARB_fbconfig_float`, and `WGL_ARB_pixel_format_float`.

O.35 Half-Precision Floating Point

This extension defines the representation of a 16-bit floating point data format, and a corresponding *type* argument which may be used to specify and read back pixel and texture images stored in this format in client memory. Half-precision floats are smaller than full precision floats, but provide a larger dynamic range than similarly sized (`short`) data types.

The name string for half-precision floating point is `GL_ARB_half_float_pixel`.

O.36 Floating-Point Textures

Floating-point textures stored in both 32- and 16-bit formats may be defined using new *internalformat* arguments to commands which specify and read back texture images.

The name string for floating-point textures is `GL_ARB_texture_float`.

O.37 Pixel Buffer Objects

The buffer object interface is expanded by adding two new binding targets for buffer objects, the pixel pack and unpack buffers. This permits buffer objects to be used to store pixel data as well as vertex array data. Pixel-drawing and -reading commands using data in pixel buffer objects may operate at greatly improved performance compared to data in client memory.

The name string for pixel buffer objects is `GL_ARB_pixel_buffer_object`. It was promoted to a core feature in OpenGL 2.1.

Index

- x*_BIAS, 138, 371
- x*_SCALE, 138, 371
- 2D, 306, 307, 390
- 2.BYTES, 309
- 3D, 306, 307
- 3D_COLOR, 306, 307
- 3D_COLOR_TEXTURE, 306, 307
- 3.BYTES, 309
- 4D_COLOR_TEXTURE, 306, 307
- 4.BYTES, 309
- 1, 177, 191, 192, 214, 323, 357
- 2, 177, 191, 192, 323, 357
- 3, 177, 191, 192, 323, 357
- 4, 177, 191, 192, 323
- ACCUM, 263, 264
- Accum, 62, 113, 263, 264, 407
- ACCUM_*_BITS, 407
- ACCUM_ALPHA_BITS, 280
- ACCUM_BLUE_BITS, 280
- ACCUM_BUFFER_BIT, 260, 339, 407
- ACCUM_GREEN_BITS, 280
- ACCUM_RED_BITS, 279
- ACTIVE_ATTRIBUTE_MAX_LENGTH, 93, 334, 452
- ACTIVE_ATTRIBUTES, 93, 334
- ACTIVE_TEXTURE, 25, 55, 56, 70, 218, 298, 316
- ACTIVE_UNIFORM_MAX_LENGTH, 97, 334
- ACTIVE_UNIFORMS, 96, 334
- ActiveTexture, 55, 56, 100, 227
- ADD, 219, 221, 222, 263, 264, 425
- ADD_SIGNED, 222
- ALL_ATTRIB_BITS, 338, 339, 407
- ALPHA, 138, 152, 165, 166, 177, 181, 185, 200, 201, 219–221, 224, 250, 268, 271, 289, 322, 323, 359, 367, 371, 372, 374, 389, 406, 410, 416
- ALPHA12, 181
- ALPHA16, 181
- ALPHA4, 181
- ALPHA8, 179, 181
- ALPHA_BIAS, 163
- ALPHA_BITS, 294, 407
- ALPHA_INTEGER, 152
- ALPHA_SCALE, 163, 219, 451
- ALPHA_TEST, 242, 407
- AlphaFunc, 242, 407
- ALWAYS, 201, 225, 243–245, 362
- AMBIENT, 81, 83
- AMBIENT_AND_DIFFUSE, 81, 83
- AND, 253
- AND_INVERTED, 253
- AND_REVERSE, 253
- Antialiasing, 128
- AreTexturesResident, 217, 310, 406
- ARRAY_BUFFER, 38, 40–43, 45–48, 331, 332
- ARRAY_BUFFER_BINDING, 46
- ArrayElement, 24, 32–34, 46, 106, 308
- ATTACHED_SHADERS, 334
- AttachShader, 90, 310
- AUTO_NORMAL, 103, 299
- AUX_{*i*}, 255–257, 262, 319
- AUX0, 255
- AUX_BUFFERS, 255, 280
- BACK, 80, 82, 83, 129, 130, 132, 244, 247, 255, 256, 258–260, 268,

- 277, 317, 355, 405
- BACK_LEFT, 256, 257, 262, 319
- BACK_RIGHT, 256, 257, 262, 319
- Begin, 16, 19–24, 33, 34, 49, 62, 64, 80, 83, 86, 108, 122, 126, 129, 132, 234, 293, 294, 300, 301, 306, 404
- BeginConditionalRender, 62, 63
- BeginQuery, 61, 62, 67, 245
- BeginTransformFeedback, 63–66
- BGR, 152, 268, 271, 322
- BGR_INTEGER, 152
- BGRA, 152, 155, 159, 268, 322, 414
- BGRA_INTEGER, 152
- BindAttribLocation, 94, 310
- BindBuffer, 38, 46, 65, 310
- BindBufferBase, 64–66, 310
- BindBufferRange, 64–66, 310
- BindFragDataLocation, 235, 236, 311
- BindFramebuffer, 278, 280, 292, 310
- BindRenderbuffer, 281, 282, 310
- BindTexture, 56, 100, 216, 217, 451
- BindVertexArray, 48, 310
- BITMAP, 131, 140, 143, 150, 151, 160, 173, 271, 324
- Bitmap, 62, 113, 173, 234, 406
- BITMAP_TOKEN, 307
- BLEND, 219, 221, 246, 247, 249, 252
- BlendColor, 249, 432
- BlendEquation, 247, 432
- BlendEquationSeparate, 247, 445
- BlendFunc, 248, 432
- BlendFuncSeparate, 248, 433
- BlitFramebuffer, 273, 276, 277, 310
- BLUE, 138, 152, 268, 271, 322, 359, 367, 371, 372, 374, 389
- BLUE_BIAS, 163
- BLUE_BITS, 294, 407
- BLUE_INTEGER, 152
- BLUE_SCALE, 163
- BOOL, 97
- BOOL_VEC2, 97
- BOOL_VEC3, 97
- BOOL_VEC4, 97
- BUFFER_ACCESS, 39, 41, 42
- BUFFER_MAP_POINTER, 39, 41–43, 332
- BUFFER_MAPPED, 39, 41–43
- BUFFER_SIZE, 39, 41, 45, 65
- BUFFER_USAGE, 39, 41, 42
- BufferData, 40, 41, 310
- BufferSubData, 41, 42, 310
- bvec2, 99
- BYTE, 29, 151, 271, 272, 309
- C3F_V3F, 36, 37
- C4F_N3F_V3F, 36, 37
- C4UB_V2F, 36, 37
- C4UB_V3F, 36, 37
- CallList, 24, 308, 309, 407
- CallLists, 24, 308, 309, 407, 452
- CCW, 79, 355
- CheckFramebufferStatus, 292, 293, 310
- CLAMP, 201, 205, 206, 406
- CLAMP_FRAGMENT_COLOR, 161, 276
- CLAMP_READ_COLOR, 270
- CLAMP_TO_BORDER, 201, 206, 406, 425
- CLAMP_TO_EDGE, 201, 206, 276, 415
- CLAMP_VERTEX_COLOR, 85
- ClampColor, 85, 161, 405
- CLEAR, 253
- Clear, 62, 113, 260, 261, 263, 407
- ClearAccum, 261, 407
- ClearBuffer, 263
- ClearBufferf, 263
- ClearBufferfi, 262
- ClearBufferfv, 262
- ClearBufferi, 263
- ClearBufferiv, 262
- ClearBufferuiv, 262
- ClearColor, 261, 262
- ClearDepth, 261, 262
- ClearIndex, 261
- ClearStencil, 261, 262
- CLIENT_ACTIVE_TEXTURE, 31, 316
- CLIENT_ALL_ATTRIB_BITS, 338, 339, 407
- CLIENT_PIXEL_STORE_BIT, 339

- CLIENT_VERTEX_ARRAY_BIT, 339
- ClientActiveTexture, 24, 31, 310, 404
- CLIP_DISTANCE i , 457
- CLIP_PLANE i , 67, 68, 457
- CLIP_PLANE0, 68
- ClipPlane, 67
- COEFF, 319
- COLOR, 51, 56, 57, 141, 145, 146, 191, 273
- Color, 24, 26, 63, 73, 83, 87, 92
- Color*, 404
- Color3, 26
- Color4, 26
- Color[size][type]v, 32
- COLOR_ARRAY, 31, 36
- COLOR_ARRAY_POINTER, 328
- COLOR_ATTACHMENT i , 255, 256, 268, 285, 290
- COLOR_ATTACHMENT m , 255, 258
- COLOR_ATTACHMENT n , 279
- COLOR_ATTACHMENT0, 255, 258, 268, 279
- COLOR_BUFFER_BIT, 260, 263, 275, 276, 339
- COLOR_INDEX, 131, 140, 143, 150, 152, 164, 173, 269, 273, 323, 324, 404
- COLOR_INDEXES, 81, 84
- COLOR_LOGIC_OP, 252
- COLOR_MATERIAL, 83, 405
- COLOR_MATRIX, 324
- COLOR_MATRIX_STACK_DEPTH, 324
- COLOR_SUM, 227, 229, 407
- COLOR_TABLE, 140, 143, 165
- COLOR_TABLE_ALPHA_SIZE, 325
- COLOR_TABLE_BIAS, 140, 141, 325
- COLOR_TABLE_BLUE_SIZE, 325
- COLOR_TABLE_FORMAT, 325
- COLOR_TABLE_GREEN_SIZE, 325
- COLOR_TABLE_INTENSITY_SIZE, 325
- COLOR_TABLE_LUMINANCE_SIZE, 325
- COLOR_TABLE_RED_SIZE, 325
- COLOR_TABLE_SCALE, 140, 141, 325
- COLOR_TABLE_WIDTH, 325
- COLOR_WRITEMASK, 259
- ColorMask, 259, 260
- ColorMaski, 259
- ColorMaterial, 82, 83, 299, 396, 405, 412
- ColorPointer, 24, 29, 30, 36, 310, 404
- ColorSubTable, 136, 142
- ColorTable, 136, 140–143, 147, 148, 170, 171, 311
- ColorTableParameter, 141
- ColorTableParameterfv, 140
- Colorub, 87
- Colorui, 87
- Colorus, 87
- COMBINE, 219, 222, 227, 425, 433
- COMBINE_ALPHA, 219, 222, 223
- COMBINE_RGB, 219, 222, 223
- COMPARE_R_TO_TEXTURE, 457
- COMPARE_REF_TO_TEXTURE, 201, 224, 457
- COMPILE, 308, 396
- COMPILE_AND_EXECUTE, 308, 309
- COMPILE_STATUS, 89, 333
- CompileShader, 89, 311
- COMPRESSED_ALPHA, 185
- COMPRESSED_INTENSITY, 185
- COMPRESSED_LUMINANCE, 185
- COMPRESSED_LUMINANCE_ALPHA, 185
- COMPRESSED_RED, 185
- COMPRESSED_RED_RGTC1, 179, 185, 399–401
- COMPRESSED_RG, 185
- COMPRESSED_RG_RGTC2, 179, 185, 401
- COMPRESSED_RGB, 185
- COMPRESSED_RGBA, 185
- COMPRESSED_SIGNED_RED_RGTC1, 179, 185, 400, 401
- COMPRESSED_SIGNED_RG_RGTC2, 179, 185, 401
- COMPRESSED_SLUMINANCE, 185, 225

