OpenGL® ES
Common Profile Specification
Version 2.0.24 (Full Specification)
(April 22, 2009)

Editors (version 2.0): Aaftab Munshi, Jon Leech

Copyright © 2002-2009 The Khronos Group Inc. All Rights Reserved.

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

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

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

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

Contents

1 Introduction **1**
- 1.1 Comments on edits to the OpenGL ES 2.0 Specification 1
- 1.2 What is the OpenGL ES Graphics System? 1
- 1.3 Programmer's View of OpenGL ES 2
- 1.4 Implementor's View of OpenGL ES 2
- 1.5 Our View 3
- 1.6 Companion Documents 3

2 OpenGL ES Operation **4**
- 2.1 OpenGL ES Fundamentals 4
  - 2.1.1 Numeric Computation 6
  - 2.1.2 Data Conversions 7
- 2.2 GL State 8
- 2.3 GL Command Syntax 9
- 2.4 Basic GL Operation 12
- 2.5 GL Errors 13
- 2.6 Primitives and Vertices 14
  - 2.6.1 Primitive Types 15
- 2.7 Current Vertex State 17
- 2.8 Vertex Arrays 18
- 2.9 Buffer Objects 21
  - 2.9.1 Vertex Arrays in Buffer Objects 23
  - 2.9.2 Array Indices in Buffer Objects 24
- 2.10 Vertex Shaders 24
  - 2.10.1 Loading and Compiling Shader Source 25
  - 2.10.2 Loading Shader Binaries 27
  - 2.10.3 Program Objects 28
  - 2.10.4 Shader Variables 30
  - 2.10.5 Shader Execution 38

2.10.6 Required State 40
2.11 Primitive Assembly and Post-Shader Vertex Processing 42
2.12 Coordinate Transformations 42
2.12.1 Controlling the Viewport 43
2.13 Primitive Clipping 44
2.13.1 Clipping Varying Outputs 45

3 Rasterization **46**
3.1 Invariance 47
3.2 Multisampling 47
3.3 Points 49
3.3.1 Point Multisample Rasterization 49
3.4 Line Segments 50
3.4.1 Basic Line Segment Rasterization 50
3.4.2 Other Line Segment Features 52
3.4.3 Line Rasterization State 53
3.4.4 Line Multisample Rasterization 53
3.5 Polygons 54
3.5.1 Basic Polygon Rasterization 55
3.5.2 Depth Offset 56
3.5.3 Polygon Multisample Rasterization 57
3.5.4 Polygon Rasterization State 58
3.6 Pixel Rectangles 58
3.6.1 Pixel Storage Modes 58
3.6.2 Transfer of Pixel Rectangles 59
3.7 Texturing 63
3.7.1 Texture Image Specification 64
3.7.2 Alternate Texture Image Specification Commands 67
3.7.3 Compressed Texture Images 71
3.7.4 Texture Parameters 73
3.7.5 Cube Map Texture Selection 74
3.7.6 Texture Wrap Modes 75
3.7.7 Texture Minification 76
3.7.8 Texture Magnification 80
3.7.9 Texture Framebuffer Attachment 80
3.7.10 Texture Completeness and Non-Power-Of-Two Textures . 81
3.7.11 Mipmap Generation 81
3.7.12 Texture State 82
3.7.13 Texture Objects 82
3.8 Fragment Shaders 84

3.8.1 Shader Variables 84
3.8.2 Shader Execution 85

4 Per-Fragment Operations and the Framebuffer 88
4.1 Per-Fragment Operations 89
4.1.1 Pixel Ownership Test 89
4.1.2 Scissor Test 91
4.1.3 Multisample Fragment Operations 91
4.1.4 Stencil Test 92
4.1.5 Depth Buffer Test 94
4.1.6 Blending 94
4.1.7 Dithering 98
4.1.8 Additional Multisample Fragment Operations 98
4.2 Whole Framebuffer Operations 99
4.2.1 Selecting a Buffer for Writing 99
4.2.2 Fine Control of Buffer Updates 100
4.2.3 Clearing the Buffers 101
4.3 Reading Pixels 102
4.3.1 Reading Pixels 102
4.3.2 Pixel Draw/Read State 105
4.4 Framebuffer Objects 105
4.4.1 Binding and Managing Framebuffer Objects 105
4.4.2 Attaching Images to Framebuffer Objects 108
4.4.3 Renderbuffer Objects 108
4.4.4 Feedback Loops Between Textures and the Framebuffer . 112
4.4.5 Framebuffer Completeness 114
4.4.6 Effects of Framebuffer State on Framebuffer Dependent Values 117
4.4.7 Mapping between Pixel and Element in Attached Image . 118
4.4.8 Errors 118

5 Special Functions 120
5.1 Flush and Finish 120
5.2 Hints 120

6 State and State Requests 122
6.1 Querying GL State 122
6.1.1 Simple Queries 122
6.1.2 Data Conversions 122
6.1.3 Enumerated Queries 123

6.1.4 Texture Queries 125
6.1.5 String Queries 126
6.1.6 Buffer Object Queries 126
6.1.7 Framebuffer Object and Renderbuffer Queries 127
6.1.8 Shader and Program Queries 127
6.2 State Tables 132

A Invariance **156**
A.1 Repeatability 156
A.2 Multi-pass Algorithms 157
A.3 Invariance Rules 157
A.4 What All This Means 158

B Corollaries **159**

C Deleting Shared Objects **161**
C.1 Effect of Shared Object Deletion on Object Namespace 161
C.2 Sharing objects across multiple OpenGL ES contexts 163
C.2.1 Updates to the state of shared objects 163
C.2.2 The effect of shared object deletion on object namespace . 163

D Version 2.0 **165**

E OES Extensions **166**
E.1 Naming Conventions 166
E.2 Promoting Extensions to Core Features 167

F Packaging and Acknowledgements **168**
F.1 Header Files and Libraries 168
F.2 Acknowledgements 168
F.3 Document History 171
F.3.1 Version 2.0.24, updated 2009/04/22 171
F.3.2 Version 2.0.24, draft of 2009/04/01 171
F.3.3 Version 2.0.23, updated 2008/08/27 172
F.3.4 Version 2.0.22, updated 2008/08/06 172
F.3.5 Version 2.0.22, updated 2008/07/17 173
F.3.6 Version 2.0.22, draft of 2008/04/30 173
F.3.7 Version 2.0.22, draft of 2008/04/24 173
F.3.8 Version 2.0.22, draft of 2008/04/08 174
F.3.9 Version 2.0.22, draft of 2008/03/12 174
F.3.10 Version 2.0.22, draft of 2008/01/20 176

F.3.11 Version 2.0.21, draft of 2008/01/11 177
F.3.12 Version 2.0.21, draft of 2008/01/10 178
F.3.13 Version 2.0.21, draft of 2008/01/03 180

List of Figures

2.1 Block diagram of the GL. 12
2.2 Vertex processing and primitive assembly. 15
2.3 Triangle strips, fans, and independent triangles. 16
2.4 Vertex transformation sequence. 42

3.1 Rasterization. 46
3.2 Visualization of Bresenham's algorithm. 51
3.3 Rasterization of non-antialiased wide lines. 52
3.4 The region used in rasterizing a multisampled line segment. 54
3.5 Transfer of pixel rectangles to the GL. 59
3.6 A texture image and the coordinates used to access it. 67

4.1 Per-fragment operations. 89
4.2 Operation of **ReadPixels**. 102

List of Tables

2.1 GL command suffixes 10
2.2 GL data types 11
2.3 Summary of GL errors 14
2.4 Vertex array sizes (values per vertex) and data types 19
2.5 Buffer object parameters and their values. 21
2.6 Buffer object initial state. 23

3.1 **PixelStore** parameters. 59
3.2 **TexImage2D** and **ReadPixels** types. 60
3.3 **TexImage2D** and **ReadPixels** formats. 60
3.4 Valid pixel format and type combinations. 61
3.5 Packed pixel formats. 62
3.6 `UNSIGNED_SHORT` formats 62
3.7 Packed pixel field assignments. 63
3.8 Conversion from RGBA pixel components to internal texture components. 66
3.9 **CopyTexImage** internal format/color buffer combinations. 69
3.10 Texture parameters and their values. 74
3.11 Selection of cube map images. 75
3.12 Correspondence of filtered texture components. 85

4.1 RGB and Alpha blend equations. 96
4.2 Blending functions. 97
4.3 **PixelStore** parameters. 103
4.4 **ReadPixels** GL data types and reversed component conversion formulas. 104
4.5 Renderbuffer image internal formats. 115

6.1 State Variable Types 133
6.2 Vertex Array Data 134

6.3 Buffer Object State 135
6.4 Transformation state 136
6.5 Rasterization 137
6.6 Multisampling 138
6.7 Textures (state per texture unit and binding point) 139
6.8 Textures (state per texture object) 140
6.9 Texture Environment and Generation 141
6.10 Pixel Operations 142
6.11 Framebuffer Control 143
6.12 Pixels 144
6.13 Shader Object State 145
6.14 Program Object State 146
6.15 Vertex Shader State 147
6.16 Hints 148
6.17 Implementation Dependent Values 149
6.18 Implementation Dependent Values (cont.) 150
6.19 Implementation Dependent Values (cont.) 151
6.20 Implementation Dependent Pixel Depths 152
6.21 Miscellaneous 153
6.22 Renderbuffer State 154
6.23 Framebuffer State 155

Chapter 1

Introduction

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

1.1 Comments on edits to the OpenGL ES 2.0 Specification

Changes in the most recent draft are typeset in magenta, as seen in this paragraph.

Editorial comments and questions are typeset in blue.

1.2 What is the OpenGL ES Graphics System?

OpenGL ES is a software interface to graphics hardware. The interface consists of a set of 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 ES requires that the graphics hardware contain a framebuffer. Many OpenGL ES calls pertain to drawing objects such as points, lines and polygons, 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 ES is specifically concerned with framebuffer manipulation.

OpenGL ES 2.0 is based on the OpenGL 2.0 graphics system, but is designed primarily for graphics hardware running on embedded and mobile devices. It re-

moves a great deal of redundant and legacy functionality, while adding a few new features. The differences between OpenGL ES and OpenGL are not described in detail in this specification; however, they are summarized in a companion document titled *OpenGL ES Common Profile Specification 2.0 (Difference Specification)*.

1.3 Programmer's View of OpenGL ES

To the programmer, OpenGL ES 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. OpenGL ES provides an immediate-mode interface, meaning that specifying an object causes it to be drawn.

A typical program that uses OpenGL ES begins with calls to open a window into the framebuffer into which the program will draw. Then, calls are made to allocate an OpenGL ES context and associate it with the window. These steps may be performed using a companion API such as the Khronos Native Platform Graphics Interface (EGL), and are documented separately. Once a context is allocated, the programmer is free to issue OpenGL ES 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 which operate directly on the framebuffer, such as reading pixels.

1.4 Implementor's View of OpenGL ES

To the implementor, OpenGL ES is a set of commands that affect the operation of graphics hardware. If the hardware consists only of an addressable framebuffer, then OpenGL ES 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 ES implementor's task is to provide the CPU software interface while dividing the work for each OpenGL ES 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 ES calls.

OpenGL ES 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, who 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 ES state information explicit, to elucidate how it changes, and to indicate what its effects are.

1.5 Our View

We view OpenGL ES as a state machine that controls 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 Companion Documents

This specification should be read together with a companion document titled *The OpenGL ES Shading Language*. The latter document (referred to as the OpenGL ES Shading Language Specification hereafter) defines the syntax and semantics of the programming language used to write vertex and fragment shaders (see sections 2.10 and 3.8). These sections may include references to concepts and terms (such as shading language variable types) defined in the companion document.

OpenGL ES 2.0 implementations are guaranteed to support at least version 1.0 of the shading language; the actual version supported may be queried as described in section 6.1.5.

Chapter 2

OpenGL ES Operation

2.1 OpenGL ES Fundamentals

OpenGL ES (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, such as the Khronos OpenKODE API, to obtain user input.

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

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

viously invoked GL commands. In general, the effects of a GL command on either GL modes or the framebuffer must be complete before any subsequent command can have any such effects.

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

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

The model for interpretation of GL commands is client-server. That is, a program (the client) issues commands, and these commands are interpreted and processed by the GL (the server). 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 framebuffers and application-created framebuffers. There is always one window-system-provided framebuffer, while application-created framebuffers can 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 window-system-provided framebuffer are ultimately controlled by the window-system that allocates framebuffer resources. It is the window-system that determines which portions of this framebuffer the GL may access at any given time and that communicates to the GL how those portions are structured. Therefore, there are no GL commands to configure the window-system-provided framebuffer or initialize the GL. Similarly, display of framebuffer contents on a monitor or LCD panel (including the transformation of individual framebuffer values by such techniques as gamma correction) is not addressed by the GL. Framebuffer configuration occurs outside of the GL in conjunction with the window-system; the initialization of a GL context occurs when the window system allocates a window for GL rendering. The EGL API defines a portable mechanism for creating GL contexts and windows for rendering into, which may be used in conjunction with different native platform window systems.

The initialization of a GL context itself occurs when the window-system allocates a window for GL rendering and is influenced by the state of the window-system-provided framebuffer.

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 Numeric Computation

The GL must perform a number of numeric computations during the course of its operation.

Implementations will normally perform computations in floating-point, and must meet the range and precision requirements defined under **"Floating-Point Computation"** below.

These requirements only apply to computations performed in GL operations outside of vertex and fragment execution (see sections 2.10 and 3.8), such as texture image specification and per-fragment operations. Range and precision requirements during shader execution differ and are as specified by the OpenGL ES Shading Language Specification.

Floating-Point Computation

We do not specify how floating-point numbers are to be represented or how operations on them are to be performed. 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 for floating-point values must be at least 2^{32}. $x \cdot 0 = 0 \cdot x = 0$. $1 \cdot x = x \cdot 1 = x$. $x + 0 = 0 + x = x$. $0^0 = 1$. (Occasionally further requirements will be specified.) Most single-precision floating-point formats meet these requirements.

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. The identities specified above do not hold if the value of x is not a floating-point number.

Fixed-Point Computation

Vertex attributes may be specified using a 32-bit two's-complement signed representation with 16 bits to the right of the binary point (fraction bits).

General Requirements

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 Data Conversions

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

In the remainder of this section, when an integer type defined in table 2.2 is being discussed, b denotes the minimum required bit width of the integer type as defined in the table. The formulas for conversion to and from unsigned integers also apply to pixel components packed into unsigned integers (see section 3.6.2), but b in these cases is defined by the specific packed pixel format and component being converted.

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

Conversion from Integer to Floating-Point

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

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

Normalized signed integers represent numbers in the range $[-1, 1]$. The conversion from a normalized signed integer c to the corresponding floating-point f is

defined as

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

Conversion from Floating-Point to Integer

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

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

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

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

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

f' is then cast to a signed integer value with exactly b bits of precision.

Conversion from Floating-Point to Framebuffer Fixed-Point

When floating-point values are to be written to the fixed-point color or depth buffers, they must initially lie in $[0, 1]$. Values are converted (by rounding to nearest) to a fixed-point value with m bits, where m is the number of bits allocated to the corresponding R, G, B, A, or depth buffer component. We assume that the fixed-point representation used represents each value $k/(2^m - 1)$, where $k \in \{0, 1, \ldots, 2^m - 1\}$, as k (e.g. 1.0 is represented in binary as a string of all ones). m must be at least as large as the number of bits in the corresponding component of the framebuffer. m must be at least 2 for A if the framebuffer does not contain an A component, or if there is only 1 bit of A in the framebuffer.

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.

2.3 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: 32-bit integer, 32-bit fixed-point, or single-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:

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

and

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

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

rtype **Name**{ϵ**1234**}{ϵ **i f**}{ϵ**v**}
(*[args ,] T arg1 , . . . , T argN [, args]*) ;

rtype is the return type of the function. The braces ({}) enclose a series of characters (or character pairs) of which one is selected. ϵ indicates no character. The

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

Letter	Corresponding GL Type
i	`int`
f	`float`

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

arguments enclosed in brackets (*[args ,]* and *[, args]*) may or may not be present. The N arguments *arg1* through *arg*N have type *T*, which corresponds to one of the type 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.

For example,

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

indicates the eight declarations

```
void Uniform1i( int location, int value );
void Uniform1f( int location, float value );
void Uniform2i( int location, int v0, int v1 );
void Uniform2f( int location, float v0, float v1 );
void Uniform3i( int location, int v0, int v1, int v2 );
void Uniform3f( int location, float v1, float v2,
   float v2 );
void Uniform4i( int location, int v0, int v1, int v2,
   int v3 );
void Uniform4f( int location, float v0, float v1,
   float v2, float v3 );
```

Arguments whose type is fixed (i.e. not indicated by a suffix on the command) are of one of the 13 types (or pointers to one of these) summarized in Table 2.2.