- COMPRESSED_SLUMINANCE_ALPHA, 185, 225
- COMPRESSED_SRGB, 185, 225
- COMPRESSED_SRGB_ALPHA, 185, 225
- COMPRESSED_TEXTURE_FORMATS, 178
- CompressedTexImage, 198
- CompressedTexImage*, 292
- CompressedTexImage1D, 196–198
- CompressedTexImage2D, 196–198
- CompressedTexImage3D, 196–198
- CompressedTexSubImage1D, 197–199
- CompressedTexSubImage2D, 198, 199
- CompressedTexSubImage3D, 198, 199
- CONSTANT, 221, 223, 361
- CONSTANT_ALPHA, 250, 432
- CONSTANT_ATTENUATION, 81
- CONSTANT_BORDER, 167, 168
- CONSTANT_COLOR, 250, 432
- CONTEXT_FLAG_FORWARD_COMPATIBLE_BIT, 329
- CONTEXT_FLAGS, 329
- CONVOLUTION_1D, 144, 146, 165, 188, 325, 326
- CONVOLUTION_2D, 143–145, 165, 188, 325, 326
- CONVOLUTION_BORDER_COLOR, 168, 326
- CONVOLUTION_BORDER_MODE, 167, 326
- CONVOLUTION_FILTER_BIAS, 143–145, 326
- CONVOLUTION_FILTER_SCALE, 143–146, 326
- CONVOLUTION_FORMAT, 326
- CONVOLUTION_HEIGHT, 326
- CONVOLUTION_WIDTH, 326
- ConvolutionFilter1D, 136, 144–146
- ConvolutionFilter2D, 136, 143–146
- ConvolutionParameter, 144, 167
- ConvolutionParameterfv, 143, 144, 168
- ConvolutionParameteriv, 145, 168
- COORD_REPLACE, 118, 121
- COPY, 253, 363
- COPY_INVERTED, 253
- COPY_PIXEL_TOKEN, 307
- CopyColorSubTable, 142, 273
- CopyColorTable, 141, 142, 273
- CopyConvolutionFilter*, 273
- CopyConvolutionFilter1D, 145, 146
- CopyConvolutionFilter2D, 145, 146
- CopyPixels, 62, 113, 135, 137, 141, 145, 146, 165, 191, 264, 273–276, 305
- CopyTexImage, 294
- CopyTexImage*, 273, 285, 292
- CopyTexImage1D, 165, 191, 193, 195, 209
- CopyTexImage2D, 165, 189, 191, 193, 195, 209
- CopyTexImage3D, 193
- CopyTexSubImage, 294
- CopyTexSubImage*, 195, 199, 273, 285
- CopyTexSubImage1D, 165, 192–195
- CopyTexSubImage2D, 165, 192–195
- CopyTexSubImage3D, 165, 192, 193, 195
- CreateProgram, 90, 310
- CreateShader, 88, 310, 444
- CreateShaderObjectARB, 444
- CULL_FACE, 129
- CullFace, 129, 130, 134
- CURRENT_BIT, 339
- CURRENT_FOG_COORD, 439
- CURRENT_FOG_COORDINATE, 439
- CURRENT_QUERY, 330
- CURRENT_RASTER_SECONDARY_COLOR, 450
- CURRENT_RASTER_TEXTURE_COORDS, 70, 395
- CURRENT_TEXTURE_COORDS, 25
- CURRENT_VERTEX_ATTRIB, 336
- CW, 79
- DECAL, 219, 220
- DECR, 244
- DECR_WRAP, 244, 433
- DELETE_STATUS, 89, 333
- DeleteBuffers, 39, 310

- DeleteFramebuffers, 278, 280, 310
- DeleteLists, 310, 407
- DeleteProgram, 91, 310
- DeleteQueries, 62, 310
- DeleteRenderbuffers, 282, 292, 310
- DeleteShader, 89, 310
- DeleteTextures, 216, 292, 310
- DeleteVertexArrays, 47, 48, 310
- DEPTH, 191, 273, 359, 367, 371, 432
- DEPTH24_STENCIL8, 180, 184
- DEPTH32F_STENCIL8, 180, 184
- DEPTH_ATTACHMENT, 262, 279, 285, 290
- DEPTH_BIAS, 138, 163
- DEPTH_BITS, 275, 294, 407
- DEPTH_BUFFER, 262, 319
- DEPTH_BUFFER_BIT, 260, 263, 275, 276, 339
- DEPTH_COMPONENT, 106, 140, 143, 150, 152, 177, 178, 184, 224, 226, 233, 267, 269, 273, 275, 290, 322, 325–327, 447
- DEPTH_COMPONENT16, 179, 184
- DEPTH_COMPONENT24, 179, 184
- DEPTH_COMPONENT32, 184
- DEPTH_COMPONENT32F, 179, 184
- DEPTH_SCALE, 138, 163
- DEPTH_STENCIL, 106, 140, 143, 150, 152, 155, 159, 160, 162, 177, 178, 184, 191, 212, 224, 226, 233, 263, 265, 267, 269, 273, 275, 284, 287, 290, 322
- DEPTH_STENCIL_ATTACHMENT, 262, 284, 285, 287, 320
- DEPTH_TEST, 245
- DEPTH_TEXTURE_MODE, 201, 214, 221, 224, 406
- DepthFunc, 245
- DepthMask, 259, 260, 265
- DepthRange, 51, 71, 315, 396
- DepthTest, 265
- DetachShader, 90, 310
- dFdx, 312
- dFdy, 312
- DIFFUSE, 81, 83
- Disable, 55–57, 60, 68, 75, 79, 80, 83, 113, 115, 117, 123, 125, 129, 131, 134, 170, 171, 226, 227, 229, 241–243, 245, 247, 252, 298, 299, 404, 405, 407
- DisableClientState, 24, 31, 36, 38, 310, 404
- Disablei, 246
- DisableVertexAttribArray, 31, 310, 336
- DITHER, 252
- DOMAIN, 319
- DONT_CARE, 312, 313, 383
- DOT3_RGB, 222
- DOT3_RGBA, 222
- DOUBLE, 29, 32
- DRAW_BUFFER, 255, 258, 268, 452
- DRAW_BUFFER*i*, 246, 247, 258, 259, 291, 452
- DRAW_BUFFER0, 258
- DRAW_FRAMEBUFFER, 264, 278–280, 284, 285, 293, 319, 365
- DRAW_FRAMEBUFFER_BINDING, 23, 150, 173, 208, 255, 257, 273, 276, 280, 293, 294
- DRAW_PIXEL_TOKEN, 307
- DrawArrays, 33, 34, 46, 48, 106, 107, 308
- DrawBuffer, 253–256, 258, 260, 263
- DrawBuffers, 254, 255, 257, 258
- DrawElements, 34, 35, 46, 48, 64, 106, 107, 308, 416
- DrawPixels, 62, 66, 113, 131, 135–137, 140, 143, 148–154, 159, 160, 162, 165, 172, 173, 176, 234, 264, 265, 271, 273, 305, 406, 450
- DrawRangeElements, 35, 46, 106, 107, 308, 386
- DST_ALPHA, 250
- DST_COLOR, 250, 432
- DYNAMIC_COPY, 39, 41
- DYNAMIC_DRAW, 39, 40
- DYNAMIC_READ, 39, 41
- EDGE_FLAG_ARRAY, 31, 36