The mapping of GL data types to data types of a specific language binding are part of the language binding definition and may be platform-dependent. Type conversion and type promotion behavior when mixing actual and formal arguments of different data types are specific to the language binding and platform. For example, the C language includes automatic conversion between integer and floating-point data types, but does not include automatic conversion between the `int` and

GL Type	Minimum Bit Width	Description
`boolean`	1	Boolean
`byte`	8	Signed 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
`fixed`	32	Signed 2's complement 16.16 scaled integer
`sizei`	32	Non-negative binary integer size
`enum`	32	Enumerated binary integer value
`intptr`	*ptrbits*	Signed 2's complement binary integer
`sizeiptr`	*ptrbits*	Non-negative binary integer size
`bitfield`	32	Bit field
`float`	32	Floating-point value
`clampf`	32	Floating-point value clamped to $[0, 1]$

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

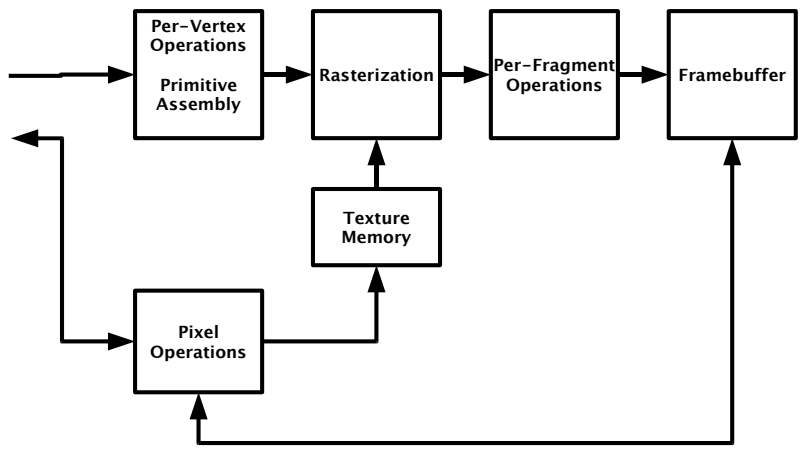


Figure 2.1. Block diagram of the GL.

`fixed`, or `float` and `fixed` GL types since the `fixed` data type is not a distinct built-in type. Regardless of language binding, the `enum` type converts to fixed-point without scaling, and integer types are converted to fixed-point by multiplying by 2^{16}.

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.

The first stage operates on geometric primitives described by vertices: points, line segments, and triangles. 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 triangle. 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, and other operations on fragment values, such as masking (see chapter 4).

Values may also be read back from the framebuffer. 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.

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. Extensions may change behavior that would otherwise generate errors in an unextended GL implementation.

Error	Description	Offending command ignored?
INVALID_ENUM	enum argument out of range	Yes
INVALID_FRAMEBUFFER_OPERATION	Framebuffer is incomplete	Yes
INVALID_VALUE	Numeric argument out of range	Yes
INVALID_OPERATION	Operation illegal in current state	Yes
OUT_OF_MEMORY	Not enough memory left to execute command	Unknown

Table 2.3: Summary of GL errors

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 error 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 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 Primitives and Vertices

In the GL, geometric objects are drawn by specifying a series of generic attribute sets using vertex arrays (see section 2.8). There are seven geometric objects that are drawn this way: points, connected line segments (line strips), line segment loops, separated line segments, triangle strips, triangle fans, and separated triangles.

Each vertex is specified with multiple generic vertex attributes. Each attribute is specified with one, two, three, or four scalar values. Generic vertex attributes can be accessed from within vertex shaders (section 2.10) and used to compute values for consumption by later processing stages.

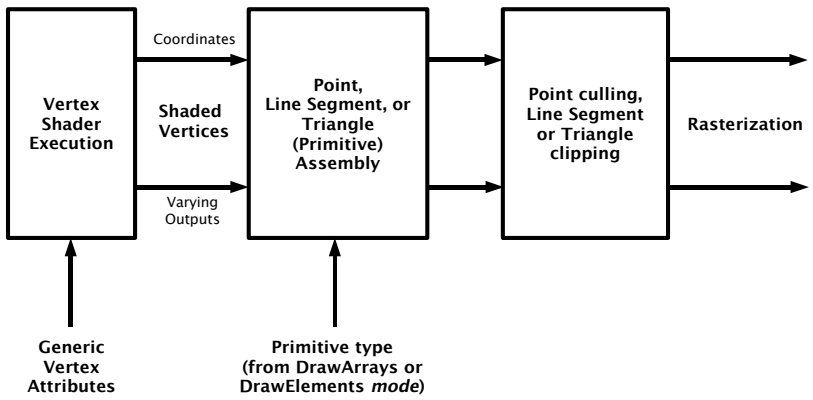


Figure 2.2. Vertex processing and primitive assembly.

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

Before vertex shader execution, the state required by a vertex is its multiple generic vertex attribute sets. After vertex shader execution, the state required by a processed vertex is its screen-space coordinates and any varying outputs written by the vertex shader.

Figure 2.2 shows the sequence of operations that builds a *primitive* (point, line segment, or triangle) 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 and varying outputs. In the case of line and triangle primitives, clipping may insert new vertices into the primitive. The vertices defining a primitive to be rasterized have varying outputs associated with them.

2.6.1 Primitive Types

A sequence of vertices is passed to the GL using the commands **DrawArrays** or **DrawElements** (see section 2.8). There is no limit to the number of vertices that

may be specified, other than the size of the vertex arrays.

The *mode* parameter of these commands determines the type of primitives to be drawn using these coordinate sets. The types, and the corresponding *mode* parameters, are:

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

Line Strips. A series of one or more connected line segments may be specified with *mode* `LINE_STRIP`. At least two vertices must be provided. In this case, the first vertex specifies the first segment's start point while the second vertex specifies the first segment's endpoint and the second segment's start point. In general, the ith vertex (for $i > 1$) specifies the beginning of the ith segment and the end of the $i - 1$st. The last vertex specifies the end of the last segment. If only one vertex is specified, then no primitive is generated.

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

Line Loops. Line loops may be specified with *mode* `LINE_LOOP`. Loops are the same as line strips except that a final segment is added from the final specified vertex to the first vertex.

The required state consists of the processed first vertex, in addition to the state required for line strips.

Separate Lines. Individual line segments, each specified by a pair of vertices, may be specified with *mode* `LINES`. The first two vertices passed define the first segment, with subsequent pairs of vertices each defining one more segment. If the number of specified vertices is odd, then the last one is ignored. The required state is the same as for line strips but it is used differently: a processed vertex holding the first endpoint of the current segment, and a boolean flag indicating whether the current vertex is odd or even (a segment start or end).

Triangle strips. A triangle strip is a series of triangles connected along shared edges, specified by giving a series of defining vertices with *mode* `TRIANGLE_STRIP`. In this case, the first three vertices define the first triangle (and their order is significant). Each subsequent vertex defines a new triangle using that point along with two vertices from the previous triangle. If fewer than three vertices are specified, no primitives are produced. See Figure 2.3.

The required state 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. The pointer is initialized to point to vertex A. Each successive vertex toggles the pointer. Therefore, the first vertex is stored as vertex A, the second stored as vertex B, the third stored as vertex A, and so on. Any vertex after

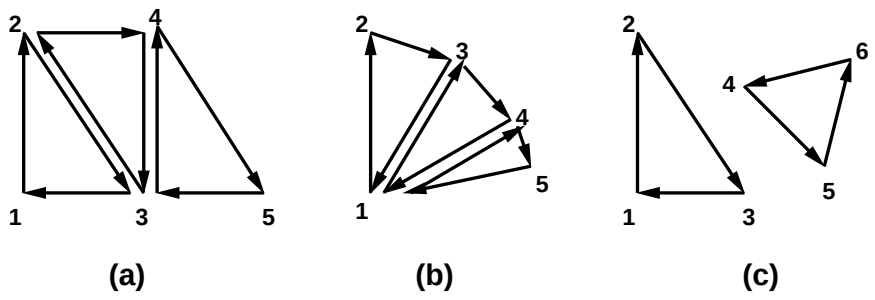


Figure 2.3. (a) A triangle strip. (b) A triangle fan. (c) Independent triangles. The numbers give the sequencing of the vertices in order within the vertex arrays. Note that in (a) and (b) triangle edge ordering is determined by the first triangle, while in (c) the order of each triangle's edges is independent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

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. Triangle fans are specified with *mode* `TRIANGLE_FAN`.

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

The order of the vertices in a triangle generated from a triangle strip, triangle fan, or separate triangles is significant in polygon rasterization and fragment shading (see sections 3.5.1 and 3.8.2).

2.7 Current Vertex State

Vertex shaders (see section 2.10) access an array of 4-component generic vertex attributes. The first slot of this array is numbered 0, and the size of the array is specified by the implementation-dependent constant `MAX_VERTEX_ATTRIBS`.

Current generic attribute values define generic attributes for a vertex when a

vertex array defining that data is not enabled, as described in section 2.8. A current value may be changed at any time by issuing one of the commands

```
void VertexAttrib{1234}{f}( uint index, T values );
void VertexAttrib{1234}{f}v( uint index, T values );
```

to load the given value(s) into the current generic attribute for 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 error `INVALID_VALUE` is generated if *index* is greater than or equal to `MAX_VERTEX_ATTRIBS`.

The **VertexAttrib*** commands can also be used to load attributes declared as a 2×2, 3×3 or 4×4 matrix in a vertex shader. 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 need to be loaded in increasing slot numbers.

The state required to support vertex specification consists of `MAX_VERTEX_ATTRIBS` four-component floating-point vectors to store generic vertex attributes. The initial values for all generic vertex attributes are $(0, 0, 0, 1)$.

2.8 Vertex Arrays

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

```
void VertexAttribPointer( uint index, int size, enum type,
    boolean normalized, sizei stride, const
    void *pointer );
```

describes the locations and organizations of these arrays. *type* specifies the data type of the values stored in the array. *size* indicates the number of values per vertex that are stored in the array. Table 2.4 indicates the allowable values for *size* and *type*. For *type* the values `BYTE`, `UNSIGNED_BYTE`, `SHORT`, `UNSIGNED_SHORT`, `FIXED`, and `FLOAT`, indicate types `byte`, `ubyte`, `short`, `ushort`, `fixed`, and

Command	Sizes	Normalized	Types
VertexAttribPointer	1,2,3,4	*flag*	`byte`, `ubyte`, `short`, `ushort`, `fixed`, `float`

Table 2.4: Vertex array sizes (values per vertex) and data types. The "normalized" column indicates whether integer types are accepted directly or normalized to $[0, 1]$ (for unsigned types) or $[-1, 1]$ (for signed types). For generic vertex attributes, integer data are normalized if and only if the **VertexAttribPointer** *normalized* flag is set.

`float`, 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** command identifies the generic vertex attribute array being described. The error `INVALID_VALUE` is generated if *index* is greater than or equal to `MAX_VERTEX_ATTRIBS`. The *normalized* parameter in the **VertexAttribPointer** command identifies whether integer types should be normalized when converted to floating-point. If *normalized* is `TRUE`, integer data are converted as specified in section 2.1.2; otherwise, the integer values are converted directly.

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

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

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

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

Transferring Array Elements

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

The command

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

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

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

The command

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

constructs a sequence of geometric primitives by successively transferring the *count* elements whose indices are stored in *indices* to the GL. The ith element transferred by **DrawElements** will be taken from element $indices[i]$ of each enabled array. *type* must be one of `UNSIGNED_BYTE` or `UNSIGNED_SHORT`, indicating that the values in *indices* are indices of GL type `ubyte` or `ushort`, respectively. *mode* specifies what kind of primitives are constructed; it accepts the same values as the *mode* parameter of **DrawArrays**. If an array corresponding to a generic attribute required by a vertex shader is not enabled, then the corresponding element is taken from the current generic attribute state (see section 2.7).

If the number of supported generic vertex attributes (the value of `MAX_VERTEX_ATTRIBS`) is n, then the client state required to implement vertex arrays consists of n boolean values, n memory pointers, n integer stride values, n symbolic constants representing array types, n integers representing values per element, and n boolean values indicating normalization. In the initial state, the boolean values are each false, the memory pointers are each `NULL`, the strides are each zero, the array types are each `FLOAT`, and the integers representing values per element are each four.

Name	Type	Initial Value	Legal Values
BUFFER_SIZE	integer	0	any non-negative integer
BUFFER_USAGE	enum	STATIC_DRAW	STATIC_DRAW, DYNAMIC_DRAW, STREAM_DRAW

Table 2.5: Buffer object parameters and their values.

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 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 ARRAY_BUFFER. The binding is effected by calling

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

with *target* set to ARRAY_BUFFER and *buffer* set to the unused name. The resulting buffer object is a new state vector, initialized with a zero-sized memory buffer, and comprising the state values listed in Table 2.5.

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

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

In the initial state the reserved name zero is bound to ARRAY_BUFFER. There is no buffer object corresponding to the name zero, so client attempts to modify or query buffer object state for the target ARRAY_BUFFER while zero is bound will generate GL errors.

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 ARRAY_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 three enumerated values, indicating the expected application usage pattern of the data store. The values are:

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

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

STREAM_DRAW The data store contents will be specified once by the application, and used at most a few times as the source of a GL drawing command.

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 2.6.

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.

Name	Value
BUFFER_SIZE	*size*
BUFFER_USAGE	*usage*

Table 2.6: Buffer object initial state.

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.

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.

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

Rendering commands **DrawArrays** and **DrawElements** operate as previously defined, except that data for enabled generic attribute 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[2].

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

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** is to source its indices from arrays passed as the *indices* parameters.

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.

The commands **BufferData** and **BufferSubData** may be used with *target* set to ELEMENT_ARRAY_BUFFER. In such event, these commands operate in the same fashion as described in section 2.9, but on the buffer currently bound to the ELEMENT_ARRAY_BUFFER target.

While a non-zero buffer object name is bound to ELEMENT_ARRAY_BUFFER, **DrawElements** sources its indices from that buffer object, using elements of the *indices* parameter as offsets into the buffer object in the same fashion as described in section 2.9.1.

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.

2.10 Vertex Shaders

Vertices specified with **DrawArrays** or **DrawElements** are processed by the *vertex shader*. Each vertex attribute consumed by the vertex shader (see section 2.10.4) is set to the corresponding generic vertex attribute value from the array element being

[2] To resume using client-side vertex arrays after a buffer object has been bound, call **BindBuffer**(ARRAY_BUFFER,0) and then specify the client vertex array pointer using the appropriate command from section 2.8.

processed, or from the corresponding current generic attribute if no vertex array is bound for that attribute.

After shader execution, processed vertices are passed on to primitive assembly (see section 2.11).

A vertex shader is defined by 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 ES Shading Language Specification.

To use a vertex shader, shader source code is first loaded into a *shader object* and then *compiled*. Alternatively, pre-compiled shader binary code may be directly loaded into a shader object. An OpenGL ES implementation must support one of these methods for loading shaders. If the boolean value `SHADER_COMPILER` is `TRUE`, then the shader compiler is supported. If the integer value `NUM_SHADER_BINARY_FORMATS` is greater than zero, then shader binary loading is supported

A vertex shader object is 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 3.8. A single program object must contain both a vertex and a fragment shader.

The vertex shader attached to the program object in use by the GL is considered *active*, and is used to process vertices. If no program object is currently in use, the results of vertex shader execution are undefined.

2.10.1 Loading and Compiling Shader Source

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; in this case only the sign of the element in *length* is considered. 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 ES 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 6.1.8). 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 ES 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 6.1.8).

Resources allocated by the shader compiler may be released with the command

void **ReleaseShaderCompiler**(void);

This is a hint from the application, and does not prevent later use of the shader compiler. If shader source is loaded and compiled after **ReleaseShaderCompiler** has been called, **CompileShader** must succeed provided there are no errors in the shader source.

The range and precision for different numeric formats supported by the shader compiler may be determined with the command **GetShaderPrecisionFormat** (see section 6.1.8).

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

If the value of SHADER_COMPILER is not TRUE, then the error INVALID_OPERATION is generated for any call to **ShaderSource**, **CompileShader**, or **ReleaseShaderCompiler**.

2.10.2 Loading Shader Binaries

Precompiled shader binaries may be loaded with the command

void **ShaderBinary**(sizei *count*, const uint **shaders*,
enum *binaryformat*, const void **binary*, sizei *length*);

shaders contains a list of *count* shader object handles. Each handle refers to a unique shader type (vertex shader or fragment shader). *binary* points to *length* bytes of pre-compiled binary shader code in client memory, and *binaryformat* denote the format of the pre-compiled code.

The binary image will be decoded according to the extension specification defining the specified *binaryformat*. OpenGL ES defines no specific binary formats, but does provide a mechanism to obtain token values for such formats provided by extensions. The number of shader binary formats supported can be obtained by querying the value of NUM_SHADER_BINARY_FORMATS. The list of specific binary formats supported can be obtained by querying the value of SHADER_BINARY_FORMATS.

Depending on the types of the shader objects in *shaders*, **ShaderBinary** will individually load binary vertex or fragment shaders, or load an executable binary that contains an optimized pair of vertex and fragment shaders stored in the same binary.

An `INVALID_ENUM` error is generated if *binaryformat* is not a supported format returned in `SHADER_BINARY_FORMATS`. An `INVALID_VALUE` error is generated if the data pointed to by *binary* does not match the specified *binaryformat*. Additional errors corresponding to specific binary formats may be generated as specified by the extensions defining those formats. An `INVALID_OPERATION` error is generated if more than one of the handles refers to the same type of shader (vertex or fragment shader.)

If **ShaderBinary** fails, the old state of shader objects for which the binary was being loaded will not be restored.

Note that if shader binary interfaces are supported, then an OpenGL ES implementation may require that an optimized pair of vertex and fragment shader binaries that were compiled together be specified to **LinkProgram**. Not specifying an optimized pair may cause **LinkProgram** to fail.

2.10.3 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*);

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 not be attached to a single program object. However, a single shader object may be attached to more than one program object.

The error INVALID_OPERATION is generated if *shader* is already attached to *program*, or if another shader object of the same type as *shader* is already attached to *program*.

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

void **DetachShader**(uint *program*, uint *shader*);

If *shader* has been flagged for deletion and is not attached to any other program object, it is deleted.

The error INVALID_OPERATION is generated if *shader* is not attached to *program*. The error INVALID_VALUE is generated if *program* is not a valid program object created with **CreateProgram**.

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 6.1.8). 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 ES Shading Language Specification. Linking will also fail if one or more of the shader objects, attached to *program* are not compiled successfully, if *program* does not contain both a vertex shader and a fragment shader, or if more active uniform or active sampler variables are used in *program* than allowed (see section 2.10.4). 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*. The error INVALID_VALUE is generated if *program* is not a valid program object created with **CreateProgram**.

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 6.1.8).

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 zero, then the current

rendering state refers to an *invalid* program object, and the results of vertex and fragment shader execution due to any **DrawArrays** or **DrawElements** commands are undefined. However, this is not an error. If *program* has not been successfully linked, the error `INVALID_OPERATION` is generated and the current rendering state is not modified.

While a valid 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.

2.10.4 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 variables that are constant during program execution. *Samplers* are a special form of uniform used for texturing (section 3.7). *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 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`, its matrix columns are taken from the (x, y) components of generic attributes i and $i + 1$. When an attribute variable is declared as a `mat3`, its matrix columns are taken from the (x, y, z) components of generic attributes i through $i + 2$. When an attribute variable is declared as a `mat4`, its matrix columns are taken from the (x, y, z, w) components of generic attributes i through $i + 3$.

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

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

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

This command provides information about the attribute selected by *index*. An *index* of 0 selects the first active attribute, and an *index* of `ACTIVE_ATTRIBUTES` -1 selects the last active attribute. The value of `ACTIVE_ATTRIBUTES` can be queried with **GetProgramiv** (see section 6.1.8). If *index* is greater than or equal to `ACTIVE_ATTRIBUTES`, the error `INVALID_VALUE` is generated. Note that *index* simply identifies a member in a list of active attributes, and has no relation to the generic attribute that the corresponding variable is bound to.

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

The name of the selected attribute is returned as a null-terminated string in *name*. The actual number of characters written into *name*, excluding the null terminator, is returned in *length*. If *length* is `NULL`, no length is returned. The maximum number of characters that may be written into *name*, including the null terminator, is specified by *bufSize*. The returned attribute name must be the name of a

generic attribute. The length of the longest attribute name in *program* is given by ACTIVE_ATTRIBUTE_MAX_LENGTH, which can be queried with **GetProgramiv** (see section 6.1.8).

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

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

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

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

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

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

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

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-existant generic attribute (one greater than or equal to `MAX_VERTEX_ATTRIBS`). **LinkProgram** will fail if the attribute bindings assigned by **BindAttribLocation** do not leave enough space to assign a location for an active matrix attribute, which requires multiple contiguous generic attributes.

BindAttribLocation may be issued before any vertex shader objects are attached to a program object. Hence it is allowed to bind any name (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. 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.

Uniform Variables

Shaders can declare named *uniform variables*, as described in the OpenGL ES 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_VECTORS`. This value represents the number of four-element floating-point, integer, or boolean vectors that can be held in uniform variable storage for a

vertex shader. 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 to zero (`FALSE` for booleans). 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 6.1.8). 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 ES 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 6.1.8).

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.

If the active uniform is an array, the uniform name returned in *name* will always be the name of the uniform array appended with "[0]".

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, SAMPLER_2D, or SAMPLER_CUBE.

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 UniformMatrix{234}fv( int location, sizei count,
   boolean transpose, const float *value );
```

The given values are loaded into the uniform variable location identified by *location*.

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

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

The **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. The matrix is specified in column-major order. *transpose* must be `FALSE`.

When loading values for a uniform declared as a boolean, a boolean vector, an array of booleans, or an array of boolean vectors, both the **Uniform*i**{**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`, either **Uniform2i**{**v**} or **Uniform2f**{**v**} can 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 the *transpose* parameter to any of the **UniformMatrix*** commands is not `FALSE`, an `INVALID_VALUE` error is generated, and no uniform values are 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 ES 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, 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 ES Shading Language specification). These values are expected to be interpolated across the primitive being rendered. The OpenGL ES Shading Language specification defines a set of built-in varying variables for vertex shaders corresponding to values required for rasterization following vertex processing.

The number of interpolators available for processing varying variables is given by the implementation-dependent constant `MAX_VARYING_VECTORS`. This value represents the number of four-element floating-point vectors that can be interpolated; varying variables declared as matrices or arrays will consume multiple interpolators. When a program is linked, any varying variable written by a vertex shader, or read by a fragment shader, 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 `MAX_VARYING_VECTORS` worth of varying variables may fail to link, unless device-dependent optimizations are able to make the program fit within available hardware resources.

2.10.5 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.

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

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 implemenation 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 3.7.7 and 3.7.8, and converted to a texture source color C_s according to table 3.12 (section 3.8.2). 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 3.7.7. 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 3.11). 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 3.12) are ignored.

Calling a sampler from a vertex shader will return $(R, G, B, A) = (0, 0, 0, 1)$ under the same conditions as defined for fragment shaders under "Texture Access" in section 3.8.2.

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 (**DrawArrays** or **DrawElements**) is issued, to determine if the currently active program object can be executed. If it cannot be executed then no fragments will be rendered, and the rendering command will generate the error `INVALID_OPERATION`.

This error is generated if:

- any two active samplers in the current program object are of different types, but refer to the same texture image unit,

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 6.1.8). 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. An empty program will always fail validation. 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.

2.10.6 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.

Additionally, one unsigned integer is required to hold the name of the current program object. Initially the current program object is invalid, as if **UseProgram** had been called with *program* set to zero.

2.11 Primitive Assembly and Post-Shader Vertex Processing

Following vertex processing, vertices are assembled into primitives according to the *mode* argument of the drawing command (see sections 2.6.1 and 2.8). The steps of primitive assembly are described in the remaining sections of this chapter and include

- Perspective division on clip coordinates (section 2.12).
- Viewport mapping, including depth range scaling (section 2.12.1).
- Primitive clipping (section 2.13).
- Clipping varying outputs (section 2.13.1).

2.12 Coordinate Transformations

Vertex shader execution yields a vertex coordinate `gl_Position` which is assumed to be in *clip* coordinates. Perspective division is carried out on clip coordinates to yield *normalized device* coordinates, followed by a *viewport* transformation to convert these coordinates into *window coordinates* (see figure 2.4).

Clip coordinates are four-dimensional homogeneous vectors consisting of x, y, z, and w coordinates (in that order). If a vertex's clip coordinates are

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix}$$

then the vertex's normalized device coordinates are

$$\begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} = \begin{pmatrix} \frac{x_c}{w_c} \\ \frac{y_e}{w_c} \\ \frac{z_e}{w_c} \end{pmatrix}.$$

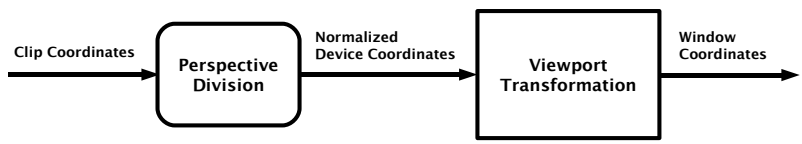


Figure 2.4. Vertex transformation sequence.

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 DepthRangef( clampf n, clampf f );
```

Each of *n* and *f* are clamped to lie within $[0, 1]$, as are all arguments of type `clampf`. z_w is taken to be represented in fixed-point with at least as many bits as there are in the depth buffer of the framebuffer, as described for framebuffer components in section 2.1.2.

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 $o_x = x + \frac{w}{2}$ and $o_y = y + \frac{h}{2}$; $p_x = w$, $p_y = h$.

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 $\frac{w}{2}$ and $\frac{h}{2}$, respectively. n and f are set to 0.0 and 1.0, respectively.

2.13 Primitive Clipping

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

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

If the primitive under consideration is a point, then clipping discards it if it lies outside the near or far clip plane; otherwise it is passed unchanged.

If the primitive is a line segment, then clipping does nothing to it if it lies entirely inside the near and far clip planes, and discards it if it lies entirely outside these planes.

If part of the line segment lies between the near and far clip planes, and part lies outside, then the line segment is clipped against these planes and new vertex coordinates are computed for one or both vertices.

This clipping produces a value, $0 \le t \le 1$, 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

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

If the primitive is a triangle, then it is passed if every one of its edges lies entirely inside the clip volume and either clipped or discarded otherwise. Clipping may cause triangle edges to be clipped, but because 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 triangle, creating a more general *polygon*.

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

A line segment or triangle whose vertices have w_c values of differing signs may generate multiple connected components after clipping. GL implementations are not required to handle this situation. That is, only the portion of the primitive that lies in the region of $w_c > 0$ need be produced by clipping.

2.13.1 Clipping Varying Outputs

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

Let the varying values assigned to the two vertices $\mathbf{P}_1$ and $\mathbf{P}_2$ of an unclipped edge be $\mathbf{c}_1$ and $\mathbf{c}_2$. The value of t (section 2.13) for a clipped point $\mathbf{P}$ is used to obtain the value associated with $\mathbf{P}$ as[3]

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

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

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

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

Chapter 3

Rasterization

Rasterization is the process by which a primitive is converted to a two-dimensional image. Each point of this image contains such information as color and depth. Thus, rasterizing a primitive consists of two parts. The first is to determine which squares of an integer grid in window coordinates are occupied by the primitive. The second is assigning a color and a depth value to each such square. The results of this process are passed on to the next stage of the GL (per-fragment operations), which uses the information to update the appropriate locations in the framebuffer. Figure 3.1 diagrams the rasterization process. The color values assigned to a fragment are determined by a fragment shader (as defined in section 3.8), which uses varying values generated by rasterization operations (sections 3.3 through 3.6.2). The final depth value is determined by the rasterization operations. The results from rasterizing a point, line, or polygon are routed through a fragment shader.

A grid square along with its parameters of assigned z (depth) and varying data 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. Points may be given differing diameters and line segments differing widths. Multisampling must be used to rasterize antialiased primitives (see section 3.2).

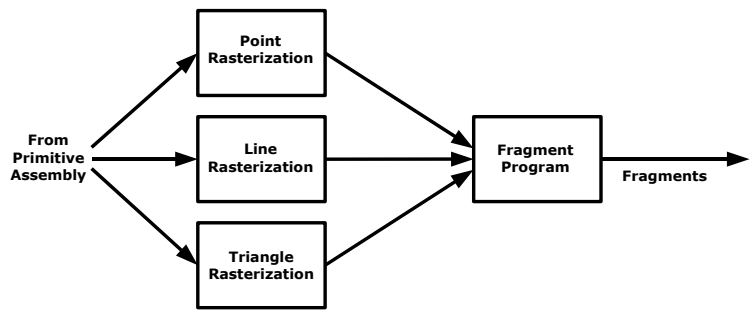


Figure 3.1. Rasterization.

3.1 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.

3.2 Multisampling

Multisampling is a mechanism to antialias all GL primitives: points, lines, and triangles. 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. 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. The color buffer coexists with the multisample buffer, however.

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

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

Multisample rasterization cannot be enabled or disabled after a GL context is created. It is enabled if the value of SAMPLE_BUFFERS is one, and disabled otherwise [1].

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 3.1 is relaxed for all multisample rasterization, because the sample locations may be a function of pixel location.

[1] When using EGL to create OpenGL ES context and surfaces, for example, multisample rasterization is enabled when the EGLConfig used to create a context and surface supports a multisample buffer.

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

3.3 Points

Point size is taken from the shader builtin `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. The range is determined by the `ALIASED_POINT_SIZE_RANGE` and may be queried as described in chapter 6. The maximum point size supported must be at least one.

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 point size.

All fragments produced in rasterizing a point are assigned the same associated data, which are those of the vertex corresponding to the point. However, the `gl_PointCoord` fragment shader input defines a per-fragment coordinate space (s, t) where s varies from 0 to 1 across the point horizontally left-to-right, and t ranges from 0 to 1 across the point vertically top-to-bottom.

The following formulas are used to evaluate (s, t) values:

$$s = \frac{1}{2} + \frac{x_f + \frac{1}{2} - x_w}{size}$$

$$t = \frac{1}{2} - \frac{y_f + \frac{1}{2} - y_w}{size}$$

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.

3.3.1 Point Multisample Rasterization

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

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

3.4 Line Segments

A line segment results from a line strip, a line loop, or a series of separate line segments. Line width may be set by calling

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

with an appropriate positive width to control 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`.

3.4.1 Basic Line Segment Rasterization

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

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

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

Essentially, a line segment starting at $\mathbf{p}_a$ and ending at $\mathbf{p}_b$ produces those fragments f for which the segment intersects R_f, except if $\mathbf{p}_b$ is contained in R_f. See figure 3.2.

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

When $\mathbf{p}_a$ and $\mathbf{p}_b$ lie on fragment centers, this characterization of fragments reduces to Bresenham's algorithm with one modification: lines produced in this description are "half-open," meaning that the final fragment (corresponding to $\mathbf{p}_b$) is not drawn. This means that when rasterizing a series of connected line segments,

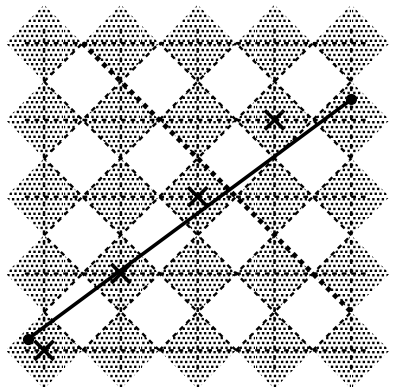


Figure 3.2. Visualization of Bresenham's algorithm. A portion of a line segment is shown. A diamond shaped region of height 1 is placed around each fragment center; those regions that the line segment exits cause rasterization to produce corresponding fragments.

shared endpoints will be produced only once rather than twice (as would occur with Bresenham's algorithm).

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

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

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

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

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

duplicate fragments, nor may any fragments be omitted so as to interrupt continuity of the connected segments.

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

$$t = \frac{(\mathbf{p}_r - \mathbf{p}_a) \cdot (\mathbf{p}_b - \mathbf{p}_a)}{\|\mathbf{p}_b - \mathbf{p}_a\|^2}. \tag{3.1}$$

(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 the clip w coordinate or an element of a vertex shader varying output, is found as

$$f = \frac{(1-t)f_a/w_a + tf_b/w_b}{(1-t)/w_a + t/w_b} \tag{3.2}$$

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, the depth value, window z, must be found using linear interpolation:

$$f = (1-t)f_a + tf_b. \tag{3.3}$$

3.4.2 Other Line Segment Features

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

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 one. 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 3.3). 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

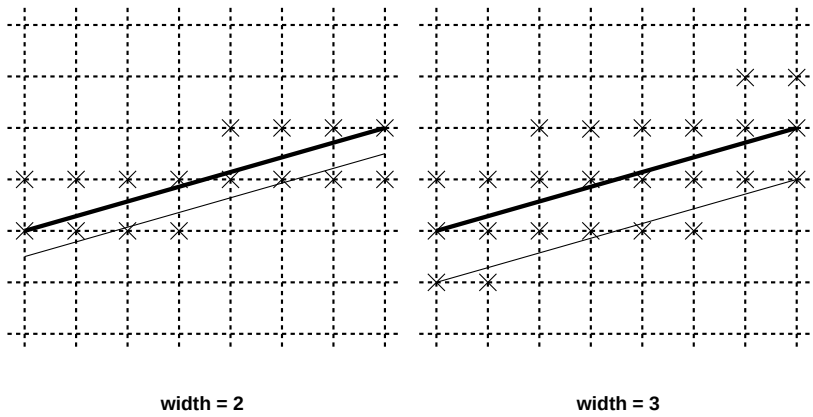


Figure 3.3. Rasterization of non-antialiased wide lines. x-major line segments are shown. The heavy line segment is the one specified to be rasterized; the light segment is the offset segment used for rasterization. x marks indicate the fragment centers produced by rasterization.

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.

3.4.3 Line Rasterization State

The state required for line rasterization consists of the floating-point line width. The initial value of the line width is 1.0.

3.4.4 Line Multisample Rasterization

If the value of `SAMPLE_BUFFERS` is one, then lines are rasterized using the following algorithm. line rasterization produces a fragment for each framebuffer pixel with one or more sample points that intersect a rectangle centered on the line segment (see figure 3.4). Two of the edges are parallel to the specified line segment; each is at a distance of one-half the line width from that segment: one above the

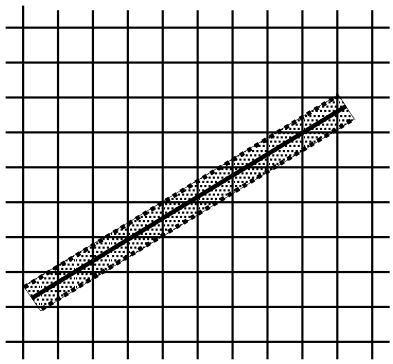


Figure 3.4. The region used in rasterizing a multisampled line segment (an x-major line segment is shown).