- EDGE_FLAG_ARRAY_POINTER, 328
- EdgeFlag, 23, 24
- EdgeFlag*, 404
- EdgeFlagPointer, 24, 29, 30, 310, 404
- EdgeFlagv, 23, 32
- ELEMENT_ARRAY_BUFFER, 38, 40, 42, 43, 45–47, 331, 332
- EMISSION, 81, 83
- Enable, 55–57, 60, 68, 75, 79, 80, 83, 113, 115, 117, 123, 125, 129, 131, 134, 170, 171, 226, 227, 229, 241–243, 245, 247, 252, 298, 299, 315, 404, 405, 407
- ENABLE_BIT, 339
- EnableClientState, 24, 31, 36, 38, 310, 404
- Enablei, 246
- EnableVertexAttribArray, 31, 48, 310, 336
- End, 16, 19–24, 33, 34, 49, 62, 80, 83, 86, 122, 129, 132, 293, 300, 301, 306, 404
- EndConditionalRender, 62, 63
- EndList, 308, 407
- EndQuery, 61, 62, 245, 246
- EndTransformFeedback, 64
- EQUAL, 201, 225, 243–245
- EQUIV, 253
- EVAL_BIT, 339
- EvalCoord, 24, 298, 299
- EvalCoord*, 407
- EvalCoord1, 299–301
- EvalCoord1d, 300
- EvalCoord1f, 300
- EvalCoord2, 299, 301, 302
- EvalMesh*, 407
- EvalMesh1, 62, 300
- EvalMesh2, 62, 300, 301
- EvalPoint, 24
- EvalPoint*, 407
- EvalPoint1, 301
- EvalPoint2, 301
- EXP, 229, 230, 351
- EXP2, 229
- EXT_pixel_buffer_object, 452
- EXT_texture_sRGB, 453
- EXTENSIONS, 137, 328, 329, 407, 460, 461
- EYE_LINEAR, 59, 60, 317, 360
- EYE_PLANE, 59
- FALSE, 23, 39, 41, 43, 76, 78, 85, 89–91, 98, 99, 108–110, 118, 136, 138, 147, 148, 160, 163, 171, 172, 201, 214, 217, 234, 242, 265, 267, 315, 324, 326, 327, 330–333, 336, 337, 344–346, 349–360, 362, 363, 370–378, 380, 381, 390
- FASTEST, 312, 313
- FEEDBACK, 303–305, 397
- FEEDBACK_BUFFER_POINTER, 328
- FeedbackBuffer, 304, 305, 310, 328, 407
- FILL, 132, 134, 135, 300, 355, 396, 410
- Finish, 311, 312, 395
- FIXED_ONLY, 85, 161, 270, 276, 277, 351
- FLAT, 85, 396
- flat, 85
- FLOAT, 29, 32, 36–38, 93, 97, 102, 150, 151, 178, 269, 270, 272, 309, 318–320, 323, 344–346
- float, 92
- FLOAT_32_UNSIGNED_INT_24_8_REV, 150, 151, 154, 155, 267, 271, 272
- FLOAT_MAT2, 93, 97, 102
- FLOAT_MAT2x3, 93, 97, 451
- FLOAT_MAT2x4, 93, 97, 451
- FLOAT_MAT3, 93, 97, 102
- FLOAT_MAT3x2, 93, 97, 451
- FLOAT_MAT3x4, 93, 97, 451
- FLOAT_MAT4, 93, 97, 102
- FLOAT_MAT4x2, 93, 97, 451
- FLOAT_MAT4x3, 93, 97, 451
- FLOAT_VEC2, 93, 97, 102
- FLOAT_VEC3, 93, 97, 102
- FLOAT_VEC4, 93, 97, 102
- Flush, 311, 395
- FlushMappedBufferRange, 44, 45

- FOG, 229, 407
- Fog, 229, 230, 407
- FOG_BIT, 339
- FOG_COLOR, 230
- FOG_COORD, 70, 229, 439, 446
- FOG_COORD_ARRAY, 31, 36, 439
- FOG_COORD_ARRAY_BUFFER_BINDING, 439, 446
- FOG_COORD_ARRAY_POINTER, 328, 439
- FOG_COORD_ARRAY_STRIDE, 439
- FOG_COORD_ARRAY_TYPE, 439
- FOG_COORD_SRC, 71, 229, 230, 439, 446
- FOG_COORDINATE, 439, 446
- FOG_COORDINATE_ARRAY, 439
- FOG_COORDINATE_ARRAY_BUFFER_BINDING, 439
- FOG_COORDINATE_ARRAY_POINTER, 439
- FOG_COORDINATE_ARRAY_STRIDE, 439
- FOG_COORDINATE_ARRAY_TYPE, 439
- FOG_COORDINATE_SOURCE, 432, 439
- FOG_DENSITY, 229
- FOG_END, 229
- FOG_HINT, 312, 407
- FOG_INDEX, 230
- FOG_MODE, 229, 230
- FOG_START, 229
- FogCoord, 24, 25, 432
- FogCoord*, 404
- FogCoord[type]v, 32
- FogCoordPointer, 24, 29, 30, 310, 404
- FRAGMENT_DEPTH, 229, 230, 351
- FRAGMENT_SHADER, 231, 333
- FRAGMENT_SHADER_DERIVATIVE_HINT, 312
- FRAMEBUFFER, 278, 284, 285, 293, 319
- FRAMEBUFFER_ATTACHMENT_ALPHA_SIZE, 320
- FRAMEBUFFER_ATTACHMENT_BLUE_SIZE, 320
- FRAMEBUFFER_ATTACHMENT_COLOR_ENCODING, 247, 248, 251, 320
- FRAMEBUFFER_ATTACHMENT_COMPONENT_TYPE, 320
- FRAMEBUFFER_ATTACHMENT_DEPTH_SIZE, 320
- FRAMEBUFFER_ATTACHMENT_GREEN_SIZE, 320
- FRAMEBUFFER_ATTACHMENT_OBJECT_NAME, 284, 286, 290, 319–321
- FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE, 284, 286, 290, 291, 294, 319–321
- FRAMEBUFFER_ATTACHMENT_RED_SIZE, 320
- FRAMEBUFFER_ATTACHMENT_STENCIL_SIZE, 320
- FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE, 287, 320, 321
- FRAMEBUFFER_ATTACHMENT_TEXTURE_LAYER, 287, 290, 295, 320, 321
- FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL, 208, 286, 288, 289, 319, 321
- FRAMEBUFFER_BINDING, 280
- FRAMEBUFFER_COMPLETE, 293
- FRAMEBUFFER_DEFAULT, 320
- FRAMEBUFFER_INCOMPLETE_ATTACHMENT, 291
- FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER, 291
- FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT, 291
- FRAMEBUFFER_INCOMPLETE_MULTISAMPLE, 292
- FRAMEBUFFER_INCOMPLETE_READ_BUFFER, 291
- FRAMEBUFFER_SRGB, 247, 248, 251
- FRAMEBUFFER_UNDEFINED, 291
- FRAMEBUFFER_UNSUPPORTED, 291, 292
- FramebufferRenderbuffer, 284, 292, 310
- FramebufferTexture, 287
- FramebufferTexture*, 286, 287, 292

- FramebufferTexture1D, 285, 286, 310
- FramebufferTexture2D, 285, 286, 310
- FramebufferTexture3D, 285–287, 310
- FramebufferTextureLayer, 287, 310
- FRONT, 80, 83, 129, 130, 132, 244, 247, 255, 256, 258–260, 268, 277, 317, 405
- FRONT_AND_BACK, 80, 82, 83, 129, 132, 244, 247, 256, 258–260, 268
- FRONT_LEFT, 256, 257, 262, 319
- FRONT_RIGHT, 256, 257, 262, 319
- FrontFace, 79, 129, 234, 405
- Frustum, 53, 54, 396, 404
- ftransform, 107
- FUNC_ADD, 247, 249, 363
- FUNC_REVERSE_SUBTRACT, 247, 249, 451
- FUNC_SUBTRACT, 247, 249
- fwidth, 312
- Gen*, 402, 404
- GenBuffers, 39, 310
- GENERATE_MIPMAP, 200, 201, 210, 214, 406, 431
- GENERATE_MIPMAP_HINT, 312, 407
- GenerateMipmap, 211, 310
- GenFramebuffers, 278, 280, 310
- GenLists, 309, 310, 407
- GenQueries, 62, 310
- GenRenderbuffers, 281, 282, 310
- GenTextures, 217, 310, 324
- GenVertexArrays, 47, 48, 310, 332
- GEQUAL, 201, 225, 243–245, 438
- Get, 25, 51, 70, 311, 314, 315
- GetActiveAttrib, 92, 93, 451
- GetActiveUniform, 96, 97, 99, 451
- GetAttachedShaders, 334
- GetAttribLocation, 93, 94
- GetBooleani_v, 259, 314
- GetBooleanv, 242, 259, 314–316, 340
- GetBufferParameter, 317
- GetBufferParameteriv, 317
- GetBufferPointerv, 331, 332
- GetBufferSubData, 331
- GetClipPlane, 316, 317
- GetColorTable, 143, 267, 325
- GetColorTableParameter, 325
- GetCompressedTexImage, 197–199, 313, 319, 323, 324
- GetConvolutionFilter, 267, 325
- GetConvolutionParameter, 326
- GetConvolutionParameteriv, 144
- GetDoublev, 314–316, 340
- GetError, 15
- GetFloatv, 242, 314–316, 324, 340
- GetFragDataLocation, 236
- GetFramebufferAttachmentParameteriv, 294, 317, 319, 320
- GetHistogram, 148, 267, 326, 446
- GetHistogramParameter, 327
- GetIntegeri_v, 314, 332
- GetIntegerv, 35, 115, 257, 258, 280, 282, 314–317, 324, 329, 340
- GetLight, 316, 317
- GetLightiv, 317
- GetMap, 317, 319
- GetMaterial, 317
- GetMaterialiv, 317
- GetMinmax, 267, 327
- GetMinmaxParameter, 328
- GetPixelMap, 317, 319
- GetPixelMapfv, 319
- GetPixelMapuiv, 319
- GetPixelMapusv, 319
- GetPointerv, 328
- GetPolygonStipple, 267, 324
- GetProgramInfoLog, 91, 334
- GetProgramiv, 90, 93, 96, 97, 102, 108, 333–335
- GetQueryiv, 330
- GetQueryObject[u]iv, 331
- GetQueryObjectiv, 330
- GetQueryObjectuiv, 330
- GetRenderbufferParameteriv, 294, 317, 321
- GetSeparableFilter, 267, 326
- GetShaderInfoLog, 89, 334
- GetShaderiv, 89, 333, 335
- GetShaderSource, 335