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 bits that correspond to sample points that intersect a retained rectangle are 1, other coverage bits are 0. Vertex shader varying outputs and depth are interpolated by substituting the corresponding sample location into equation 3.1, then using the result to evaluate equation 3.2. An implementation may choose to assign the same varying values to more than one sample.

Not all widths need be supported for multisampled line segments, but width 1.0 segments must be provided. As with the point width, the GL implementation may be queried for the range and number of gradations of available multisampled line widths.

3.5 Polygons

A polygon results from a triangle strip, triangle fan, or series of separate triangles. Like points and line segments, polygon rasterization is controlled by several variables.

3.5.1 Basic Polygon Rasterization

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

$$a = \frac{1}{2}\sum_{i=0}^{n-1} x_w^i y_w^{i\oplus 1} - x_w^{i\oplus 1} y_w^i \tag{3.4}$$

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

```
void FrontFace( enum dir );
```

Setting *dir* to `CCW` (corresponding to counter-clockwise orientation of the projected polygon in window coordinates) indicates that the sign of a should be reversed prior to use. Setting *dir* to CW (corresponding to clockwise orientation) uses the sign of a is as computed above. Front face determination requires one bit of state, and is initially set to `CCW`.

If the sign of the area computed by equation 3.4 (including the possible reversal of this sign as indicated by the last call to **FrontFace**) is positive, the polygon is front facing; otherwise, it is back facing. This determination is used in conjunction with the **CullFace** enable bit and mode value to decide whether or not a particular polygon is rasterized. The **CullFace** mode is set by calling

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

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

The rule for determining which fragments are produced by polygon rasterization is called *point sampling*. The two-dimensional projection obtained by taking the x and y window coordinates of the polygon's vertices is formed. Fragment centers that lie inside of this polygon are produced by rasterization. Special treatment is given to a fragment whose center lies on a polygon boundary edge. In

such a case we require that if two polygons lie on either side of a common edge (with identical endpoints) on which a fragment center lies, then exactly one of the polygons results in the production of the fragment during rasterization.

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

$$p = ap_a + bp_b + cp_c,$$

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

$$a = \frac{\mathrm{A}(pp_bp_c)}{\mathrm{A}(p_ap_bp_c)}, \quad b = \frac{\mathrm{A}(pp_ap_c)}{\mathrm{A}(p_ap_bp_c)}, \quad c = \frac{\mathrm{A}(pp_ap_b)}{\mathrm{A}(p_ap_bp_c)},$$

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

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

$$f = \frac{af_a/w_a + bf_b/w_b + cf_c/w_c}{a/w_a + b/w_b + c/w_c} \tag{3.5}$$

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.

Just as with line segment rasterization, the depth value, window z, must be found using linear interpolation:

$$f = af_a + bf_b + cf_c$$

3.5.2 Depth Offset

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

`void` **PolygonOffset**(`float` *factor*, `float` *units*);

factor scales the maximum depth slope of the polygon, and *units* scales an implementation-dependent constant that relates to the usable resolution of the depth buffer. The resulting values are summed to produce the polygon offset value. Both *factor* and *units* may be either positive or negative.

The maximum depth slope m of a triangle is

$$m = \sqrt{\left(\frac{\partial z_w}{\partial x_w}\right)^2 + \left(\frac{\partial z_w}{\partial y_w}\right)^2} \tag{3.6}$$

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

$$m = \max\left\{\left|\frac{\partial z_w}{\partial x_w}\right|, \left|\frac{\partial z_w}{\partial y_w}\right|\right\}. \tag{3.7}$$

The minimum resolvable difference r is an implementation-dependent constant. 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.

The offset value o for a polygon is

$$o = m * factor + r * units. \tag{3.8}$$

m is computed as described above, as a function of depth values in the range [0,1], and o is applied to depth values in the same range.

Boolean state value `POLYGON_OFFSET_FILL` determines whether o is applied during the rasterization of polygons. This boolean state value is enabled and disabled using the commands **Enable** and **Disable**. If `POLYGON_OFFSET_FILL` is enabled, o is added to the depth value of each fragment produced by the rasterization of a polygon.

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.

3.5.3 Polygon Multisample Rasterization

If the value of `SAMPLE_BUFFERS` is one, then polygons are rasterized using the following algorithm. Polygon rasterization produces a fragment for each framebuffer pixel with one or more sample points that satisfy the point sampling criteria described in section 3.5.1, 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.

Coverage bits that correspond to sample points that satisfy the point sampling criteria are 1, other coverage bits are 0. Vertex shader varying outputs and depth are interpolated by substituting the corresponding sample location into the barycentric equations described in section 3.5.1, using equation 3.5 or its approximation that omits w components. An implementation may choose to assign the same set of varying values to more than one sample by barycentric evaluation using any location within the pixel including the fragment center or one of the sample locations.

3.5.4 Polygon Rasterization State

The state required for polygon rasterization consists of the factor and bias values of the polygon offset equation. The initial polygon offset factor and bias values are both 0; initially polygon offset is disabled.

3.6 Pixel Rectangles

Rectangles of color values may be specified to the GL using **TexImage2D** and related commands described in section 3.7.1. Some of the parameters and operations governing the operation of **TexImage2D** are shared by **ReadPixels** (used to obtain pixel values from the framebuffer); the discussion of **ReadPixels**, however, is deferred until section 4.3, after the framebuffer has been discussed in detail. Nevertheless, we note in this section when parameters and state pertaining to **TexImage2D** also pertain to **ReadPixels**.

This section describes only how these rectangles are defined in client memory, and the steps involved in transferring pixel rectangles from client memory to the GL or vice-versa.

Parameters controlling the encoding of pixels in client memory (for reading and writing) are set with the command **PixelStorei**.

3.6.1 Pixel Storage Modes

Pixel storage modes affect the operation of **TexImage2D** and **ReadPixels** (as well as other commands; see section 3.7) when one of these commands is issued. Pixel storage modes are set with the command

`void` **PixelStorei**(`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`.

Parameter Name	Type	Initial Value	Valid Range
UNPACK_ALIGNMENT	integer	4	1,2,4,8

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

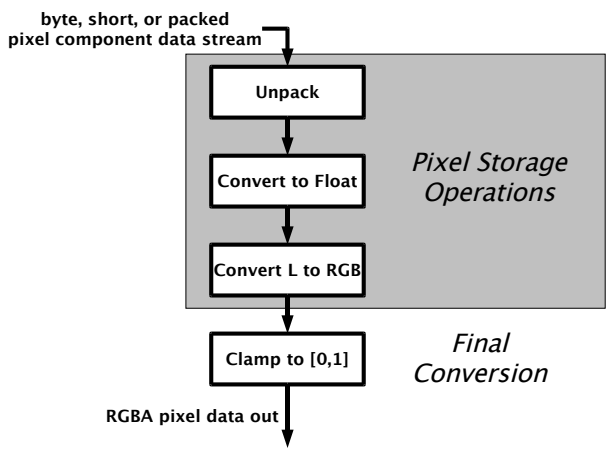


Figure 3.5. Transfer of pixel rectangles to the GL. Output is RGBA pixels.

3.6.2 Transfer of Pixel Rectangles

The process of transferring pixels encoded in client memory to the GL is diagrammed in figure 3.5. We describe the stages of this process in the order in which they occur.

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

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

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

data is a pointer to the data to be drawn. These data are represented with one of two GL data types, specified by *type*. The correspondence between the four *type*

type Parameter Token Name	Corresponding GL Data Type	Special Interpretation
UNSIGNED_BYTE	ubyte	No
UNSIGNED_SHORT_5_6_5	ushort	Yes
UNSIGNED_SHORT_4_4_4_4	ushort	Yes
UNSIGNED_SHORT_5_5_5_1	ushort	Yes

Table 3.2: **TexImage2D** 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 3.6.2. **ReadPixels** accepts only a subset of these types (see section 4.3.1).

Format Name	Element Meaning and Order	Target Buffer
ALPHA	A	Color
RGB	R, G, B	Color
RGBA	R, G, B, A	Color
LUMINANCE	Luminance	Color
LUMINANCE_ALPHA	Luminance, A	Color

Table 3.3: **TexImage2D** and **ReadPixels** formats. The second column gives a description of and the number and order of elements in a group. **ReadPixels** accepts only a subset of these formats (see section 4.3.1).

token values and the GL data types they indicate is given in table 3.2.

Unpacking

Data are taken from client memory as a sequence of unsigned bytes or unsigned shorts (GL data types ubyte and ushort). These elements are grouped into sets of one, two, three, or four values, depending on the *format*, to form a group. Table 3.3 summarizes the format of groups obtained from memory.

The values of each GL data type are interpreted as they would be specified in the language of the client's GL binding.

Not all combinations of *format* and *type* are valid. The combinations accepted by the GL are defined in table 3.4. Additional restrictions may be imposed by specific commands.

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

Format	Type	Bytes per Pixel
RGBA	UNSIGNED_BYTE	4
RGB	UNSIGNED_BYTE	3
RGBA	UNSIGNED_SHORT_4_4_4_4	2
RGBA	UNSIGNED_SHORT_5_5_5_1	2
RGB	UNSIGNED_SHORT_5_6_5	2
LUMINANCE_ALPHA	UNSIGNED_BYTE	2
LUMINANCE	UNSIGNED_BYTE	1
ALPHA	UNSIGNED_BYTE	1

Table 3.4: Valid pixel format and type combinations.

row pointed to by the *data* pointer passed to **TexImage2D**. The number of groups in a row is *width*; If p indicates the location in memory of the first element of the first row, then the first element of the Nth row is indicated by

$$p + Nk \tag{3.9}$$

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

$$k = \begin{cases} nl & s \geq a, \\ a/s \lceil snl/a \rceil & s < a \end{cases} \tag{3.10}$$

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 ubytes, 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.

A *type* of UNSIGNED_SHORT_5_6_5, UNSIGNED_SHORT_4_4_4_4, or UNSIGNED_SHORT_5_5_5_1 is a special case in which all the components of each group are packed into a single unsigned short. The number of components per packed pixel is fixed by the type, and must match the number of components per group indicated by the *format* parameter, as listed in table 3.5. The error INVALID_OPERATION is generated 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.

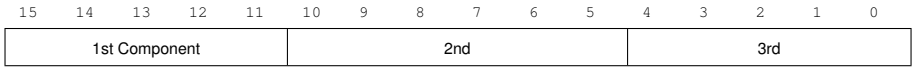
Bitfield locations of the first, second, third, and fourth components of each packed pixel type are illustrated in table 3.6. Each bitfield is interpreted as an unsigned integer value. If the base GL type is supported with more than the minimum precision (e.g. a 9-bit byte) the packed components are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

type Parameter Token Name	GL Data Type	Number of Components	Matching Pixel Formats
UNSIGNED_SHORT_5_6_5	ushort	3	RGB
UNSIGNED_SHORT_4_4_4_4	ushort	4	RGBA
UNSIGNED_SHORT_5_5_5_1	ushort	4	RGBA

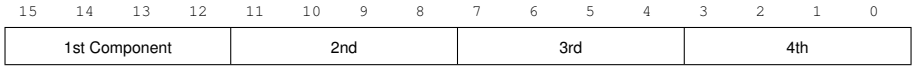
Table 3.5: Packed pixel formats.

Components are packed with the first component in the most significant bits of the bitfield, and successive component occupying progressively less significant locations. The most significant bit of each component is packed in the most significant bit location of its location in the bitfield.

UNSIGNED_SHORT_5_6_5:



UNSIGNED_SHORT_4_4_4_4:



UNSIGNED_SHORT_5_5_5_1:

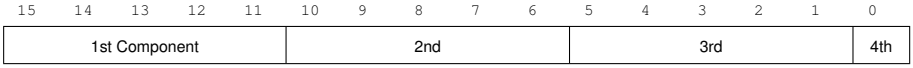


Table 3.6: UNSIGNED_SHORT formats

Format	First Component	Second Component	Third Component	Fourth Component
RGB	red	green	blue	
RGBA	red	green	blue	alpha

Table 3.7: Packed pixel field assignments.

The assignment of component to fields in the packed pixel is as described in table 3.7

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.

Conversion to floating-point

Each element in a group is converted to a floating-point value according to the appropriate formula as described in section 2.1.2 for the corresponding integer, unsigned integer, or unsigned integer bitfield type of that element.

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

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 1.0. If any of R, G, or B is missing from the group, each missing element is added and assigned a value of 0.0.

3.7 Texturing

Texture lookups map a portion of one or more specified images onto a fragment or vertex. This mapping is accomplished in shaders by *sampling* the color of an

image at the location indicated by specified (s, t, r) *texture coordinates*. Texture lookups are typically used to modify a fragment's RGBA color but may be used for any purpose in a shader.

Shaders support texturing using at least MAX_VERTEX_TEXTURE_IMAGE_UNITS images for vertex shaders (see section 2.10.5) and at least MAX_TEXTURE_IMAGE_UNITS images for fragment shaders (see section 3.8.2). Multiple sets of texture coordinates may be specified in generic vertex attributes or computed by the shader; these coordinates are used to sample separate images.

The following subsections (up to and including section 3.7.7) specify GL operation with a single texture, including specification of the image to be texture mapped and the means by which the image is filtered when sampled. The operations described here are applied separately for each texture sampled by a shader.

The details of sampling a texture within a shader are described in the OpenGL ES Shading Language Specification.

The command

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

specifies the active texture image unit selector, ACTIVE_TEXTURE. Each texture image unit consists of all the texture state defined in section 3.7.

The active texture unit selector selects the texture image unit accessed by commands involving texture image processing defined in section 3.7. Such commands include all variants of **TexImage** commands, **BindTexture**, and queries of all such state. If the texture image unit number corresponding to the current value of ACTIVE_TEXTURE is greater than or equal to the implementation-dependent constant MAX_COMBINED_TEXTURE_IMAGE_UNITS, the error INVALID_OPERATION is generated by any such command.

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

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

3.7.1 Texture Image Specification

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 texture image. *target* must be one of TEXTURE_2D for a two-dimensional 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. *format*, *type*, and *data* specify the format of the image data, the type of those data, and a pointer to the image data in client memory, as described in section 3.6.2.

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.

The groups in memory are treated as being arranged in a rectangle. The rectangle is an image, whose size and organization are specified by the *width* and *height* parameters to **TexImage2D**.

The selected groups are processed as described in section 3.6.2, stopping after final expansion to RGBA. Each R, G, B, or A value so generated is clamped to $[0, 1]$.

Components are then selected from the resulting R, G, B, or A values to obtain a texture with the *base internal format* specified by *internalformat*, which must match *format*; no conversions between formats are supported during texture image processing.[2] Table 3.8 summarizes the mapping of R, G, B, and A values to texture components, as a function of the base internal format of the texture image. *internalformat* may be one of the five internal format symbolic constants listed in table 3.8. Specifying a value for *internalformat* that is not one of the above values generates the error INVALID_VALUE. If *internalformat* does not match *format*, the error INVALID_OPERATION is generated.

[2]When a non-RGBA *format* and *internalformat* are specified, implementations are not required to actually create and then discard unnecessary R, G, B, or A components. The abstract model defined by section 3.6.2 is used only for consistency and ease of description.

Base Internal Format	RGBA	Internal Components
ALPHA	A	A
LUMINANCE	R	L
LUMINANCE_ALPHA	R,A	L,A
RGB	R,G,B	R,G,B
RGBA	R,G,B,A	R,G,B,A

Table 3.8: Conversion from RGBA pixel components to internal texture components. Texture components R, G, B, A, and L are converted back to RGBA colors during filtering as shown in table 3.12.

The GL stores the resulting texture with internal component resolutions of its own choosing. The allocation of internal component resolution may vary based on any **TexImage2D** parameter (except *target*), but the allocation must not be a function of any other state and cannot be changed once established. Allocation must be invariant; the same allocation must be chosen each time a texture image is specified with the same parameter values.

The image itself (pointed to by *data*) is a sequence of groups of values. The first group is the lower left 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 the image. 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 3.8. Counting from zero, each resulting Nth texel is assigned internal integer coordinates (i, j), where

$$i = (N \bmod width)$$
$$j = (\lfloor \frac{N}{width} \rfloor \bmod height)$$

Thus the last row of the image is indexed with the highest value of j.

Each color component is converted (by rounding to nearest) to a fixed-point value with n bits, where n is the number of bits of storage allocated to that component in the image array. We assume that the fixed-point representation used represents each value $k/(2^n - 1)$, where $k \in \{0, 1, \ldots, 2^n - 1\}$, as k (e.g. 1.0 is represented in binary as a string of all ones).

The *level* argument to **TexImage2D** 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 and is known as the *level zero array* (or the image array of level zero). If *level* is less than zero, the error INVALID_VALUE is generated. If

level is greater than zero, and either *width* or *height* is not a power of two, the error INVALID_VALUE is generated.

If the *border* argument to **TexImage2D** is not zero, then the error INVALID_VALUE is generated.

If w_t and h_t are the specified image *width* and *height*, and if either w_t or h_t are less than zero, then the error INVALID_VALUE is generated.