- GetString, 328, 329, 407
- GetStringi, 329
- GetTexEnv, 317, 452
- GetTexEnviv, 317
- GetTexGen, 317
- GetTexGeniv, 317
- GetTexImage, 215, 267, 322, 323, 325–327
- GetTexLevelParameter, 317, 318
- GetTexParameter, 294, 317, 318
- GetTexParameterfv, 215, 217
- GetTexParameterI, 317
- GetTexParameterIiv, 318
- GetTexParameterIuiv, 318
- GetTexParameteriv, 215, 217
- GetTexparameteriv, 317
- GetTransformFeedbackVarying, 102, 103
- GetUniform*, 337
- GetUniformfv, 336
- GetUniformiv, 336
- GetUniformLocation, 96, 97, 100
- GetUniformuiv, 337
- GetVertexAttribdv, 335, 336
- GetVertexAttribfv, 335, 336
- GetVertexAttribIiv, 335, 336
- GetVertexAttribIuiv, 335, 336
- GetVertexAttribiv, 335, 336
- GetVertexAttribPointerv, 336
- gl., 102, 235, 236
- GL_ARB_color_buffer_float, 467
- GL_ARB_depth_texture, 432, 463
- GL_ARB_draw_buffers, 444, 466
- GL_ARB_fragment_program, 438, 464, 466
- GL_ARB_fragment_program_shadow, 466
- GL_ARB_fragment_shader, 437, 438, 443, 465
- GL_ARB_half_float_pixel, 467
- GL_ARB_matrix_palette, 463
- GL_ARB_multisample, 424, 461
- GL_ARB_multitexture, 425, 461
- GL_ARB_occlusion_query, 438, 465
- GL_ARB_pixel_buffer_object, 450, 467
- GL_ARB_point_parameters, 433, 462
- GL_ARB_point_sprite, 445, 466
- GL_ARB_shader_objects, 437, 439, 443, 465
- GL_ARB_shading_language_100, 444, 465
- GL_ARB_shadow, 432, 463, 466
- GL_ARB_shadow_ambient, 464
- GL_ARB_texture_border_clamp, 426, 462
- GL_ARB_texture_compression, 423, 462
- GL_ARB_texture_cube_map, 424, 462
- GL_ARB_texture_env_add, 425, 462
- GL_ARB_texture_env_combine, 425, 463
- GL_ARB_texture_env_crossbar, 434, 463
- GL_ARB_texture_env_dot3, 425, 463
- GL_ARB_texture_float, 467
- GL_ARB_texture_mirrored_repeat, 434, 463
- GL_ARB_texture_non_power_of_two, 444, 465
- GL_ARB_texture_rectangle, 466
- GL_ARB_transpose_matrix, 426, 461
- GL_ARB_vertex_blend, 462
- GL_ARB_vertex_buffer_object, 437, 438, 464
- GL_ARB_vertex_program, 431, 434, 464
- GL_ARB_vertex_shader, 437, 439, 443, 465
- GL_ARB_window_pos, 434, 464
- GL_ATI_separate_stencil, 445
- gl_BackColor, 78, 85, 107
- gl_BackSecondaryColor, 78, 85, 107
- gl_ClipDistance[], 68
- gl_ClipVertex, 68, 107
- gl_Color, 233
- GL_EXT_bgra, 414
- GL_EXT_blend_color, 418
- GL_EXT_blend_equation_separate, 445
- GL_EXT_blend_func_separate, 433
- GL_EXT_blend_logic_op, 410, 445
- GL_EXT_blend_minmax, 418
- GL_EXT_blend_subtract, 418
- GL_EXT_color_subtable, 417

- GL_EXT_color_table, 417
- GL_EXT_convolution, 417
- GL_EXT_copy_texture, 411
- GL_EXT_draw_range_elements, 416
- GL_EXT_fog_coord, 432
- GL_EXT_histogram, 418
- GL_EXT_multi_draw_arrays, 432
- GL_EXT_packed_pixels, 415
- GL_EXT_polygon_offset, 410
- GL_EXT_rescale_normal, 415
- GL_EXT_secondary_color, 433
- GL_EXT_separate_specular_color, 415
- GL_EXT_shadow_funcs, 438
- GL_EXT_stencil_two_side, 445
- GL_EXT_stencil_wrap, 433
- GL_EXT_subtexture, 411
- GL_EXT_texture, 410, 411
- GL_EXT_texture3D, 414
- GL_EXT_texture_lod_bias, 434
- GL_EXT_texture_object, 411
- GL_EXT_texture_sRGB, 450
- GL_EXT_vertex_array, 409
- gl_FogFragCoord, 70, 107
- gl_FragColor, 234, 235, 257, 258
- gl_FragCoord, 233
- gl_FragCoord.z, 394
- gl_FragData, 235, 257, 258
- gl_FragData[n], 234, 235
- gl_FragDepth, 234, 235, 394
- gl_FrontColor, 78, 85, 107
- gl_FrontFacing, 234
- gl_FrontSecondaryColor, 78, 85, 107
- GL_HP_convolution_border_modes, 417
- GL_NV_blend_square, 432
- GL_PERSPECTIVE_CORRECT_HINT, 450
- gl_PointSize, 107, 117
- gl_Position, 101, 107
- gl_PrimitiveID, 234
- gl_SecondaryColor, 233
- GL_SGI_color_matrix, 417
- GL_SGIS_generate_mipmap, 431
- GL_SGIS_multitexture, 422
- GL_SGIS_texture_edge_clamp, 416
- GL_SGIS_texture_lod, 416
- gl_TexCoord, 107
- gl_VertexID, 106, 234
- gl_[]hyperpage, 101
- GLX_ARB_fbconfig_float, 467
- GREATER, 201, 225, 243–245
- GREEN, 138, 152, 268, 271, 322, 359, 367, 371, 372, 374, 389
- GREEN_BIAS, 163
- GREEN_BITS, 294, 407
- GREEN_INTEGER, 152
- GREEN_SCALE, 163
- HALF_FLOAT, 29, 151, 269, 270, 272
- Hint, 312, 407
- HINT_BIT, 339
- HISTOGRAM, 147, 171, 326, 327
- Histogram, 146, 147, 171, 311
- HISTOGRAM_ALPHA_SIZE, 327
- HISTOGRAM_BLUE_SIZE, 327
- HISTOGRAM_FORMAT, 327
- HISTOGRAM_GREEN_SIZE, 327
- HISTOGRAM_LUMINANCE_SIZE, 327
- HISTOGRAM_RED_SIZE, 327
- HISTOGRAM_SINK, 327
- HISTOGRAM_WIDTH, 327
- INCR, 244
- INCR_WRAP, 244, 433
- INDEX, 320, 389
- Index, 24, 26
- Index*, 404
- Index[type]v, 32
- INDEX_ARRAY, 31, 36
- INDEX_ARRAY_POINTER, 328
- INDEX_LOGIC_OP, 252
- INDEX_OFFSET, 138, 163, 371
- INDEX_SHIFT, 138, 163, 371
- IndexMask, 259, 260
- IndexPointer, 24, 29, 30, 310, 404
- INFO_LOG_LENGTH, 333, 335
- InitNames, 302, 407
- INT, 29, 93, 97, 102, 151, 271, 272, 309, 318, 320
- INT_SAMPLER_1D, 97

- INT_SAMPLER_1D_ARRAY, 97
- INT_SAMPLER_2D, 97
- INT_SAMPLER_2D_ARRAY, 97
- INT_SAMPLER_3D, 97
- INT_SAMPLER_CUBE, 97
- INT_VEC2, 93, 97, 102
- INT_VEC3, 93, 97, 102
- INT_VEC4, 93, 97, 102
- INTENSITY, 148, 165, 166, 177, 183, 185, 200, 201, 220, 221, 224, 323, 359, 372, 406, 410
- INTENSITY12, 183
- INTENSITY16, 183
- INTENSITY4, 183
- INTENSITY8, 183
- INTERLEAVED_ATTRIBS, 65, 101, 102, 334
- INTERLEAVED_ATTRIBS, 379
- InterleavedArrays, 24, 36, 37, 310
- INTERPOLATE, 222
- INVALID_ENUM, 16, 17, 31, 56, 60, 80, 137, 143, 148, 150, 191, 196–199, 215, 255, 257, 263, 267, 268, 321, 323, 325–327
- INVALID_FRAMEBUFFER_OPERATION, 17, 23, 142, 146, 150, 173, 195, 268, 273, 276, 293, 294
- INVALID_OPERATION, 17, 24, 39, 42, 43, 45, 46, 48, 55, 56, 61–64, 66, 88, 90, 91, 94, 96, 99, 100, 108, 137, 139, 150, 154, 178, 187, 191, 195–199, 211, 216, 235, 236, 255, 257, 258, 263–265, 267–269, 271, 273, 275–278, 282, 284–287, 293, 298, 303, 305, 308, 316, 319–324, 330, 332, 336, 337, 404, 405
- INVALID_VALUE, 16, 17, 27, 29, 31, 33–35, 41, 45, 51, 54, 55, 63, 65, 80, 88, 93, 94, 96, 101, 102, 116, 117, 123, 136, 138, 139, 141, 142, 144, 147, 177, 186–188, 191–194, 196, 198, 210, 217, 229, 235, 241, 247, 255, 257, 259, 260, 282, 286, 287, 297, 298, 300, 308, 315, 318, 323, 324, 329, 332, 336, 405, 406, 451
- INVERT, 244, 253
- Is, 311
- IsBuffer, 331
- IsEnabled, 241, 249, 315, 340
- IsEnabledi, 249, 315
- IsFramebuffer, 337
- IsList, 310, 407
- IsProgram, 333
- IsQuery, 329, 330
- IsRenderbuffer, 337
- IsShader, 333
- IsTexture, 324
- IsVertexArray, 332
- KEEP, 244, 362
- LEFT, 247, 255, 256, 258, 259, 268
- LEQUAL, 201, 214, 225, 243–245, 358, 438
- LESS, 201, 225, 243–245, 363
- Light, 79–81
- LIGHTi, 80, 397, 405
- Light*, 404
- LIGHT0, 80
- LIGHT_MODEL_AMBIENT, 81
- LIGHT_MODEL_COLOR_CONTROL, 81
- LIGHT_MODEL_LOCAL_VIEWER, 81
- LIGHT_MODEL_TWO_SIDE, 81
- LIGHTING, 75, 405
- LIGHTING_BIT, 339
- LightModel, 79, 81
- LightModel*, 405
- LINE, 132, 134, 300, 301, 355, 410
- LINE_BIT, 339
- LINE_LOOP, 20, 64
- LINE_RESET_TOKEN, 307
- LINE_SMOOTH, 123, 128
- LINE_SMOOTH_HINT, 312
- LINE_STIPPLE, 125, 405
- LINE_STRIP, 20, 64, 300

- LINE_TOKEN, 307
- LINEAR, 104, 201, 206–208, 210, 212, 214, 229, 275, 276, 288, 320
- LINEAR_ATTENUATION, 81
- LINEAR_MIPMAP_LINEAR, 201, 208–210, 289
- LINEAR_MIPMAP_NEAREST, 201, 208–210, 289
- LINES, 20, 64, 126
- LineStipple, 125, 405
- LineWidth, 123, 405
- LINK_STATUS, 90, 333
- LinkProgram, 66, 90, 91, 93, 94, 96, 100–102, 236, 311
- LIST_BIT, 339
- ListBase, 309, 311, 407
- LOAD, 263, 264
- LoadIdentity, 53, 404
- LoadMatrix, 52, 53, 404
- LoadMatrix[fd], 52
- LoadName, 302, 303, 407
- LoadTransposeMatrix, 52, 404
- LoadTransposeMatrix[fd], 52
- LOGIC_OP, 252, 445, 447
- LogicOp, 252, 253
- LOWER_LEFT, 118, 121
- LUMINANCE, 152, 160, 165, 166, 177, 183, 185, 200–202, 214, 220, 221, 224, 268, 270, 271, 322, 323, 358, 359, 372, 374, 399, 406, 410
- LUMINANCE12, 183
- LUMINANCE12_ALPHA12, 183
- LUMINANCE12_ALPHA4, 183
- LUMINANCE16, 183
- LUMINANCE16_ALPHA16, 183
- LUMINANCE4, 183
- LUMINANCE4_ALPHA4, 183
- LUMINANCE6_ALPHA2, 183
- LUMINANCE8, 183
- LUMINANCE8_ALPHA8, 183
- LUMINANCE_ALPHA, 152, 160, 165, 166, 177, 183, 185, 220, 221, 268, 270, 271, 322, 323, 406
- MAJOR_VERSION, 329
- Map*, 407
- Map1, 296–298, 315
- MAP1_COLOR_4, 297
- MAP1_INDEX, 297
- MAP1_NORMAL, 297
- MAP1_TEXTURE_COORD_1, 297, 299
- MAP1_TEXTURE_COORD_2, 297, 299
- MAP1_TEXTURE_COORD_3, 297
- MAP1_TEXTURE_COORD_4, 297
- MAP1_VERTEX_3, 297
- MAP1_VERTEX_4, 297
- Map2, 297, 298, 315
- MAP2_VERTEX_3, 299
- MAP2_VERTEX_4, 299
- MAP_COLOR, 138, 163, 164
- MAP_FLUSH_EXPLICIT_BIT, 44, 45
- MAP_INVALIDATE_BUFFER_BIT, 44, 45
- MAP_INVALIDATE_RANGE_BIT, 44, 45
- MAP_READ_BIT, 43–45
- MAP_STENCIL, 138, 164
- MAP_UNSYNCHRONIZED_BIT, 44, 45
- MAP_VERTEX_3, 299
- MAP_VERTEX_4, 299
- MAP_WRITE_BIT, 44, 45
- Map{12}, 298
- MapBuffer, 41, 42, 65, 66, 310
- MapBufferRange, 43, 44
- MapGrid*, 407
- MapGrid1, 300
- MapGrid2, 300
- mat2, 92
- mat2x3, 92
- mat2x4, 92
- mat3, 92
- mat3x2, 92
- mat3x4, 92
- mat4, 92
- mat4x2, 92
- mat4x3, 92
- Material, 24, 79–81, 84, 396
- Material*, 404

- MATRIX_MODE, 55
- MatrixMode, 51, 404
- MAX, 247, 249
- MAX_3D_TEXTURE_SIZE, 187, 286
- MAX_ARRAY_TEXTURE_LAYERS, 187
- MAX_ATTRIB_STACK_DEPTH, 338, 407
- MAX_CLIENT_ATTRIB_STACK_DEPTH, 338, 407
- MAX_CLIP_DISTANCES, 457
- MAX_CLIP_PLANES, 457
- MAX_COLOR_ATTACHMENTS, 255, 256, 258, 278, 285, 293
- MAX_COLOR_MATRIX_STACK_DEPTH, 324
- MAX_COMBINED_TEXTURE_IMAGE_UNITS, 56, 105, 316
- MAX_CONVOLUTION_HEIGHT, 144, 326
- MAX_CONVOLUTION_WIDTH, 144, 326
- MAX_CUBE_MAP_TEXTURE_SIZE, 187, 286
- MAX_DRAW_BUFFERS, 235, 236, 247, 249, 257, 259
- MAX_ELEMENTS_INDICES, 35
- MAX_ELEMENTS_VERTICES, 35
- MAX_EVAL_ORDER, 297, 298
- MAX_FRAGMENT_UNIFORM_COMPONENTS, 231
- MAX_PIXEL_MAP_TABLE, 139, 163
- MAX_PROGRAM_TEXEL_OFFSET, 204
- MAX_RENDERBUFFER_SIZE, 282
- MAX_SAMPLES, 282, 283
- MAX_TEXTURE_COORDS, 25, 28, 38, 55, 56, 316, 446
- MAX_TEXTURE_IMAGE_UNITS, 105, 233, 446
- MAX_TEXTURE_LOD_BIAS, 203
- MAX_TEXTURE_SIZE, 187, 286
- MAX_TEXTURE_UNITS, 17, 56, 227, 338, 446
- MAX_TRANSFORM_FEEDBACK_
- INTERLEAVED_
- COMPONENTS, 102
- MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS, 65, 66, 101, 332
- MAX_TRANSFORM_FEEDBACK_SEPARATE_COMPONENTS, 102
- MAX_VARYING_COMPONENTS, 100, 101, 457
- MAX_VARYING_FLOATS, 457
- MAX_VERTEX_ATTRIBS, 26–29, 31, 38, 92, 94, 336
- MAX_VERTEX_TEXTURE_IMAGE_UNITS, 105
- MAX_VERTEX_UNIFORM_COMPONENTS, 95
- MAX_VIEWPORT_DIMS, 330
- MIN, 247, 249
- MIN_PROGRAM_TEXEL_OFFSET, 204
- MINMAX, 148, 171, 327, 328
- Minmax, 148, 172
- MINMAX_FORMAT, 328
- MINMAX_SINK, 328
- MINOR_VERSION, 329
- MIRRORED_REPEAT, 201, 206, 434
- MODELVIEW, 51, 56, 57
- MODELVIEW_MATRIX, 316
- MODULATE, 219–222, 360
- MULT, 263, 264
- MultiDrawArrays, 34, 46, 106, 107, 432
- MultiDrawElements, 35, 46, 47, 106, 107, 432
- MULTISAMPLE, 116, 122, 128, 134, 172, 173, 241, 253, 254
- MULTISAMPLE_BIT, 339
- MultiTexCoord, 24, 25, 31
- MultiTexCoord[size][type]v, 32
- MultMatrix, 52, 53, 404
- MultMatrix[fd], 53
- MultTransposeMatrix, 52, 404
- MultTransposeMatrix[fd], 53
- N3F_V3F, 36, 37
- NAND, 253

- NEAREST, 104, 201, 205, 206, 208, 210, 212, 224, 275, 288
- NEAREST_MIPMAP_LINEAR, 201, 208–210, 212, 214, 288
- NEAREST_MIPMAP_NEAREST, 201, 208–210, 212, 224, 288
- NEVER, 201, 225, 243–245
- NewList, 308, 309, 407
- NICEST, 312, 313
- NO_ERROR, 15
- NONE, 106, 201, 214, 215, 224, 233, 253, 255–258, 263, 268, 290, 291, 318, 320, 358, 359, 367
- NOOP, 253
- noperspective, 87
- NOR, 253
- Normal, 24, 25, 92
- Normal3, 12, 25
- Normal3*, 404
- Normal3[type]v, 32
- Normal3d, 12
- Normal3dv, 13
- Normal3f, 12
- Normal3fv, 13
- NORMAL_ARRAY, 31, 36, 38
- NORMAL_ARRAY_BUFFER_BINDING, 46
- NORMAL_ARRAY_POINTER, 328
- NORMAL_MAP, 59, 60, 424
- NORMALIZE, 57, 404
- NormalPointer, 24, 28–30, 36, 46, 310, 404
- NOTEQUAL, 201, 225, 243–245
- NULL, 38, 39, 41–44, 47, 89, 93, 97, 102, 332, 334, 335, 341, 346, 349
- NUM_COMPRESSED_TEXTURE_FORMATS, 178
- NUM_EXTENSIONS, 329
- OBJECT_LINEAR, 59, 60, 317
- OBJECT_PLANE, 59
- ONE, 249, 250, 363
- ONE_MINUS_CONSTANT_ALPHA, 250, 432
- ONE_MINUS_CONSTANT_COLOR, 250, 432
- ONE_MINUS_DST_ALPHA, 250
- ONE_MINUS_DST_COLOR, 250, 432
- ONE_MINUS_SRC_ALPHA, 223, 250
- ONE_MINUS_SRC_COLOR, 223, 250, 431
- OPERAND_n_ALPHA, 219, 223, 227, 451
- OPERAND_n_RGB, 219, 223, 227, 451
- OR, 253
- OR_INVERTED, 253
- OR_REVERSE, 253
- ORDER, 319
- Ortho, 53, 54, 396, 404
- OUT_OF_MEMORY, 16, 17, 41, 42, 282, 308
- PACK_ALIGNMENT, 267, 370
- PACK_IMAGE_HEIGHT, 267, 323, 370
- PACK_LSB_FIRST, 267, 370
- PACK_ROW_LENGTH, 267, 370
- PACK_SKIP_IMAGES, 267, 323, 370
- PACK_SKIP_PIXELS, 267, 370
- PACK_SKIP_ROWS, 267, 370
- PACK_SWAP_BYTES, 267, 370
- PASS_THROUGH_TOKEN, 307
- PassThrough, 306, 407
- PERSPECTIVE_CORRECTION_HINT, 312, 407
- PIXEL_MAP_A_TO_A, 139, 163
- PIXEL_MAP_B_TO_B, 139, 163
- PIXEL_MAP_G_TO_G, 139, 163
- PIXEL_MAP_I_TO_A, 139, 164
- PIXEL_MAP_I_TO_B, 139, 164
- PIXEL_MAP_I_TO_G, 139, 164
- PIXEL_MAP_I_TO_I, 139, 164
- PIXEL_MAP_I_TO_R, 139, 164
- PIXEL_MAP_R_TO_R, 139, 163
- PIXEL_MAP_S_TO_S, 139, 164
- PIXEL_MODE_BIT, 339
- PIXEL_PACK_BUFFER, 38, 40, 42, 43, 45, 136, 265, 331, 332, 450
- PIXEL_PACK_BUFFER_BINDING, 271, 319, 322