The maximum allowable width and height of a two-dimensional texture image must be at least 2^{k-lod} for image arrays of level zero through k, where k is the log base 2 of MAX_TEXTURE_SIZE. and lod is the level-of-detail of the image array. 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.

The maximum allowable width and height of a cube map texture must be the same, and must be at least 2^{k-lod} for image arrays of level zero through k, where k is the log base 2 of MAX_CUBE_MAP_TEXTURE_SIZE.

An implementation may allow an image array of level zero to be created only if that single image array can be supported. Additional constraints on the creation of image arrays of level one or greater are described in more detail in section 3.7.10.

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

We shall refer to the decoded image as the *texture array*. A texture array has width and height w_t and h_t as defined above.

An element (i, j) of the texture array is called a *texel*. The *texture value* used in texturing a fragment is determined by that fragment's associated (s, t) coordinates, but does not necessarily correspond to any actual texel. See figure 3.6.

If the *data* argument of **TexImage2D** is a null pointer (a zero-valued pointer in the C implementation), a texture array is created with the specified *target*, *level*, *internalformat*, *width*, and *height*, 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.

3.7.2 Alternate Texture Image Specification Commands

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*,

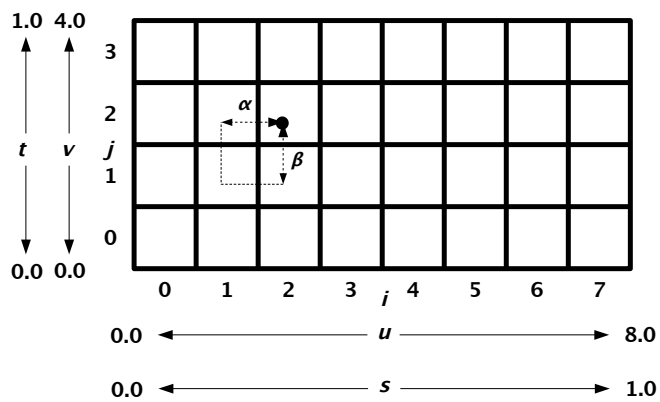


Figure 3.6. A texture image and the coordinates used to access it. This is a texture with $w_t = 8$ and $h_t = 4$. α and β, values used in blending adjacent texels to obtain a texture value, are also shown.

	Texture Format				
Color Buffer	A	L	LA	RGB	RGBA
A	✓	–	–	–	–
RGB	–	✓	–	✓	–
RGBA	✓	✓	✓	✓	✓

Table 3.9: **CopyTexImage** internal format/color buffer combinations.

```
enum internalformat, int x, int y, sizei width,
sizei height, int border );
```

defines a texture array in exactly the manner of **TexImage2D**, except that the image data are taken from the framebuffer rather than from client memory. *target* must be one of `TEXTURE_2D`, `TEXTURE_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, or `TEXTURE_CUBE_MAP_NEGATIVE_Z`. *x*, *y*, *width*, and *height* correspond precisely to the corresponding arguments to **ReadPixels** (refer to section 4.3.1); they specify the image's *width* and *height*, and the lower left (x, y) coordinates of the framebuffer region to be copied. The image is taken from the color buffer of the framebuffer exactly as if these arguments were passed to **ReadPixels** with argument *format* set to `RGBA`, stopping after conversion of RGBA values. Subsequent processing is identical to that described for **TexImage2D**, beginning with clamping of the R, G, B, and A 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**. *internalformat* is further constrained such that color buffer components can be dropped during the conversion to *internalformat*, but new components cannot be added. For example, an RGB color buffer can be used to create `LUMINANCE` or `RGB` textures, but not `ALPHA`, `LUMINANCE_ALPHA`, or `RGBA` textures. Table 3.9 summarizes the allowable framebuffer and base internal format combinations. If the framebuffer format is not compatible with the base texture format, an `INVALID_OPERATION` error is generated. 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.

Two additional commands,

void **TexSubImage2D**(enum *target*, int *level*, int *xoffset*, int *yoffset*, sizei *width*, sizei *height*, enum *format*, enum *type*, void **data*);
void **CopyTexSubImage2D**(enum *target*, int *level*, int *xoffset*, int *yoffset*, int *x*, int *y*, sizei *width*, sizei *height*);

respecify only a rectangular subregion of an existing texture array. No change is made to the *internalformat*, *width*, or *height*, parameters of the specified texture array, nor is any change made to texel values outside the specified subregion. The *target* arguments of **TexSubImage2D** and **CopyTexSubImage2D** must be one of TEXTURE_2D, 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. The *level* parameter of each command specifies the level of the texture array that is modified. If *level* is less than zero or greater than the base 2 logarithm of the maximum texture width or height, the error INVALID_VALUE is generated.

TexSubImage2D arguments *width*, *height*, *format*, *type*, and *data* match the corresponding arguments to **TexImage2D**, meaning that they are specified using the same values, and have the same meanings.

CopyTexSubImage2D arguments *x*, *y*, *width*, and *height* match the corresponding arguments to **CopyTexImage2D**. 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, and A pixel group values to the texture components is controlled by the *internalformat* of the texture 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 texture array being respecified as apply to the *format* and *internalformat* arguments of its **TexImage** counterparts.

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 texture array, address as in figure 3.6. Taking w_t and h_t to be the specified width and height of the texture array, and taking x, y, w, and h to be the *xoffset*, *yoffset*, *width*, and *height* argument values, any of the following relationships generates the error INVALID_VALUE:

$$x < 0$$

$$x + w > w_t$$

$$y < 0$$

$$y + h > h_t$$

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

$$i = x + (n \bmod w)$$

$$j = y + (\lfloor \frac{n}{w} \rfloor \bmod h)$$

Calling **CopyTexImage2D** or **CopyTexSubImage2D** will result in an `INVALID_FRAMEBUFFER_OPERATION` error if the object bound to `READ_FRAMEBUFFER_BINDING` is not framebuffer complete (see section 4.4.5).

Texture Copying Feedback Loops

Calling **CopyTexImage2D** or **CopyTexSubImage2D** will result in undefined behavior if the destination texture image level is also bound to to the selected read buffer (see section 4.3.1) of the read framebuffer. This situation is discussed in more detail in the description of feedback loops in section 4.4.3.

3.7.3 Compressed Texture Images

Texture images may also be specified or modified using image data already stored in a known compressed image format. The GL defines no specific compressed formats, *but compressed formats* may be defined by GL extensions. There is a mechanism to obtain token values for compressed formats; the number of specific compressed internal formats supported 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 *internalformat* parameters accepted by **CompressedTexImage2D** and suitable for general-purpose usage. The renderer will not enumerate formats with restrictions that need to be specifically understood prior to use.

The command

```
void CompressedTexImage2D( enum target, int level,
    enum internalformat, sizei width, sizei height,
    int border, sizei imageSize, void *data );
```

defines a texture image, with incoming data stored in a specific compressed image format. The *target*, *level*, *internalformat*, *width*, *height*, and *border* parameters have the same meaning as in **TexImage2D**. *data* points to compressed image data stored in the compressed image format corresponding to *internalformat*.

For all compressed internal formats, the compressed image will be decoded according to the definition of *internalformat*. Compressed texture images are treated as an array of *imageSize* `ubyte`s beginning at address *data*. 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 not allow *width* or *height* values that are not a multiple of 4. 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 with respect to image contents, meaning that if the GL accepts and stores a texture image in compressed form, **CompressedTexImage2D** will accept any properly encoded compressed texture image of the same width, height, compressed image size, and compressed internal format for storage at the same texture level.

Respecifying Subimages of Compressed Textures

The command

```
void CompressedTexSubImage2D( enum target, int level,
    int xoffset, int yoffset, sizei width, sizei height,
    enum format, sizei imageSize, void *data );
```

respecifies only a rectangular region of an existing texture array, with incoming data stored in a known compressed image format. The *target*, *level*, *xoffset*, *yoffset*, *width*, *height*, and *format* parameters have the same meaning as in **TexSubImage2D**. *data* points to compressed image data stored in the compressed image format corresponding to *format*.

The image pointed to by *data* and the *imageSize* parameter is interpreted as though it was provided to **CompressedTexImage2D**. This command does 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 with respect to image contents, meaning that if the GL accepts and stores a texture image in compressed form, **CompressedTexSubImage2D** will accept any properly encoded compressed texture image of the same width, height, compressed image size, and compressed internal format for storage at the same texture level.

Calling **CompressedTexSubImage2D** will result in an INVALID_OPERATION error if *xoffset* or *yoffset* is not equal to zero, or if *width* and *height* do not match the width and height of the texture, 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.