- PIXEL_UNPACK_BUFFER, 38, 40, 42, 43, 45, 136, 331, 332, 450
- PIXEL_UNPACK_BUFFER_BINDING, 139, 150, 196
- PixelMap, 135, 138, 139, 277, 452
- PixelStore, 24, 135, 136, 138, 267, 277, 310
- PixelTransfer, 135, 138, 169, 277
- PixelZoom, 162, 172, 406
- POINT, 132, 134, 300, 301, 355, 410
- POINT_BIT, 339
- POINT_DISTANCE_ATTENUATION, 117
- POINT_FADE_THRESHOLD_SIZE, 117
- POINT_SIZE_MAX, 117
- POINT_SIZE_MIN, 117
- POINT_SMOOTH, 117, 122, 405
- POINT_SMOOTH_HINT, 312
- POINT_SPRITE, 117, 118, 122, 218, 317, 405, 452
- POINT_SPRITE_COORD_ORIGIN, 118, 121, 445, 446
- POINT_TOKEN, 307
- PointParameter, 117, 433
- PointParameter*, 118
- POINTS, 20, 64, 300
- PointSize, 116
- POLYGON, 20, 23, 64, 234, 405
- POLYGON_BIT, 339
- POLYGON_OFFSET_FILL, 134
- POLYGON_OFFSET_LINE, 134
- POLYGON_OFFSET_POINT, 134
- POLYGON_SMOOTH, 129, 134
- POLYGON_SMOOTH_HINT, 312
- POLYGON_STIPPLE, 131, 405
- POLYGON_STIPPLE_BIT, 339
- POLYGON_TOKEN, 307
- PolygonMode, 128, 132, 134, 135, 303, 305, 405
- PolygonOffset, 133
- PolygonStipple, 131, 136, 405
- PopAttrib, 337, 338, 340, 397, 407, 451
- PopClientAttrib, 24, 310, 337, 338, 340, 407
- PopMatrix, 56, 404
- PopName, 302, 407
- POSITION, 81, 317
- POST_COLOR_MATRIX_x_BIAS, 138
- POST_COLOR_MATRIX_x_SCALE, 138
- POST_COLOR_MATRIX_ALPHA_BIAS, 170
- POST_COLOR_MATRIX_ALPHA_SCALE, 170
- POST_COLOR_MATRIX_BLUE_BIAS, 170
- POST_COLOR_MATRIX_BLUE_SCALE, 170
- POST_COLOR_MATRIX_COLOR_TABLE, 140, 171
- POST_COLOR_MATRIX_GREEN_BIAS, 170
- POST_COLOR_MATRIX_GREEN_SCALE, 170
- POST_COLOR_MATRIX_RED_BIAS, 170
- POST_COLOR_MATRIX_RED_SCALE, 170
- POST_CONVOLUTION_x_BIAS, 138
- POST_CONVOLUTION_x_SCALE, 138
- POST_CONVOLUTION_ALPHA_BIAS, 169
- POST_CONVOLUTION_ALPHA_SCALE, 169
- POST_CONVOLUTION_BLUE_BIAS, 169
- POST_CONVOLUTION_BLUE_SCALE, 169
- POST_CONVOLUTION_COLOR_TABLE, 140, 170
- POST_CONVOLUTION_GREEN_BIAS, 169
- POST_CONVOLUTION_GREEN_SCALE, 169
- POST_CONVOLUTION_RED_BIAS, 169
- POST_CONVOLUTION_RED_SCALE, 169

- PREVIOUS, 221, 223, 361
- PRIMARY_COLOR, 223
- PRIMITIVES_GENERATED, 67, 330
- PrioritizeTextures, 217, 218, 406
- PROJECTION, 51, 56, 57
- PROXY_COLOR_TABLE, 140, 143, 311
- PROXY_HISTOGRAM, 147, 148, 311, 327
- PROXY_POST_COLOR_MATRIX_COLOR_TABLE, 140, 311
- PROXY_POST_CONVOLUTION_COLOR_TABLE, 140, 311
- PROXY_TEXTURE_1D, 178, 188, 215, 311, 318
- PROXY_TEXTURE_1D_ARRAY, 178, 187, 215, 311, 318
- PROXY_TEXTURE_2D, 178, 187, 215, 311, 318
- PROXY_TEXTURE_2D_ARRAY, 176, 178, 215, 311, 318
- PROXY_TEXTURE_3D, 176, 215, 311, 318
- PROXY_TEXTURE_CUBE_MAP, 178, 187, 215, 311, 318
- PushAttrib, 337, 338, 340, 407, 451
- PushClientAttrib, 24, 310, 337, 338, 340, 407
- PushMatrix, 56, 404
- PushName, 302, 407
- Q, 59, 60, 317
- QUAD_STRIP, 22, 64, 234, 405
- QUADRATIC_ATTENUATION, 81
- QUADS, 23, 64, 234, 405
- QUERY_BY_REGION_NO_WAIT, 63
- QUERY_BY_REGION_WAIT, 63
- QUERY_COUNTER_BITS, 330
- QUERY_NO_WAIT, 63
- QUERY_RESULT, 330, 452
- QUERY_RESULT_AVAILABLE, 331, 452
- QUERY_WAIT, 63
- R, 59, 60, 317, 456
- R11F_G11F_B10F, 179, 180, 182
- R16, 179, 181
- R16F, 179, 182
- R16I, 179, 182
- R16UI, 179, 182
- R32F, 179, 182
- R32I, 179, 182
- R32UI, 179, 182
- R3_G3_B2, 181
- R8, 179, 181
- R8I, 179, 182
- R8UI, 179, 182
- RASTERIZER_DISCARD, 113
- RasterPos, 70, 108, 294, 303, 396, 434
- RasterPos*, 405
- RasterPos2, 70
- RasterPos3, 70
- RasterPos4, 70
- READ_BUFFER, 268, 291, 295
- READ_FRAMEBUFFER, 264, 278–280, 284, 285, 293, 319, 365
- READ_FRAMEBUFFER_BINDING, 142, 146, 195, 265, 268, 269, 273, 276, 280
- READ_ONLY, 39, 42
- READ_WRITE, 39, 41, 42, 349
- ReadBuffer, 256, 267, 268, 277
- ReadPixels, 66, 135, 137, 151, 152, 154, 165, 264–268, 270, 271, 273, 275, 294, 310, 322, 324, 325, 406, 450
- Rect, 48, 49, 129
- Rect*, 405
- RED, 138, 140, 152, 177, 181, 182, 185, 200–202, 220, 221, 224, 268, 271, 289, 322, 323, 359, 367, 371, 372, 374, 389
- RED_BIAS, 163
- RED_BITS, 294, 407
- RED_INTEGER, 152
- RED_SCALE, 163
- REDUCE, 167, 169, 373
- REFLECTION_MAP, 59, 60, 424
- RENDER, 303, 304, 390

- RENDERBUFFER, 281, 282, 284, 294, 320, 321, 368
- RENDERBUFFER_ALPHA_SIZE, 321
- RENDERBUFFER_BINDING, 282
- RENDERBUFFER_BLUE_SIZE, 321
- RENDERBUFFER_DEPTH_SIZE, 321
- RENDERBUFFER_GREEN_SIZE, 321
- RENDERBUFFER_HEIGHT, 283, 321
- RENDERBUFFER_INTERNAL_FORMAT, 283, 321
- RENDERBUFFER_RED_SIZE, 321
- RENDERBUFFER_SAMPLES, 283, 291, 293, 321
- RENDERBUFFER_STENCIL_SIZE, 321
- RENDERBUFFER_WIDTH, 283, 321
- RenderbufferStorage, 283, 292, 310
- RenderbufferStorageMultisample, 282, 283, 310
- RENDERER, 328
- RenderMode, 303–305, 310, 407
- REPEAT, 201, 206, 214, 358
- REPLACE, 219, 220, 222, 244
- REPLICATE_BORDER, 167, 168
- RESCALE_NORMAL, 57, 404
- ResetHistogram, 327
- ResetMinmax, 327
- RETURN, 263, 264
- RG, 140, 152, 177, 181, 182, 185, 220, 221, 268, 271, 289, 322, 323, 456
- RG16, 179, 181
- RG16F, 179, 182
- RG16I, 179, 182
- RG16UI, 179, 182
- RG32F, 179, 182
- RG32I, 179, 182
- RG32UI, 179, 182
- RG8, 179, 181
- RG8I, 179, 182
- RG8UI, 179, 182
- RG_INTEGER, 152
- RGB, 152, 155, 159, 165, 166, 177, 181, 182, 185, 219–221, 250, 268, 270, 271, 289, 322, 323, 410
- RGB10, 181
- RGB10_A2, 179, 181
- RGB12, 181
- RGB16, 179, 181
- RGB16F, 179, 182
- RGB16I, 179, 182
- RGB16UI, 179, 182
- RGB32F, 179, 182
- RGB32I, 179, 182
- RGB32UI, 179, 182
- RGB4, 181
- RGB5, 181
- RGB5_A1, 181
- RGB8, 179, 181
- RGB8I, 179, 182
- RGB8UI, 179, 182
- RGB9_E5, 140, 179, 180, 182, 226, 270
- RGB_INTEGER, 152
- RGB_SCALE, 219, 451
- RGBA, 141, 142, 145–148, 152, 155, 159, 165, 166, 177, 181–183, 185, 220, 221, 268, 273, 289, 322, 323, 369, 372–375
- RGBA12, 182
- RGBA16, 179, 182
- RGBA16F, 179, 182
- RGBA16I, 179, 183
- RGBA16UI, 179, 183
- RGBA2, 181
- RGBA32F, 179, 182
- RGBA32I, 179, 183
- RGBA32UI, 179, 183
- RGBA4, 181
- RGBA8, 179, 181
- RGBA8I, 179, 182
- RGBA8UI, 179, 182
- RGBA_INTEGER, 152
- RIGHT, 247, 255, 256, 258, 259, 268
- Rotate, 53, 396, 404
- S, 59, 60, 317
- SAMPLE_ALPHA_TO_COVERAGE, 241
- SAMPLE_ALPHA_TO_ONE, 241, 242

- SAMPLE_BUFFERS, 115, 122, 128, 134, 172, 173, 241, 245, 253, 254, 260, 265, 267, 273, 276, 277, 293
- SAMPLE_COVERAGE, 241, 242
- SAMPLE_COVERAGE_INVERT, 241, 242
- SAMPLE_COVERAGE_VALUE, 241, 242
- SampleCoverage, 242
- sampler2D, 100
- SAMPLER_1D, 97
- SAMPLER_1D_ARRAY, 97
- SAMPLER_1D_ARRAY_SHADOW, 97
- SAMPLER_1D_SHADOW, 97
- SAMPLER_2D, 97
- SAMPLER_2D_ARRAY, 97
- SAMPLER_2D_ARRAY_SHADOW, 97
- SAMPLER_2D_SHADOW, 97
- SAMPLER_3D, 97
- SAMPLER_CUBE, 97
- SAMPLER_CUBE_SHADOW, 97
- SAMPLES, 115, 116, 246, 277, 293
- SAMPLES_PASSED, 62, 63, 245, 330
- Scale, 53, 54, 396, 404
- Scissor, 241
- SCISSOR_BIT, 339
- SCISSOR_TEST, 241
- SECONDARY_COLOR_ARRAY, 31, 36
- SECONDARY_COLOR_ARRAY_POINTER, 328
- SecondaryColor, 24, 26, 433
- SecondaryColor3, 26, 445
- SecondaryColor3*, 404
- SecondaryColor3[type]v, 32
- SecondaryColorPointer, 24, 29, 30, 310, 404
- SELECT, 303, 304, 397
- SelectBuffer, 303, 304, 310, 328, 407
- SELECTION_BUFFER_POINTER, 328
- SEPARABLE_2D, 145, 165, 188, 326
- SeparableFilter2D, 136, 145
- SEPARATE_ATTRIBS, 65, 101, 102, 334
- SEPARATE_SPECULAR_COLOR, 77
- SET, 253
- ShadeModel, 85, 405
- SHADER_SOURCE_LENGTH, 333, 335
- SHADER_TYPE, 109, 333
- ShaderSource, 88, 89, 311, 335
- SHADING_LANGUAGE_VERSION, 328, 444, 449
- SHININESS, 81
- SHORT, 29, 151, 271, 272, 309
- SINGLE_COLOR, 76, 77, 352
- SLUMINANCE, 183, 225
- SLUMINANCE8, 225
- SLUMINANCE8_ALPHA8, 224
- SLUMINANCE_ALPHA, 224
- SLUMINANCE_ALPHA8, 183
- SMOOTH, 85, 351
- SOURCE0_ALPHA, 439
- SOURCE0_RGB, 439
- SOURCE1_ALPHA, 439
- SOURCE1_RGB, 439
- SOURCE2_ALPHA, 439
- SOURCE2_RGB, 439
- SPECULAR, 81, 83
- SPHERE_MAP, 59, 60, 424
- SPOT_CUTOFF, 81
- SPOT_DIRECTION, 81, 317
- SPOT_EXPONENT, 81
- SRC0_ALPHA, 439
- SRC0_RGB, 439
- SRC1_ALPHA, 439
- SRC1_RGB, 439
- SRC2_ALPHA, 439
- SRC2_RGB, 439
- SRC_ALPHA, 221, 223, 250, 361
- SRC_ALPHA_SATURATE, 250
- SRC_COLOR, 221, 223, 250, 361, 431
- SRCn_ALPHA, 219, 223, 227, 451
- SRCn_RGB, 219, 223, 227, 451
- SRGB, 224, 247, 248, 251, 320
- SRGB8, 179, 182, 224
- SRGB8_ALPHA8, 179, 182, 224
- SRGB_ALPHA, 224
- STACK_OVERFLOW, 17, 56, 303, 338

- STACK_UNDERFLOW, 17, 56, 303, 338
- STATIC_COPY, 39, 40
- STATIC_DRAW, 39, 40, 349
- STATIC_READ, 39, 40
- STENCIL, 273, 359, 367
- STENCIL_ATTACHMENT, 262, 279, 285, 291
- STENCIL_BITS, 275, 294, 407
- STENCIL_BUFFER, 262, 319
- STENCIL_BUFFER_BIT, 260, 263, 275, 276, 339
- STENCIL_INDEX, 140, 143, 150, 152, 162, 176, 265, 267, 269, 273, 275, 283, 290, 323
- STENCIL_INDEX1, 283
- STENCIL_INDEX16, 283
- STENCIL_INDEX4, 283
- STENCIL_INDEX8, 283
- STENCIL_TEST, 243
- StencilFunc, 243, 244, 395
- StencilFuncSeparate, 243, 244
- StencilMask, 259, 260, 265, 395
- StencilMaskSeparate, 259, 260, 265
- StencilOp, 243, 244
- StencilOpSeparate, 243, 244
- STREAM_COPY, 39, 40
- STREAM_DRAW, 39, 40
- STREAM_READ, 39, 40
- SUBTRACT, 222
- T, 59, 317
- T2F_C3F_V3F, 36, 37
- T2F_C4F_N3F_V3F, 36, 37
- T2F_C4UB_V3F, 36, 37
- T2F_N3F_V3F, 36, 37
- T2F_V3F, 36, 37
- T4F_C4F_N3F_V4F, 36, 37
- T4F_V4F, 36, 37
- TABLE_TOO_LARGE, 17, 141, 147
- TexCoord, 24, 25
- TexCoord*, 404
- TexCoord1, 25
- TexCoord2, 25
- TexCoord3, 25
- TexCoord4, 25
- TexCoordPointer, 24, 29–31, 36, 310, 404
- TexEnv, 55, 56, 218, 227, 406, 451
- TexEnv*, 118
- TexGen, 55, 59, 60, 316
- TexGen*, 404
- TexImage, 56, 193
- TexImage*, 406
- TexImage1D, 136, 165, 167, 184, 188, 189, 191–193, 196, 209, 215, 311
- TexImage2D, 136, 165, 167, 184, 187–189, 191, 193, 196, 209, 215, 311, 450
- TexImage3D, 136, 176, 184, 186, 187, 189, 193, 196, 209, 215, 311, 323
- TexParameter, 56, 200, 406
- TexParameter*, 406
- TexParameter[if], 204, 209
- TexParameterf, 218
- TexParameterfv, 218
- TexParameterI, 200
- TexParameterI, 218
- TexParameterIiv, 200
- TexParameterIuiv, 200
- TexParameteriv, 200, 218
- TexSubImage, 193
- TexSubImage*, 195, 199
- TexSubImage1D, 136, 165, 192–195, 198
- TexSubImage2D, 136, 165, 192–195, 198
- TexSubImage3D, 136, 192, 193, 195, 198
- TEXTURE, 51, 55–57, 221, 223, 286, 290, 294, 320, 321, 361
- TEXTURE_{*i*}, 25, 56
- TEXTURE0, 25, 32, 38, 56, 57, 298, 305, 338, 348, 360
- TEXTURE1, 338
- TEXTURE_{*x*}D, 357
- TEXTURE_ID, 177, 188, 191, 192, 200, 211, 215–217, 226, 286, 317,

- 318, 322, 407
- TEXTURE_1D_ARRAY, 177, 187, 188, 191, 192, 200, 211, 215–217, 317, 318, 322, 357, 407
- TEXTURE_2D, 56, 100, 177, 187, 191, 192, 200, 211, 215–217, 226, 286, 317, 318, 322, 407
- TEXTURE_2D_ARRAY, 176, 178, 186, 192, 197, 199, 200, 211, 215–217, 317, 318, 322, 357, 407
- TEXTURE_3D, 176, 186, 192, 200, 211, 215–217, 226, 286, 317, 318, 322, 407
- TEXTURE_ALPHA_SIZE, 318
- TEXTURE_ALPHA_TYPE, 318
- TEXTURE_BASE_LEVEL, 186, 200, 201, 208, 209, 214, 288, 289
- TEXTURE_BIT, 338, 339, 451
- TEXTURE_BLUE_SIZE, 318
- TEXTURE_BLUE_TYPE, 318
- TEXTURE_BORDER, 197, 199, 319
- TEXTURE_BORDER_COLOR, 200, 201, 207, 214, 318, 406
- TEXTURE_COMPARE_FAIL_VALUE_ARB, 464
- TEXTURE_COMPARE_FUNC, 201, 214, 221, 224
- TEXTURE_COMPARE_MODE, 106, 201, 214, 221, 224, 233, 432
- TEXTURE_COMPONENTS, 319
- TEXTURE_COMPRESSED_IMAGE_SIZE, 197, 199, 319, 324
- TEXTURE_COMPRESSION_HINT, 312
- TEXTURE_COORD_ARRAY, 31, 36
- TEXTURE_COORD_ARRAY_POINTER, 328
- TEXTURE_CUBE_MAP, 178, 188, 200, 211, 215–217, 226, 317, 318, 357, 407
- TEXTURE_CUBE_MAP_*, 188
- TEXTURE_CUBE_MAP_NEGATIVE_X, 187, 191, 192, 202, 286, 318, 322
- TEXTURE_CUBE_MAP_NEGATIVE_Y, 187, 191, 192, 202, 286, 318, 322
- TEXTURE_CUBE_MAP_NEGATIVE_Z, 187, 191, 192, 202, 286, 318, 322
- TEXTURE_CUBE_MAP_POSITIVE_X, 187, 188, 191, 192, 202, 286, 318, 322
- TEXTURE_CUBE_MAP_POSITIVE_Y, 187, 191, 192, 202, 286, 318, 322
- TEXTURE_CUBE_MAP_POSITIVE_Z, 187, 191, 192, 202, 286, 318, 322
- TEXTURE_CUBE_MAP_POSITIVE_X, 367
- TEXTURE_DEPTH, 197–199, 319
- TEXTURE_DEPTH_SIZE, 318
- TEXTURE_DEPTH_TYPE, 318
- TEXTURE_ENV, 218, 219, 317, 407
- TEXTURE_ENV_COLOR, 219
- TEXTURE_ENV_MODE, 219, 227, 425
- TEXTURE_FILTER_CONTROL, 218, 317
- TEXTURE_GEN_*, 404
- TEXTURE_GEN_MODE, 59, 60
- TEXTURE_GEN_Q, 60
- TEXTURE_GEN_R, 60
- TEXTURE_GEN_S, 60
- TEXTURE_GEN_T, 60
- TEXTURE_GREEN_SIZE, 318
- TEXTURE_GREEN_TYPE, 318
- TEXTURE_HEIGHT, 195, 197–199, 319
- TEXTURE_INTENSITY_SIZE, 318
- TEXTURE_INTENSITY_TYPE, 318
- TEXTURE_INTERNAL_FORMAT, 197, 199, 319
- TEXTURE_LOD_BIAS, 201, 203, 218, 434
- TEXTURE_LUMINANCE_SIZE, 318
- TEXTURE_LUMINANCE_TYPE, 318
- TEXTURE_MAG_FILTER, 201, 211, 212, 214, 224