3.7.4 Texture Parameters

Various parameters control how the texture 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 );
```

target is the target, which must be TEXTURE_2D 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 3.10. In the first form of the command, *param* is a value to which to set a single-valued parameter; in the second form of the command, *params* is an array of parameters whose type depends on the parameter being set.

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	integer	CLAMP_TO_EDGE, REPEAT, MIRRORED_REPEAT
TEXTURE_WRAP_T	integer	CLAMP_TO_EDGE, REPEAT, MIRRORED_REPEAT
TEXTURE_MIN_FILTER	integer	NEAREST, LINEAR, NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, LINEAR_MIPMAP_LINEAR,
TEXTURE_MAG_FILTER	integer	NEAREST, LINEAR

Table 3.10: Texture parameters and their values.

3.7.5 Cube Map Texture Selection

When a cube map sampler is called from a shader, the $\begin{pmatrix} s & t & r \end{pmatrix}$ texture coordinates are treated as a direction vector $\begin{pmatrix} r_x & r_y & r_z \end{pmatrix}$ emanating from the center of a cube (the q coordinate can be ignored, since it merely scales the vector without affecting the direction.) At texture application time, the interpolated per-fragment direction vector selects one of the cube map face's two-dimensional images based on the largest magnitude coordinate direction (the major axis direction). If two or more coordinates have the identical magnitude, the implementation may define the rule to disambiguate this situation. The rule must be deterministic and depend only on $\begin{pmatrix} r_x & r_y & r_z \end{pmatrix}$. The target column in table 3.11 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 3.11, an updated $\begin{pmatrix} s & t \end{pmatrix}$ is calculated as follows:

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

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

This new $\begin{pmatrix} s & t \end{pmatrix}$ is used to find a texture value in the determined face's two-dimensional texture image using the rules given in sections 3.7.6 through 3.7.8.

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

3.7.6 Texture Wrap Modes

Wrap modes defined by the values of TEXTURE_WRAP_S or TEXTURE_WRAP_T respectively affect the interpretation of s and t texture coordinates. The effect of each mode is described below.

Wrap Mode REPEAT

Wrap mode REPEAT ignores the integer part of texture coordinates, using only the fractional part. (For a number f, the fractional part is $f - \lfloor f \rfloor$, regardless of the sign of f; recall that the $\lfloor\rfloor$ function truncates towards $-\infty$.)

REPEAT is the default behavior for all texture coordinates.

Wrap Mode CLAMP_TO_EDGE

Wrap mode CLAMP_TO_EDGE clamps texture coordinates at all mipmap levels such that the texture filter never samples outside the texture image. The color returned when clamping is derived only from texels at the edge of the texture image.

Texture coordinates are clamped to the range $[min, max]$. The minimum value is defined as

$$min = \frac{1}{2N}$$

where N is the size of the texture image in the direction of clamping. The maximum value is defined as

$$max = 1 - min$$

so that clamping is always symmetric about the $[0, 1]$ mapped range of a texture coordinate.

Wrap Mode `MIRRORED_REPEAT`

Wrap mode `MIRRORED_REPEAT` first mirrors the texture coordinate, where mirroring a value f computes

$$\text{mirror}(f) = \begin{cases} f - \lfloor f \rfloor, & \lfloor f \rfloor \text{ is even} \\ 1 - (f - \lfloor f \rfloor), & \lfloor f \rfloor \text{ is odd} \end{cases}$$

The mirrored coordinate is then clamped as described above for wrap mode `CLAMP_TO_EDGE`.

3.7.7 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

$$\lambda(x, y) = \log_2[\rho(x, y)] \tag{3.11}$$

If $\lambda(x, y)$ is less than or equal to the constant c (described below in section 3.7.8) the texture is said to be magnified; if it is greater, the texture is minified.

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 $t(x, y)$ analogously. Let $u(x, y) = w_t \times s(x, y)$ and $v(x, y) = h_t \times t(x, y)$, where w_t and h_t are equal to the width and height of the level zero array. For a polygon, ρ is given at a fragment with window coordinates (x, y) by

$$\rho = \max\left\{\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2}\right\} \tag{3.12}$$

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

$$\rho = \sqrt{\left(\frac{\partial u}{\partial x}\Delta x + \frac{\partial u}{\partial y}\Delta y\right)^2 + \left(\frac{\partial v}{\partial x}\Delta x + \frac{\partial v}{\partial y}\Delta y\right)^2} \Big/ l, \tag{3.13}$$

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, $\rho \equiv 1$.

While it is generally agreed that equations 3.12 and 3.13 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 $|\partial u/\partial x|$, $|\partial u/\partial y|$, $|\partial v/\partial x|$, $|\partial v/\partial y|$,

2. Let

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

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

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

When λ indicates minification, the value assigned to `TEXTURE_MIN_FILTER` is used to determine how the texture value for a fragment is selected. When `TEXTURE_MIN_FILTER` is `NEAREST`, the texel in the level zero array that is nearest (in Manhattan distance) to that specified by (s, t) is obtained. This means the texel at location (i, j) becomes the texture value, with i given by

$$i = \begin{cases} \lfloor u \rfloor, & s < 1 \\ w_t - 1, & s = 1 \end{cases} \tag{3.14}$$

(Recall that if `TEXTURE_WRAP_S` is `REPEAT`, then $0 \leq s < 1$.) Similarly, j is found as

$$j = \begin{cases} \lfloor v \rfloor, & t < 1 \\ h_t - 1, & t = 1 \end{cases} \tag{3.15}$$

When TEXTURE_MIN_FILTER is LINEAR, a 2×2 square of texels in the level zero array is selected. This square is obtained by first wrapping texture coordinates as described in section 3.7.6, then computing

$$i_0 = \begin{cases} \lfloor u - 1/2 \rfloor \bmod w_t, & \texttt{TEXTURE_WRAP_S} \text{ is } \texttt{REPEAT} \\ \lfloor u - 1/2 \rfloor, & otherwise \end{cases}$$

and

$$j_0 = \begin{cases} \lfloor v - 1/2 \rfloor \bmod h_t, & \texttt{TEXTURE_WRAP_T} \text{ is } \texttt{REPEAT} \\ \lfloor v - 1/2 \rfloor, & otherwise \end{cases}$$

Then

$$i_1 = \begin{cases} (i_0 + 1) \bmod w_t, & \texttt{TEXTURE_WRAP_S} \text{ is } \texttt{REPEAT} \\ i_0 + 1, & otherwise \end{cases}$$

and

$$j_1 = \begin{cases} (j_0 + 1) \bmod h_t, & \texttt{TEXTURE_WRAP_T} \text{ is } \texttt{REPEAT} \\ j_0 + 1, & otherwise \end{cases}$$

Let

$$\alpha = \mathrm{frac}(u - 1/2)$$

$$\beta = \mathrm{frac}(v - 1/2)$$

where $\mathrm{frac}(x)$ denotes the fractional part of x.

The texture value τ is found as

$$\tau = (1-\alpha)(1-\beta)\tau_{i_0 j_0} + \alpha(1-\beta)\tau_{i_1 j_0} + (1-\alpha)\beta\tau_{i_0 j_1} + \alpha\beta\tau_{i_1 j_1} \tag{3.16}$$

where τ_{ij} is the texel at location (i, j) in the texture image.

Rendering Feedback Loops

A rendering feedback loop can occur when a texture is attached to an attachment point of the currently bound framebuffer object. In this case rendering results are undefined. The exact conditions are detailed in section 4.4.4.

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 level zero array has dimensions $w_b \times h_b$, then there are $\lfloor \log_2(\max(w_b, h_b)) \rfloor + 1$ image arrays in the mipmap. Each array subsequent to the level zero array has dimensions

$$\max(1, \lfloor \frac{w_b}{2^i} \rfloor) \times \max(1, \lfloor \frac{h_b}{2^i} \rfloor) \rfloor)$$

until the last array is reached with dimension 1×1.

Each array in a mipmap is defined using **TexImage2D** or **CopyTexImage2D**; the array being set is indicated with the level-of-detail argument *level*. Level-of-detail numbers proceed from zero for the original texture array through $q = \lfloor \log_2(\max(w_b, h_b)) \rfloor$ 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 zero through q must be defined, as discussed in section 3.7.10.

If any dimension of any array in a mipmap is not a power of two (e.g. if rounding down as described above is performed), then the mipmap is described as a *non-power-of-two* texture. Non-power-of-two textures have restrictions on the allowed texture wrap modes and filters, as described in section 3.8.2.

The mipmap is used in conjunction with the level of detail to approximate the application of an appropriately filtered texture to a fragment. Let c be the value of λ at which the transition from minification to magnification occurs (since this discussion pertains to minification, we are concerned only with values of λ where $\lambda > c$).

For mipmap filters `NEAREST_MIPMAP_NEAREST` and `LINEAR_MIPMAP_NEAREST`, the dth mipmap array is selected, where

$$d = \begin{cases} 0, & \lambda \leq \frac{1}{2} \\ \lceil \lambda + \frac{1}{2} \rceil - 1, & \lambda > \frac{1}{2}, \lambda \leq q + \frac{1}{2} \\ q, & \lambda > q + \frac{1}{2} \end{cases} \tag{3.17}$$

The rules for `NEAREST` or `LINEAR` filtering are then applied to the selected array.

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

$$d_1 = \begin{cases} q, & \lambda \geq q \\ \lfloor \lambda \rfloor, & otherwise \end{cases} \tag{3.18}$$

$$d_2 = \begin{cases} q, & \lambda \geq q \\ d_1 + 1, & otherwise \end{cases} \tag{3.19}$$

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. The final texture value is then found as

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

3.7.8 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 (equations 3.14 and 3.15 are used); LINEAR behaves exactly as LINEAR for TEXTURE_MIN_FILTER (equation 3.16 is used). The level zero 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 $c = 0$.

3.7.9 Texture Framebuffer Attachment

The texture values are considered undefined if all of the following conditions are true:

- The current FRAMEBUFFER_BINDING names an application-created framebuffer object F.
- The texture is attached to one of the attachment points, A, of framebuffer object F.
- TEXTURE_MIN_FILTER is NEAREST or LINEAR, and the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL for attachment point A is zero; or, TEXTURE_MIN_FILTER is NEAREST_MIPMAP_NEAREST, NEAREST_MIPMAP_LINEAR, LINEAR_MIPMAP_NEAREST, or LINEAR_MIPMAP_LINEAR, and the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL for attachment point A is within the the inclusive range from zero to last mip-level.

3.7.10 Texture Completeness and Non-Power-Of-Two Textures

A texture is said to be complete if all the image arrays and texture parameters required to utilize the texture for texture application is consistently defined.

A two-dimensional texture is *complete* if the following conditions all hold true:

- The set of mipmap arrays zero through q (where q is defined in the **Mipmapping** discussion of section 3.7.7) were each specified with the same format, internal format, and type.
- The dimensions of the arrays follow the sequence described in the **Mipmapping** discussion of section 3.7.7.
- Each dimension of the level zero array is positive.

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

- The level zero arrays of each of the six texture images making up the cube map have identical, positive, and square dimensions.
- The level zero arrays were each specified with the same format, internal format, and type.

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

Effects of Completeness on Texture Application

Texture lookups performed in vertex and fragment shaders are affected by completeness of the texture being sampled as described in sections 2.10.5 and 3.8.2.

Effects of Completeness on Texture Image Specification

An implementation may allow a texture image array of level one or greater to be created only if a complete set of image arrays consistent with the requested array can be supported.

3.7.11 Mipmap Generation

Mipmaps can be generated with the command

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

target is the target, which must be TEXTURE_2D or TEXTURE_CUBE_MAP.

GenerateMipmap computes a complete set of mipmap arrays (as defined in section 3.7.10) derived from the level zero array. Array levels one through q are replaced with the derived arrays, regardless of their previous contents. The level zero array is left unchanged by this computation.

The internal formats of the derived mipmap arrays all match those of the level zero array, and the dimensions of the derived arrays follow the requirements described in section 3.7.10.

The contents of the derived arrays are computed by repeated, filtered reduction of the level zero array. No particular filter algorithm is required, though a box filter is recommended as the default filter.

For cube maps, the error INVALID_OPERATION is generated if the texture bound to *target* is not cube complete.

If either the width or height of the level zero array are not a power or two, the error INVALID_OPERATION is generated.

If the level zero array is stored in a compressed internal format, the error INVALID_OPERATION is generated.

3.7.12 Texture State

The state necessary for texture can be divided into two categories. First, there are the seven sets of mipmap arrays (one for the two-dimensional target and six for the cube map texture targets) and their number. Each array has associated with it a width and height, an integer describing the internal format of the image, six integer values describing the resolutions of each of the red, green, blue, alpha, luminance, and intensity components of the image, a boolean describing whether the image is compressed or not, and an integer size of a compressed image. Each initial texture array is null (zero width and height). Next, there are the two sets of texture properties; each consists of the selected minification and magnification filters, and the wrap modes for s and t. 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 and t wrap modes are both set to REPEAT.

3.7.13 Texture Objects

In addition to the default textures TEXTURE_2D and TEXTURE_CUBE_MAP, named two-dimensional 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_2D 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 3.7.12, set to the same initial values. If the new texture object is bound to TEXTURE_2D, or TEXTURE_CUBE_MAP, it is and remains a two-dimensional or cube map texture respectively until it is deleted.

BindTexture may also be used to bind an existing texture object to either TEXTURE_2D 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.

In the initial state, TEXTURE_2D and TEXTURE_CUBE_MAP have two-dimensional 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 two-dimensional and cube map texture are therefore operated upon, queried, and applied as TEXTURE_2D 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_2D, or TEXTURE_CUBE_MAP is deleted, it is as though **BindTexture** had been executed with the same *target* and *texture* zero. 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 only when they are first bound, just as if they were unused.

The texture object name space, including the initial texture object, 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 is as if all texture units which are bound to that texture object are rebound to texture object zero.

3.8 Fragment Shaders

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

A fragment shader is defined by 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, or polygon. The language used for fragment shaders is described in the OpenGL ES Shading Language Specification.

Alternatively, fragment shaders may be defined by pre-compiled shader binary code, in the same way as described for vertex shaders in section 2.10.

Fragment shaders are created as described in section 2.10.1 using a *type* parameter of `FRAGMENT_SHADER`. They are attached to and used in program objects as described in section 2.10.3.

The fragment shader attached to the program object in use by the GL is considered *active*, and is used to process fragments. If no program object is currently in use, the results of fragment shader execution are undefined.

3.8.1 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_VECTORS`. This value represents the number of four-element floating-point, integer, or boolean vectors that can be held in uniform variable storage for a fragment shader. 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 ES Shading Language Specification defines a set of built-in varying variables that can be be accessed by a fragment shader. These built-in varying variables include the fragment's position, eye z coordinate, and front-facing flag.

Texture Base Internal Format	Texture source color (R_s, G_s, B_s)	Texture source alpha 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
RGB	(R_t, G_t, B_t)	1
RGBA	(R_t, G_t, B_t)	A_t

Table 3.12: Correspondence of filtered texture components to texture source color components. The values R_t, G_t, B_t, A_t, L_t, and I_t are respectively the red, green, blue, alpha, luminance, and intensity components of the filtered texture value τ (see table 3.8.

A vertex shader may define one or more *varying* variables (see section 2.10.4 and the OpenGL ES Shading Language Specification). These values are interpolated across the primitive being rendered. The results of these interpolations are available when varying variables of the same name are defined in the fragment shader.

3.8.2 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, or polygon rasterization.

Texture Access

When a texture lookup is performed in a fragment shader, the GL computes the filtered texture value τ in the manner described in sections 3.7.7 and 3.7.8, and converts it to a texture source color C_s according to table 3.12. The GL returns a four-component vector (R_s, G_s, B_s, A_s) to the fragment shader. For the purposes of level-of-detail calculations, the derivates $\frac{du}{dx}$, $\frac{du}{dy}$, $\frac{dv}{dx}$, $\frac{dv}{dy}$, $\frac{dw}{dx}$ and $\frac{dw}{dy}$ may be approximated by a differencing algorithm as detailed in section 8.8 of the OpenGL ES Shading Language specification.

Calling a sampler from a fragment shader will return $(R, G, B, A) = (0, 0, 0, 1)$ if any of the following conditions are true:

- A two-dimensional sampler is called, the minification filter is one that requires a mipmap (neither NEAREST nor LINEAR), and the sampler's associ-

ated texture object is not complete, as defined in sections 3.7.1 and 3.7.10,

- A two-dimensional sampler is called, the minification filter is not one that requires a mipmap (either NEAREST nor LINEAR), and either dimension of the level zero array of the associated texture object is not positive.
- A two-dimensional sampler is called, the corresponding texture image is a non-power-of-two image (as described in the **Mipmapping** discussion of section 3.7.7), and either the texture wrap mode is not CLAMP_TO_EDGE, or the minification filter is neither NEAREST nor LINEAR.
- A cube map sampler is called, any of the corresponding texture images are non-power-of-two images, and either the texture wrap mode is not CLAMP_TO_EDGE, or the minification filter is neither NEAREST nor LINEAR.
- A cube map sampler is called, and either the corresponding cube map texture image is not cube complete, or TEXTURE_MIN_FILTER is one that requires a mipmap and the texture is not mipmap cube complete.

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 ES 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 3.5.2. The $\frac{1}{w}$ value is computed from the w_c coordinate (see section 2.12).

The built-in variable gl_FrontFacing is set to true if the fragment is generated from a front facing primitive, and false otherwise. For fragments generated from polygon primitives the determination is made by examining the sign of the area computed by equation 3.4 of section 3.5.1 (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.

Shader Outputs

The OpenGL ES Shading Language specification describes the values that may be output by a fragment shader. These are `gl_FragColor` and `gl_FragData[0]` [3]. The final fragment color values or the final fragment data values written by a fragment shader are clamped to the range $[0, 1]$ and then converted to fixed-point as described in section 2.1.2 for framebuffer color components.

Writing to `gl_FragColor` or `gl_FragData[0]` specifies the fragment color (color number zero) that will be used by subsequent stages of the pipeline. 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 both `gl_FragColor` and `gl_FragData[0]`. In this case, a compile or link error will result. A shader statically assigns a value to a variable if, after preprocessing, 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.

[3] `gl_FragData` is supported for compatibility with the desktop OpenGL Shading Language, but only a single fragment color output is allowed in the OpenGL ES Shading Language.

Chapter 4

Per-Fragment Operations and the Framebuffer

The framebuffer consists of a set of pixels arranged as a two-dimensional array. The height and width of this array may vary from one GL implementation to another. 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 also vary depending on the particular GL implementation or context.

Further there are two classes of framebuffers: the default framebuffer supplied by the window-system-provided and application-created framebuffer objects. Every OpenGL ES context has a single default window-system-provided framebuffer. Applications can optionally create additional non-displayable framebuffer objects. For more information on application-created framebuffer objects, see section 4.4.

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

For the default window-system provided framebuffer, the color buffers consist of either or both of a *front* (single) buffer and a *back* buffer. Typically the contents of the front buffer are displayed on a color monitor while the contents of the back buffer are invisible. The color buffers must have the same number of bitplanes, although a context may not provide both types of buffers. Further, an implementation or context may not provide depth or stencil buffers [1].

[1] However, an OpenGL ES implementation must support at least one config with a depth bit depth

For application-created framebuffer objects, the color buffers are not visible, and consequently the names of the color buffers are not related to a display device. The name of the color buffer of an application-created framebuffer object is COLOR_ATTACHMENT0. The names of the depth and stencil buffers are DEPTH_ATTACHMENT and STENCIL_ATTACHMENT. For more information about the buffers of an application-created framebuffer object, see section 4.4.2. To be considered framebuffer complete (see section 4.4.5), all color buffers attached to an application-created framebuffer object must have the same number of bitplanes. Depth and stencil buffers may optionally be attached to application-created framebuffers as well.

Color buffers consist of R, G, B, and, optionally, A unsigned integer values. The number of bitplanes in each of the color buffers, the depth buffer, and the stencil buffer is dependent on the currently bound framebuffer. For the default framebuffer, the number of bitplanes is fixed. For application-created framebuffer objects, however, the number of bitplanes in a given logical buffer may change if the state of the corresponding framebuffer attachment or attached image changes.

The initial state of all provided bitplanes is undefined.

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.

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 of 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.

While an application-created framebuffer object is bound to FRAMEBUFFER, the pixel ownership test always passes. The pixels of application-created frambuffer objects are always owned by OpenGL ES , not the window system. Only while the window-system-provided framebuffer named zero is bound to FRAMEBUFFER does the window system control pixel ownership.

of 16 or higher and a stencil bit depth of 8 or higher

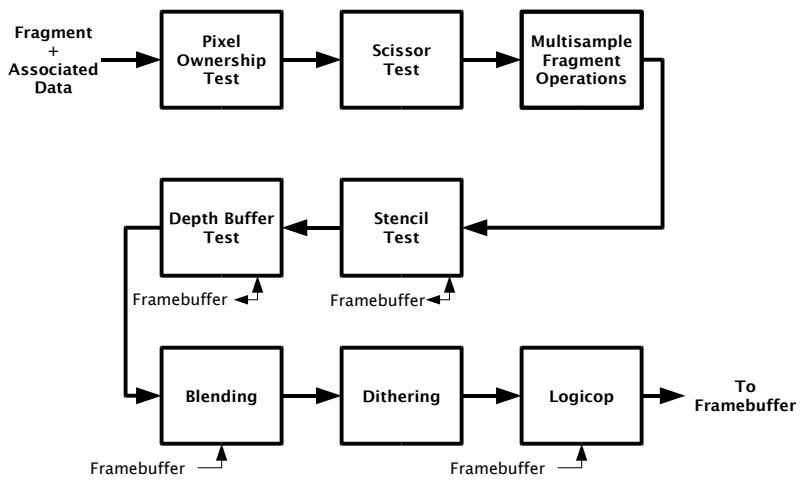


Figure 4.1. Per-fragment operations.

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_COVERAGE, SAMPLE_COVERAGE_VALUE, and SAMPLE_COVERAGE_INVERT. No changes to the fragment alpha or coverage values are made at this step if the value of SAMPLE_BUFFERS is not one.

SAMPLE_ALPHA_TO_COVERAGE and SAMPLE_COVERAGE are enabled and disabled by calling **Enable** and **Disable** with *cap* specified as one of the two token values. Both values are queried by calling **IsEnabled** with *cap* set to the desired token value. If SAMPLE_ALPHA_TO_COVERAGE is enabled, 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.

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. 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.

Finally, if SAMPLE_COVERAGE is enabled, the fragment coverage is ANDed with another temporary coverage. This temporary coverage is generated in the same manner as the one described above, but as a function of the value of SAMPLE_COVERAGE_VALUE. The function need not be identical, but it must have the same properties of proportionality and invariance. If SAMPLE_COVERAGE_-INVERT is TRUE, the temporary coverage is inverted (all bit values are inverted) before it is ANDed with the fragment coverage.

The values of SAMPLE_COVERAGE_VALUE and SAMPLE_COVERAGE_INVERT are specified by calling

void **SampleCoverage**(clampf *value*, boolean *invert*);

with *value* set to the desired coverage value, and *invert* set to TRUE or FALSE. *value* is clamped to [0,1] before being stored as SAMPLE_COVERAGE_VALUE. SAMPLE_COVERAGE_VALUE is queried by calling **GetFloatv** with *pname* set to SAMPLE_COVERAGE_VALUE. SAMPLE_COVERAGE_INVERT is queried by calling **GetBooleanv** with *pname* set to SAMPLE_COVERAGE_INVERT.

4.1.4 Stencil Test

The stencil test conditionally discards a fragment based on the outcome of a comparison between the value in the stencil buffer at location (x_w, y_w) and a reference value. The test is enabled or disabled with the **Enable** and **Disable** commands, using the symbolic constant STENCIL_TEST. When disabled, the stencil test and associated modifications are not made, and the fragment is always passed.

The stencil test is controlled with

void **StencilFunc**(enum *func*, int *ref*, uint *mask*);
void **StencilFuncSeparate**(enum *face*, enum *func*, int *ref*,
uint *mask*);
void **StencilOp**(enum *sfail*, enum *dpfail*, enum *dppass*);
void **StencilOpSeparate**(enum *face*, enum *sfail*, enum *dpfail*,
enum *dppass*);

There are two sets of stencil-related state, the front stencil state set and the back stencil state set. Stencil tests and writes use the front set of stencil state when processing fragments rasterized from non-polygon primitives (points, 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 face culling (see section 3.5.1).

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. It is clamped to the range $[0, 2^s - 1]$, where s is the number of bits in the stencil buffer. The s least significant bits of *mask* are bitwise ANDed with both the reference and the stored stencil value, and the resulting masked values are those that participate in the comparison controlled by *func*. *func* is a symbolic constant that determines the stencil comparison function; the eight symbolic constants are NEVER, ALWAYS, LESS, LEQUAL, EQUAL, GEQUAL, GREATER, or NOTEQUAL. Accordingly, the stencil test passes never, always, and if the masked reference value is less than, less than or equal to, equal to, greater than or equal to, greater than, or not equal to the masked stored value in the stencil buffer.

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

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

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

If the stencil test fails, the incoming fragment is discarded. The state required consists of the most recent values passed to **StencilFunc** or **StencilFuncSeparate** and to **StencilOp** or **StencilOpSeparate**, and a bit indicating whether stencil testing is enabled or disabled. In the initial state, stenciling is disabled, the front and back stencil reference value are both zero, the front and back stencil comparison functions are both ALWAYS, and the front and back stencil mask are both 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**.

4.1.5 Depth Buffer Test

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

The comparison is specified with

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

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

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

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

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

4.1.6 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. Each of these floating-point values is clamped to $[0, 1]$ and converted back to a fixed-point value in the manner described in section 2.1.2 for framebuffer color components. 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 is enabled or disabled using **Enable** or **Disable** with the symbolic constant `BLEND`. If it is disabled, proceed to the next operation.

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`, or `FUNC_REVERSE_SUBTRACT`.

Destination (framebuffer) components are taken to be fixed-point values represented according to the scheme described in section 2.1.2 for framebuffer color components, as are source (fragment) components. Constant color components are taken to be floating-point values.

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 components are treated as if carried out in floating-point.

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,

Mode	RGB Components	Alpha Compoonent
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	$R = R_d * D_r - R_s * S_r$ $G = G_d * D_g - G_s * S_g$ $B = B_d * D_b - B_s * S_b$	$A = A_d * D_a - A_s * S_a$

Table 4.1: RGB and alpha blend equations.

and D_r, D_g, D_b, and D_a are the red, green, blue, and alpha components of the destination weighting factors determined by the destination blend function. Blend functions are described below.

Blend Functions

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

```
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 );
```

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

The four parameters are clamped to the range $[0, 1]$ before being stored. The constant color can be used in both the source and destination blending functions

Blending State

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

Blending occurs once for each color buffer currently enabled for writing (section 4.2.1) using each buffer's color for C_d. If a color buffer has no A value, then A_d is taken to be 1.

4.1.7 Dithering

Dithering selects between two color values. Consider the value of any of the color components as a fixed-point value with m bits to the left of the binary point, where m is the number of bits allocated to that component in the framebuffer; call each such value c. For each c, dithering selects a value c_1 such that $c_1 \in \{\max\{0, \lceil c \rceil - 1\}, \lceil c \rceil\}$ (after this selection, treat c_1 as a fixed point value in [0,1] with m bits). This selection may depend on the x_w and y_w coordinates of the pixel. c must not be larger than the maximum value representable in the framebuffer for either the component or the index, as appropriate.

Many dithering algorithms are possible, but a dithered value produced by any algorithm must depend only the incoming value and the fragment's x and y window coordinates. If dithering is disabled, then each color component is truncated to a fixed-point value with as many bits as there are in the corresponding component in the framebuffer.

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.

4.1.8 Additional Multisample Fragment Operations

If the value of `SAMPLE_BUFFERS` is one, the stencil test, depth test, blending, and dithering operations are performed for each pixel sample, rather than just once for

each fragment. Failure of the stencil or depth test results in termination of the processing of that sample, rather than discarding of the fragment. All operations are performed on the color, depth, and stencil values stored in the multisample buffer (to be described in a following section). The contents of the color buffer 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 corresponding coverage bit is 0, no operations are performed for that sample.

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

After all operations have been completed on the multisample buffer, the color sample values are combined to produce a single color value, and that value is written into the color buffer selected for writing (see section 4.2.1). An implementation may defer the writing of the color buffer until a later time, but the state of the framebuffer must behave as if the color buffer was 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

Color values are written into the front buffer for single buffered contexts, or into the back buffer for back buffered contexts. The type of context is determined when creating a GL context.

4.2.2 Fine Control of Buffer Updates

Four commands are used to mask the writing of bits to each of the logical framebuffers after all per-fragment operations have been performed. The command

```
void ColorMask( boolean r, boolean g, boolean b,
    boolean a );
```

controls the writing of R, G, B and A values to the color buffer. *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). In the initial state, all color values are enabled for writing.

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 4.1.4). The clear operation always uses the front stencil write mask when clearing the stencil buffer.

The state required for the various masking operations is 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 Update s

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

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, and STENCIL_BUFFER_BIT, indicating the color buffer, the depth buffer, and the stencil buffer, 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 the color buffer. Each of the specified components is clamped to $[0, 1]$ and converted to fixed-point as described in section 2.1.2 for framebuffer color components.

void **ClearDepthf**(clampf *d*);

takes a value that is clamped to the range $[0, 1]$ and converted to fixed-point according to the rules for a window z value given in section 2.12.1. Similarly,

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

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

When **Clear** is called, the only per-fragment operations that are applied (if enabled) are the pixel ownership test, the scissor test, and dithering. The masking operations described in the last section (4.2.2) are also effective. If a buffer is not present, then a **Clear** directed at that buffer has no effect.

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

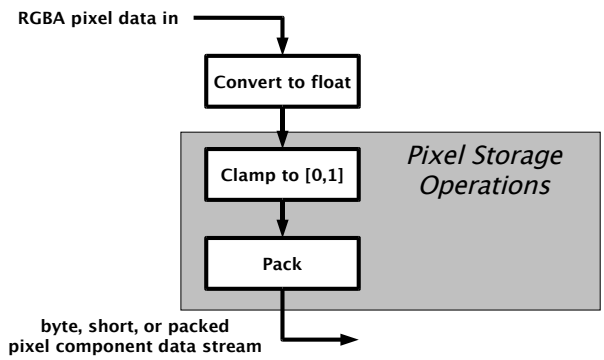


Figure 4.2. Operation of **ReadPixels**. Operations in dashed boxes may be enabled or disabled.

Clearing the Multisample Buffer

The color samples of the multisample buffer are cleared when the color buffer is cleared, as specified by the **Clear** mask bit `COLOR_BUFFER_BIT`.

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

4.3 Reading Pixels

Pixels may be read from the framebuffer to client memory using the **ReadPixels** commands, as described below. Pixels may also be copied from client memory or the framebuffer to texture images in the GL using the **TexImage2D** and **CopyTexImage2D** commands, as described in section 3.7.1.

4.3.1 Reading Pixels

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

Parameter Name	Type	Initial Value	Valid Range
PACK_ALIGNMENT	integer	4	1,2,4,8

Table 4.3: **PixelStore** parameters pertaining to **ReadPixels**.

Pixels are read using

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

The arguments after *x* and *y* to **ReadPixels** are those described in section 3.6.2 defining pixel rectangles. Only two combinations of *format* and *type* are accepted. The first is *format* RGBA and *type* UNSIGNED_BYTE. The second is an implementation-chosen format from among those defined in table 3.4, excluding formats LUMINANCE and LUMINANCE_ALPHA. The values of *format* and *type* for this format may be determined by calling **GetIntegerv** with the symbolic constants IMPLEMENTATION_COLOR_READ_FORMAT and IMPLEMENTATION_-COLOR_READ_TYPE, respectively. The implementation-chosen format may vary depending on the format of the currently bound rendering surface. Unsupported combinations of *format* and *type* will generate an INVALID_OPERATION error. The pixel storage modes that apply to **ReadPixels** are summarized in Table 4.3.

Obtaining Pixels from the Framebuffer

The buffer from which values are obtained is the color buffer used for writing (see section 4.2.1). If FRAMEBUFFER_BINDING is non-zero, pixel values are read from the buffer attached as the COLOR_ATTACHMENT0 attachment to the currently bound framebuffer object.

ReadPixels obtains values from the color buffer (with lower left hand corner at $(0, 0)$) for each pixel $(x + i, y + j)$ for $0 \leq i < width$ and $0 \leq j < height$; this pixel is said to be the ith pixel in the jth row. If any of these pixels lies outside of the window allocated to the current GL context, the values obtained for those pixels are undefined. Results are also undefined for individual pixels that are not owned by the current context. Otherwise, **ReadPixels** obtains values from the color buffer, regardless of how those values were placed there.

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.

type Parameter	GL Data Type	Component Conversion Formula
INT	int	$c = [(2^{32} - 1)f - 1]/2$
UNSIGNED_BYTE	ubyte	$c = (2^8 - 1)f$
UNSIGNED_SHORT_5_6_5	ushort	$c = (2^N - 1)f$
UNSIGNED_SHORT_4_4_4_4	ushort	$c = (2^N - 1)f$
UNSIGNED_SHORT_5_5_5_1	ushort	$c = (2^N - 1)f$

Table 4.4: Reversed component conversions, used when component data are being returned to client memory. Color 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.

Conversion of RGBA values

The R, G, B, and A values form a group of elements. Each element is taken to be a fixed-point value in $[0, 1]$ with m bits, as described in section 2.1.2 for framebuffer color components.

Final Conversion

Each component is first clamped to $[0, 1]$. Then the appropriate conversion formula from table 4.4 is applied to the component.

Placement in Client Memory

Groups of elements are placed in memory just as they are taken from memory for **TexImage2D**. That is, the ith group of the jth row (corresponding to the ith pixel in the jth row) is placed in memory just where the ith group of the jth row would be taken from for **TexImage2D**. See **Unpacking** under section 3.6.2. The only difference is that the storage mode parameters whose names begin with PACK_ are used instead of those whose names begin with UNPACK_. If *format* is ALPHA, only the corresponding single element is written. Otherwise all the elements of each group are written.

4.3.2 Pixel Draw/Read State

The state required for pixel operations consists of the parameters that are set with **PixelStore**. This state has been summarized in tables 3.1. State set with **PixelStore** is GL client state.

4.4 Framebuffer Objects

As described in chapters 1 and 2, OpenGL ES renders into (and reads values from) a framebuffer. OpenGL ES defines two classes of framebuffers: window-system-provided framebuffers and application-created framebuffers.

By default, OpenGL ES uses the window-system-provided 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 window-system-provided framebuffer, including its images, can not be changed by OpenGL ES , nor can the window-system-provided framebuffer itself, or its images, be deleted by OpenGL ES .

The routines described in the following sections, however, can be used to create, destroy, and modify the state and attachments of application-created framebuffer objects.

Application-created 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. 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.

By allowing the images of a renderbuffer to be attached to a framebuffer, OpenGL ES provides a mechanism to support *off-screen* rendering. Further, by allowing the images of a texture to be attached to a framebuffer, OpenGL ES provides a mechanism to support *render to texture*.

4.4.1 Binding and Managing Framebuffer Objects

The operations described in chapter 4 affect the images attached to the framebuffer object bound to the target `FRAMEBUFFER`. By default, the framebuffer bound to the target `FRAMEBUFFER` is zero, specifying the default implementation-dependent framebuffer provided by the windowing system. When the framebuffer bound to target `FRAMEBUFFER` is not zero, but instead names an application-created frame-

buffer object, then the operations described in chapter 4 affect the application-created framebuffer object rather than the default framebuffer.

The namespace for framebuffer objects is the unsigned integers, with zero reserved by OpenGL ES to refer to the default framebuffer. A framebuffer object is created by binding an unused name to the target FRAMEBUFFER. The binding is effected by calling

void **BindFramebuffer**(enum *target*, uint *framebuffer*);

with *target* set to FRAMEBUFFER and *framebuffer* set to the unused name. The resulting framebuffer object is a new state vector. There is one color attachment point, plus one each for the depth and stencil attachment points.

BindFramebuffer may also be used to bind an existing framebuffer object to *target*. 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. The current FRAMEBUFFER binding can be queried using GetIntegerv(FRAMEBUFFER_BINDING).

While a framebuffer object is bound to the target FRAMEBUFFER, OpenGL ES 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. In particular, queries of the values specified in table 6.20 (Implementation Dependent Pixel Depths) are derived from the currently bound framebuffer object. The framebuffer object bound to the target FRAMEBUFFER is used as the destination of fragment operations and as the source of pixel reads such as **ReadPixels**.

In the initial state, the reserved name zero is bound to the target FRAMEBUFFER. There is no application created framebuffer object corresponding to the name zero. Instead, the name zero refers to the window-system-provided framebuffer. All queries and operations on the framebuffer while the name zero is bound to the target FRAMEBUFFER operate on this default framebuffer. On some implementations, the properties of the default window system provided framebuffer can change over time (e.g., in response to window system events such as attaching the context to a new window system drawable.)

Application created framebuffer objects (i.e., those with a non-zero name) differ from the default window-system-provided framebuffer in a few important ways. First and foremost, unlike the window-system-provided framebuffer, application created framebuffers have modifiable attachment points for each logical buffer in the framebuffer. Framebuffer attachable images can be attached to and detached from these attachment points. Also, the size and format of the images attached to application created framebuffers are controlled entirely within the OpenGL ES

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, application-created 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, or front color bitplanes.
- The only color buffer bitplanes are the ones defined by the framebuffer attachment point named `COLOR_ATTACHMENT0`.
- 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 is no multisample buffer, so the value of the implementation-dependent state variables `SAMPLES` and `SAMPLE_BUFFERS` are both 0.

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 the target `FRAMEBUFFER` is deleted, it is as though **BindFramebuffer** had been executed with the *target* of `FRAMEBUFFER` and *framebuffer* of zero. Unused names in *framebuffers* are silently ignored, as is the value zero.

The command

`void` **GenFramebuffers**(`sizei` *n*, `uint` **framebuffers*);

returns *n* previously unused framebuffer object names in *framebuffers*. 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.

4.4.2 Attaching Images to Framebuffer Objects

Framebuffer-attachable images may be attached to, and detached from, application-created framebuffer objects. In contrast, the image attachments of the window-system-provided framebuffer may not be changed by OpenGL ES .

A single framebuffer-attachable image may be attached to multiple application-created framebuffer objects, potentially avoiding some data copies, and possibly decreasing memory consumption.

For each logical buffer, the framebuffer object stores a set of state which defines the logical buffer's *attachment point*. The attachment point state contains enough information to identify the single image attached to the attachment point, or to indicate that no image is attached. The per-logical buffer attachment point state is listed in table 6.23.

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

4.4.3 Renderbuffer Objects

A renderbuffer is a data storage object containing a single image of a renderable internal format. OpenGL ES 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 OpenGL ES . A renderbuffer object is created by binding an unused name to `RENDERBUFFER`. The binding is effected by calling

`void` **BindRenderbuffer**(`enum` *target*, `uint` *renderbuffer*);

with *target* set to `RENDERBUFFER` and *renderbuffer* set to the unused 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 6.22. 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, OpenGL ES 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 errors.

Using **GetIntegerv**, the current RENDERBUFFER binding can be queried as RENDERBUFFER_BINDING.

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. 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 RenderbufferStorage( enum target, enum internalformat,
    sizei width, sizei height );
```

establishes the data storage, format, and dimensions of a renderbuffer object's image. *target* must be RENDERBUFFER. *internalformat* must be one of the color-renderable, depth-renderable, or stencil-renderable formats described in table 4.5. *width* and *height* are the dimensions in pixels of the renderbuffer. If either *width* or *height* is greater than the value of MAX_RENDERBUFFER_SIZE, the error INVALID_VALUE is generated. If OpenGL ES is unable to create a data store of the requested size, the error OUT_OF_MEMORY is generated. **RenderbufferStorage** deletes any existing data store for the renderbuffer and the contents of the data store after calling **RenderbufferStorage** are undefined.

An OpenGL ES 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 actual resolution in bits of each component of the allocated image can be queried with **GetRenderbufferParameteriv**.

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 `FRAMEBUFFER`. An `INVALID_OPERATION` error is generated if the current value of `FRAMEBUFFER_BINDING` is zero when **FramebufferRenderbuffer** is called. *attachment* should be set to one of the attachment points `COLOR_ATTACHMENT0`, `DEPTH_ATTACHMENT` or `STENCIL_ATTACHMENT`. *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 `INVALID_OPERATION` is generated. If *renderbuffer* is zero, then the value of *renderbuffertarget* is ignored.

If *renderbuffer* is not zero and if **FramebufferRenderbuffer** is successful, then the renderbuffer named *renderbuffer* will be used as the logical buffer identified by *attachment* of the framebuffer currently bound to *target*. The value of `FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE` for the specified attachment point is set to `RENDERBUFFER` and the value of `FRAMEBUFFER_ATTACHMENT_OBJECT_NAME` is set to *renderbuffer*. All other state values of the attachment point specified by *attachment* are set to their default values listed in table 6.23. No change is made to the state of the renderbuffer object and any previous attachment to the *attachment* logical buffer of the framebuffer object bound to framebuffer *target* is broken. If, on the other hand, the attachment is not successful, then no change is made to the state of either the renderbuffer object or the framebuffer object.

Calling **FramebufferRenderbuffer** with the *renderbuffer* name zero will detach the image, if any, identified by *attachment*, in the framebuffer currently bound to *target*. All state values of the attachment point specified by *attachment* in the object bound to *target* are set to their default values listed in table 6.23.

If a renderbuffer object is deleted while its image is attached to 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.

Attaching Texture Images to a Framebuffer

OpenGL ES supports copying the rendered contents of the framebuffer into the images of a texture object through the use of the routines **CopyTexImage2D** and **CopyTexSubImage2D**. Additionally, OpenGL ES 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 the command

```
void FramebufferTexture2D( enum target, enum attachment,
    enum textarget, uint texture, int level );
```

The *target* must be `FRAMEBUFFER`. An `INVALID_OPERATION` is generated if the current value of `FRAMEBUFFER_BINDING` is zero when FramebufferTexture2D is called. *attachment* must be one of the attachment points of the framebuffer.

If *texture* is zero, then *textarget* and *level* are ignored. If *texture* is not zero, then *texture* must either name an existing texture object with an target of *textarget*, or *texture* must name an existing cube map texture and *textarget* must be one of: `TEXTURE_CUBE_MAP_POSITIVE_X`, `TEXTURE_CUBE_MAP_POSITIVE_Y`, `TEXTURE_CUBE_MAP_POSITIVE_Z`, `TEXTURE_CUBE_MAP_NEGATIVE_X`, `TEXTURE_CUBE_MAP_NEGATIVE_Y`, or `TEXTURE_CUBE_MAP_NEGATIVE_Z`. Otherwise, `INVALID_OPERATION` is generated.

level specifies the mipmap level of the texture image to be attached to the framebuffer and must be 0. Otherwise, `INVALID_VALUE` is generated.