- TEXTURE_MAX_LEVEL, 200, 201, 209, 214, 289
 TEXTURE_MAX_LOD, 200, 201, 204, 214, 452
 TEXTURE_MIN_FILTER, 201, 205, 207–209, 212–214, 224, 288
 TEXTURE_MIN_LOD, 200, 201, 204, 214, 452
 TEXTURE_PRIORITY, 200, 201, 214, 218, 406
 TEXTURE_RECTANGLE_ARB, 466
 TEXTURE_RED_SIZE, 318
 TEXTURE_RED_TYPE, 318
 TEXTURE_RESIDENT, 214, 217, 318
 TEXTURE_SHARED_SIZE, 318
 TEXTURE_STENCIL_SIZE, 318
 TEXTURE_WIDTH, 195, 197–199, 319
 TEXTURE_WRAP_R, 201, 205, 206, 406
 TEXTURE_WRAP_S, 201, 205, 206, 406
 TEXTURE_WRAP_T, 201, 205, 206, 406
 TEXTUREn, 223, 227
 TRANSFORM_BIT, 339
 TRANSFORM_FEEDBACK_BUFFER, 64, 66
 TRANSFORM_FEEDBACK_BUFFER_BINDING, 332
 TRANSFORM_FEEDBACK_BUFFER_MODE, 334
 TRANSFORM_FEEDBACK_BUFFER_SIZE, 332
 TRANSFORM_FEEDBACK_BUFFER_START, 332
 TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN, 66, 67, 330
 TRANSFORM_FEEDBACK_VARYING_MAX_LENGTH, 102, 334
 TRANSFORM_FEEDBACK_VARYINGS, 102, 334
 TransformFeedbackVaryings, 65, 101, 102, 310
 Translate, 53, 54, 396, 404
 TRANSPOSE_COLOR_MATRIX, 316, 324
 TRANSPOSE_MODELVIEW_MATRIX, 316
 TRANSPOSE_PROJECTION_MATRIX, 316
 TRANSPOSE_TEXTURE_MATRIX, 316
 TRIANGLE_FAN, 22, 64
 TRIANGLE_STRIP, 21, 64
 TRIANGLES, 22, 23, 64, 65
 TRUE, 23, 29, 39, 42, 43, 69, 76, 78, 85, 89, 90, 99, 108, 118, 121, 136, 138, 147, 148, 200, 201, 210, 217, 234, 242, 259, 267, 270, 310, 315, 324, 326, 327, 330–333, 336, 337, 343, 351, 356, 363, 364, 405, 431
 ui}v, 262
 Uniform, 98
 Uniform*, 95, 99, 100
 Uniform*f{v}, 98
 Uniform*i{v}, 98
 Uniform*ui{v}, 98
 Uniform1i{v}, 98, 100
 Uniform1iv, 99
 Uniform2{if ui}*, 99
 Uniform4f{v}, 99
 Uniform4i{v}, 99
 Uniform{1,2,3,4}ui, 98
 Uniform{1,2,3,4}uiv, 98
 UniformMatrix*, 446
 UniformMatrix2x4fv, 98
 UniformMatrix3fv, 99
 UniformMatrix{234}fv, 98
 UniformMatrix{2x3,3x2,2x4,4x2,3x4,4x3}fv, 98, 449
 UniformBuffer, 43, 44, 310
 UNPACK_ALIGNMENT, 136, 153, 176, 370
 UNPACK_IMAGE_HEIGHT, 136, 176, 370
 UNPACK_LSB_FIRST, 136, 160, 370

- UNPACK_ROW_LENGTH, 136, 153, 176, 370
- UNPACK_SKIP_IMAGES, 136, 176, 188, 370
- UNPACK_SKIP_PIXELS, 136, 153, 160, 370
- UNPACK_SKIP_ROWS, 136, 153, 160, 370
- UNPACK_SWAP_BYTES, 136, 150, 153, 370
- UNSIGNED_BYTE, 29, 34, 37, 151, 156, 271, 272, 309, 399
- UNSIGNED_BYTE_2_3_3_REV, 151, 155, 156, 272
- UNSIGNED_BYTE_3_3_2, 151, 155, 156, 272
- UNSIGNED_INT, 29, 34, 93, 97, 102, 151, 158, 271, 272, 309, 318–320, 323
- UNSIGNED_INT_10_10_10_2, 151, 155, 158, 272
- UNSIGNED_INT_10F_11F_11F_REV, 151, 155, 158, 159, 270, 272
- UNSIGNED_INT_24_8, 150, 151, 155, 158, 267, 271, 272
- UNSIGNED_INT_2_10_10_10_REV, 151, 155, 158, 272
- UNSIGNED_INT_5_9_9_9_REV, 151, 155, 158, 159, 181, 270, 272
- UNSIGNED_INT_8_8_8_8, 151, 155, 158, 272
- UNSIGNED_INT_8_8_8_8_REV, 151, 155, 158, 272
- UNSIGNED_INT_SAMPLER_1D, 97
- UNSIGNED_INT_SAMPLER_1D_ARRAY, 97
- UNSIGNED_INT_SAMPLER_2D, 97
- UNSIGNED_INT_SAMPLER_2D_ARRAY, 97
- UNSIGNED_INT_SAMPLER_3D, 97
- UNSIGNED_INT_SAMPLER_CUBE, 97
- UNSIGNED_INT_VEC2, 93, 97, 102
- UNSIGNED_INT_VEC3, 93, 97, 102
- UNSIGNED_INT_VEC4, 93, 97, 102
- UNSIGNED_NORMALIZED, 318, 320
- UNSIGNED_SHORT, 29, 34, 151, 157, 271, 272, 309, 319, 323
- UNSIGNED_SHORT_1_5_5_5_REV, 151, 155, 157, 272
- UNSIGNED_SHORT_4_4_4_4, 151, 155, 157, 272
- UNSIGNED_SHORT_4_4_4_4_REV, 151, 155, 157, 272
- UNSIGNED_SHORT_5_5_5_1, 151, 155, 157, 272
- UNSIGNED_SHORT_5_6_5, 151, 155, 157, 272
- UNSIGNED_SHORT_5_6_5_REV, 151, 155, 157, 272
- UPPER_LEFT, 118, 121, 354
- UseProgram, 66, 91, 103
- V2F, 36, 37
- V3F, 36, 37
- VALIDATE_STATUS, 108, 333
- ValidateProgram, 108, 311, 333
- vec2, 92
- vec3, 92
- vec4, 92, 99
- VENDOR, 328
- VERSION, 328, 329
- Vertex, 11, 24, 70, 92, 299
- Vertex*, 404
- Vertex2, 24, 27, 49
- Vertex2sv, 11
- Vertex3, 24, 27
- Vertex3f, 11
- Vertex4, 24, 27
- Vertex[size][type]v, 33
- VERTEX_ARRAY, 31, 38
- VERTEX_ARRAY_BINDING, 316, 336
- VERTEX_ARRAY_POINTER, 328
- VERTEX_ATTRIB_ARRAY_BUFFER_BINDING, 336, 447
- VERTEX_ATTRIB_ARRAY_ENABLED, 336
- VERTEX_ATTRIB_ARRAY_INTEGER, 336

VERTEX_ATTRIB_ARRAY_NORMALIZED, ZERO, 244, 249, 250, 363
336
VERTEX_ATTRIB_ARRAY_POINTER,
336
VERTEX_ATTRIB_ARRAY_SIZE, 336
VERTEX_ATTRIB_ARRAY_STRIDE,
336
VERTEX_ATTRIB_ARRAY_TYPE,
336
VERTEX_PROGRAM_POINT_SIZE,
117
VERTEX_PROGRAM_TWO_SIDE, 78,
79, 405
VERTEX_SHADER, 88, 333
VertexAttrib, 24, 26, 63
VertexAttrib*, 27, 28, 92, 404, 451
VertexAttrib1*, 27
VertexAttrib2*, 27
VertexAttrib3*, 27
VertexAttrib4, 26
VertexAttrib4*, 27
VertexAttrib4N, 27
VertexAttrib4Nub, 27
VertexAttrib[size][type]v, 32
VertexAttrib[size]N[type]v, 32
VertexAttribI, 27
VertexAttribI4, 27
VertexAttribI[size][type]v, 32
VertexAttribIPointer, 29, 30, 48, 310,
336
VertexAttribPointer, 24, 29, 30, 48, 310,
336, 405
VertexPointer, 24, 28, 30, 38, 310, 404
Viewport, 51
VIEWPORT_BIT, 339

WGL_ARB_multisample, 424
WGL_ARB_pixel_format_float, 467
WindowPos, 71, 303, 434, 445
WindowPos*, 405
WindowPos2, 71
WindowPos3, 71
WRITE_ONLY, 39, 42

XOR, 253