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`.

If *texture* is not zero, and if **FramebufferTexture2D** is successful, then the specified texture image will be used as the logical buffer identified by *attachment* of the framebuffer currently bound to *target*. The value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE for the specified attachment point is set to TEXTURE and the value of FRAMEBUFFER_ATTACHMENT_OBJECT_NAME is set to *texture*. Additionally, the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL for the named attachment point is set to *level*. If *texture* is a cubemap texture then the value of FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE the named attachment point is set to *textarget*. All other state values of the attachment point specified by *attachment* are set to their default values listed in table 6.23. No change is made to the state of the texture object, and any previous attachment to the *attachment* logical buffer of the framebuffer object bound to framebuffer *target* is broken. If, on the other hand, the attachment is not successful, then no change is made to the state of either the texture object or the framebuffer object.

Calling **FramebufferTexture2D** with *texture* name zero will detach the image identified by *attachment*, if any, in the framebuffer currently bound to *target*. All state values of the attachment point specified by *attachment* are set to their default values listed in table 6.23.

If a texture object is deleted while its image is attached to the currently bound framebuffer, then it is as if **FramebufferTexture2D** had been called, with a *texture* of 0, 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.

4.4.4 Feedback Loops Between Textures and the Framebuffer

A *feedback loop* may exist when a texture object is used as both the source and destination of a GL operation. When a feedback loop exists, undefined behavior results. This section describes *rendering feedback loops* (see section 3.7.7) and *texture copying feedback loops* (see section 3.7.2) in more detail.

Rendering Feedback Loops

The mechanisms for attaching textures to a framebuffer object do not prevent a two-dimensional texture level from being attached to the draw framebuffer while the same texture is bound to a texture unit. While this conditions holds, texturing operations accessing that image will produce undefined results, as described at the end of section 3.7.7. Conditions resulting in such undefined behavior are defined

in more detail below. Such undefined texturing operations are likely to leave the final results of fragment processing operations undefined, and should be avoided.

Special precautions need to be taken to avoid attaching a texture image to the currently bound framebuffer while the texture object is currently bound and enabled for texturing. Doing so could lead to the creation of a *rendering feedback loop* between the writing of pixels by OpenGL ES 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, but the values of fragments rendered while in this state will be undefined. The values of texture samples may be undefined as well, as described under "Rendering Feedback Loops" in section 3.7.7.

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 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 0 (the level zero array 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 of mipmap levels specified for the texture object T.

For the purposes of this discussion, it is *possible* to sample from the texture object T bound to texture unit U if the active fragment or vertex shader contains any instructions that might sample from the texture object T bound to U, even if those instructions might only be executed conditionally.

Texture Copying Feedback Loops

Similarly to rendering feedback loops, it is possible for a texture image to be attached to the read framebuffer while the same texture image is the destination of a **CopyTexImage2D** operation, as described under "Texture Copying Feedback Loops" in section 3.7.2. While this condition holds, a texture copying feedback loop between the writing of texels by the copying operation and the reading of those same texels when used as pixels in the read framebuffer may exist. In this scenario, the values of texels written by the copying operation will be undefined.

Specifically, the values of copied texels are undefined if all of the following conditions are true:

- an image from texture object T is attached to the currently bound framebuffer at attachment point A
- the selected read buffer is attachment point A
- T is bound to the texture target of a **CopyTexImage2D** operation
- the *level* argument of the copying operation selects the same image that is attached to A

4.4.5 Framebuffer Completeness

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. A framebuffer must be complete to effectively be used as the destination for OpenGL ES framebuffer rendering operations and the source for OpenGL ES framebuffer read operations.

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. Image internal formats are summarized in table 4.5. *Color-renderable* formats contain red, green, blue, and possibly alpha components; *depth-renderable* formats contain depth components; and *stencil-renderable* formats contain stencil components.

Formats not listed in table 4.5, including compressed internal formats. are not color-, depth-, or stencil-renderable, no matter which components they contain.

Sized Internal Format	Renderable Type	R bits	G bits	B bits	A bits	D bits	S bits
DEPTH_COMPONENT16	depth-renderable					16	
RGBA4	color-renderable	4	4	4	4		
RGB5_A1	color-renderable	5	5	5	1		
RGB565	color-renderable	5	6	5			
STENCIL_INDEX8	stencil-renderable						8

Table 4.5: Renderbuffer image formats, showing their renderable type (color-, depth-, or stencil-renderable) and the number of bits each format contains for color (R, G, B, A), depth (D), and stencil (S) components.

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* must be non-zero.
- If *attachment* is COLOR_ATTACHMENT0, 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.

Framebuffer Completeness

In this subsection, each rule is followed by an error enum in **bold**.

The framebuffer object *target* is said to be *framebuffer complete* if it is the window-system-provided framebuffer, or if all the following conditons are true:

- 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`

- All attached images have the same width and height.

  `FRAMEBUFFER_INCOMPLETE_DIMENSIONS`

- The combination of internal formats of the attached images does not violate an implementation-dependent set of restrictions.

  `FRAMEBUFFER_UNSUPPORTED`

The enum in bold after each clause of the framebuffer completeness rules specifies the return value of **CheckFramebufferStatus** that is generated when that clause is violated. If more than one clause is violated, it is implementation-dependent as to exactly which enum 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 **FramebufferTexture2D** or **FramebufferRenderbuffer**.
- Detaching an image from the framebuffer with **FramebufferTexture2D** or **FramebufferRenderbuffer**.
- Changing the width, height, or internal format of a texture image that is attached to the framebuffer by calling **TexImage2D**, **CopyTexImage2D** and **CompressedTexImage2D**.
- Changing the width, height, or 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.

Although OpenGL ES 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. There must exist, however, at least one combination of internal formats for which the framebuffer cannot be FRAMEBUFFER_UNSUPPORTED.

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 is 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*);

If *target* is not FRAMEBUFFER, INVALID_ENUM is generated. If **CheckFramebufferStatus** generates an error, 0 is returned.

Otherwise, an enum is returned that identifies whether or not the framebuffer bound to *target* is complete, and if not complete the enum identifies one of the rules of framebuffer completeness that is violated. If the framebuffer is complete, then FRAMEBUFFER_COMPLETE is returned.

Effects of Framebuffer Completeness on Framebuffer Operations

If the currently bound framebuffer is not framebuffer complete, then it is an error to attempt to use the framebuffer for writing or reading. This means that rendering commands such as **DrawArrays** and **DrawElements**, as well as commands that read the framebuffer such as **ReadPixels** and **CopyTexSubImage**, will generate the error INVALID_FRAMEBUFFER_OPERATION if called while the framebuffer is not framebuffer complete.

4.4.6 Effects of Framebuffer State on Framebuffer Dependent Values

The values of the state variables listed in table 6.20 (Implementation Dependant Pixel Depths) may change when a change is made to FRAMEBUFFER_BINDING, to the state of the currently bound framebuffer object, or to an image attached to the currently bound framebuffer object.

When FRAMEBUFFER_BINDING is zero, the values of the state variables listed in table 6.20 are implementation defined.

When 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 6.20 are undefined.

When FRAMEBUFFER_BINDING is non-zero and the currently bound framebuffer object is framebuffer complete, then the values of the state variables listed in table 6.20 are completely determined by FRAMEBUFFER_BINDING, the state of the currently bound framebuffer object, and the state of the images attached to the currently bound framebuffer object.

4.4.7 Mapping between Pixel and Element in Attached Image

When 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) correspond 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 value in the level zero array of that texture at the same coordinates.

Conversion to Framebuffer-Attachable Image Components

When an enabled color value is written to the framebuffer while FRAMEBUFFER_BINDING is non-zero, for each draw buffer the R, G, B, and A values are converted to internal components 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 by **ColorMask**, **DepthMask**, **StencilMask**, and **StencilMaskSeparate** are also effective.

4.4.8 Errors

The error INVALID_FRAMEBUFFER_OPERATION is generated if the value returned by **CheckFramebufferStatus** is not FRAMEBUFFER_COMPLETE, and any attempts to render to or read from the framebuffer are made.

The error INVALID_OPERATION is generated if **GetFramebufferAttachmentParameteriv** is called while the value of FRAMEBUFFER_BINDING is zero.

The error INVALID_OPERATION is generated if **FramebufferRenderbuffer** or **FramebufferTexture2D** is called while the value of FRAMEBUFFER_BINDING

is zero.

The error INVALID_OPERATION is generated if **RenderbufferStorage** is called while the value of RENDERBUFFER_BINDING is zero.

The error INVALID_VALUE is generated if **RenderbufferStorage** is called with a *width* or *height* that is greater than MAX_RENDERBUFFER_SIZE.

The error INVALID_ENUM is generated if **RenderbufferStorage** is called with an *internalformat* that is not among the list of supported color, depth or stencil formats.

The error INVALID_OPERATION is generated if **FramebufferRenderbuffer** is called and *renderbuffer* is not the name of a renderbuffer object.

The error INVALID_OPERATION is generated if **FramebufferTexture2D** is called and *texture* is not the name of a texture object.

The error INVALID_VALUE is generated if **FramebufferTexture2D** is called with a *level* that is less than zero.

The error INVALID_VALUE is generated if **FramebufferTexture2D** is called with a *level* that is greater than 0.

The error INVALID_ENUM is generated if **CheckFramebufferStatus** is called and *target* is not FRAMEBUFFER.

The error OUT_OF_MEMORY is generated if OpenGL ES is unable to create a data store of the required size when calling **RenderbufferStorage**.

The error INVALID_OPERATION is generated if **GenerateMipmap** is called with a *target* of TEXTURE_CUBE_MAP and the texture object currently bound to TEXTURE_CUBE_MAP is not cube complete.

Chapter 5

Special Functions

This chapter describes additional GL functionality that does not fit easily into any of the preceding chapters: flushing and finishing (used to synchronize the GL command stream), and hints.

5.1 Flush and Finish

The command

```
void Flush( void );
```

indicates that all commands that have previously been sent to the GL must complete in finite time.

The command

```
void Finish( void );
```

forces all previous GL commands to complete. **Finish** does not return until all effects from previously issued commands on GL client and server state and the framebuffer are fully realized.

5.2 Hints

Certain aspects of GL behavior, when there is room for variation, may be controlled with hints. A hint is specified using

```
void Hint( enum target, enum hint );
```

target is a symbolic constant indicating the behavior to be controlled, and *hint* is a symbolic constant indicating what type of behavior is desired. *target* must be `GENERATE_MIPMAP_HINT`, indicating the desired quality and performance of mipmap level generation with **GenerateMipmap**. *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.

The interpretation of hints is implementation-dependent. An implementation may ignore them entirely.

The initial value of all hints is `DONT_CARE`.

Chapter 6

State and State Requests

The state required to describe the GL machine is enumerated in section 6.2. Most state is set through the calls described in previous chapters, and can be queried using the calls described in section 6.1.

6.1 Querying GL State

6.1.1 Simple Queries

Much of the GL state is completely identified by symbolic constants. The values of these state variables can be obtained using a set of **Get** commands. There are four commands for obtaining simple state variables:

```
void GetBooleanv( enum value, boolean *data );
void GetIntegerv( enum value, int *data );
void GetFloatv( enum value, float *data );
```

The commands obtain boolean, integer, or floating-point 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. In addition

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

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

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 **DepthRangef** 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 the INT entry of Table 4.4; 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, and an integer value is coerced to floating-point.

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 **DepthRangef** parameters are returned in the order *n* followed by *f*.

Most texture state variables are qualified by the value of ACTIVE_TEXTURE to determine which server texture state vector is queried. Tables 6.2, 6.7, 6.9, and 6.18 indicate those state variables which are qualified by ACTIVE_TEXTURE during state queries. 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.

6.1.3 Enumerated Queries

Other commands exist to obtain state variables that are identified by a category (texture ID, buffer object name, etc.) as well as a symbolic constant. These are

The command

```
void GetTexParameter{if}v( enum target, enum value,
    T data );
```

returns information about *target*, which may be one of TEXTURE_2D or TEXTURE_CUBE_MAP, indicating the currently bound two-dimensional or cube map texture object. *value* is a symbolic value indicating which texture parameter is to be obtained. *value* must be one of the symbolic values in table 3.10.

The command

```
void GetBufferParameteriv( enum target, enum value,
    T data );
```

returns information about *target*, which may be one of ARRAY_BUFFER or ELEMENT_ARRAY_BUFFER, indicating the currently bound vertex array or element array buffer object. *value* is a symbolic value indicating which buffer object parameter is to be obtained, and must be one of the symbolic values in table 2.6.

The command

void **GetFramebufferAttachmentParameteriv**(enum *target*, enum *attachment*, enum *pname*, int **params*);

returns information about framebuffer objects. *target* must be FRAMEBUFFER. *attachment* must be one of the attachment points COLOR_ATTACHMENT0, DEPTH_ATTACHMENT, or STENCIL_ATTACHMENT. *pname* must be one of the following: FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE, FRAMEBUFFER_ATTACHMENT_OBJECT_NAME, FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL, or FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE.

If the framebuffer currently bound to *target* is zero, then INVALID_OPERATION is generated.

Upon successful return from **GetFramebufferAttachmentParameteriv**, if *pname* is FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE, then param will contain one of NONE, TEXTURE, or RENDERBUFFER, identifying the type of object which contains the attached image.

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, INVALID_ENUM 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 cubemap texture object which contains the attached image. Otherwise *params* will contain the value zero.

- Otherwise, INVALID_ENUM is generated.

If the value of FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE is NONE, then querying any other *pname* will generate INVALID_ENUM.

The command

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

returns information about renderbuffer objects. *target* must be RENDERBUFFER. *pname* must be one of the symbolic values in table 6.22 other than RENDERBUFFER_BINDING.

If the renderbuffer currently bound to *target* is zero, then INVALID_OPERATION is generated.

Upon successful return from **GetRenderbufferParameteriv**, if *pname* is RENDERBUFFER_WIDTH, RENDERBUFFER_HEIGHT, or RENDERBUFFER_INTERNAL_FORMAT, then *params* will contain the width in pixels, height in pixels, or internal format, respectively, of the image of the renderbuffer currently bound to *target*.

Upon successful return from **GetRenderbufferParameteriv**, 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, INVALID_ENUM is generated.

6.1.4 Texture Queries

The command

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

returns TRUE if *texture* is the name of a texture object. If *texture* is zero, or is a nonzero value that is not the name of a texture object, or if an error condition occurs, **IsTexture** returns FALSE. A name returned by **GenTextures**, but not yet bound, is not the name of a texture object.

6.1.5 String Queries

The command

```
ubyte *GetString( enum name );
```

returns a pointer to a static string describing some aspect of the current GL connection [1]. The possible values for *name* are 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 string is laid out as follows:

"OpenGL ES *N.M vendor-specific information*"

The SHADING_LANGUAGE_VERSION string is laid out as follows:

"OpenGL ES GLSL ES *N.M vendor-specific information*"

The version number is either of the form *major_number.minor_ number* or *major_number.minor_number.release_number*, where the numbers all have one or more digits. The *release_number* and vendor specific information are optional. However, if present, then they pertain to the server and their format and contents are implementation-dependent.

GetString returns the version number (returned in the VERSION string) and the extension names (returned in the EXTENSIONS string) that can be supported on the connection. Thus, if the client and server support different versions and/or extensions, a compatible version and list of extensions is returned.

6.1.6 Buffer Object Queries

The command

```
boolean IsBuffer( uint buffer );
```

returns TRUE if *buffer* is the name of an buffer object. If *buffer* is zero, or if *buffer* is a non-zero value that is not the name of an buffer object, **IsBuffer** returns FALSE.

[1] Applications making copies of these static strings should never use a fixed-length buffer, because the strings may grow unpredictably between releases, resulting in buffer overflow when copying. This is particularly true of the EXTENSIONS string, which has become extremely long in some GL implementations.

6.1.7 Framebuffer Object and Renderbuffer 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** returns `FALSE`.

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** returns `FALSE`.

6.1.8 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 a shader 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 sucessfully, 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.

If the value of SHADER_COMPILER is not TRUE, then the error INVALID_OPERATION is generated if *pname* is COMPILE_STATUS, INFO_LOG_LENGTH, or SHADER_SOURCE_LENGTH.

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 shader has been flagged for deletion and FALSE is returned otherwise. If *pname* is LINK_STATUS, TRUE is returned if the shader was last compiled sucessfully, 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.

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.

If the value of SHADER_COMPILER is not TRUE, then the error INVALID_-OPERATION is generated.

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**.

If the value of SHADER_COMPILER is not TRUE, then the error INVALID_OPERATION is generated.

The command

```
void GetShaderPrecisionFormat( enum shadertype,
    enum precisiontype, int *range, int *precision );
```

returns the range and precision for different numeric formats supported by the shader compiler. *shadertype* must be VERTEX_SHADER or FRAGMENT_SHADER. *precisiontype* must be one of LOW_FLOAT, MEDIUM_FLOAT, HIGH_FLOAT, LOW_INT, MEDIUM_INT or HIGH_INT. *range* points to an array of two integers in which encodings of the format's numeric range are returned. If min and max are the smallest and largest values representable in the format, then the values returned are defined to be

$$range[0] = \lfloor log_2(|min|) \rfloor$$

$$range[1] = \lfloor log_2(|max|) \rfloor$$

precision points to an integer in which the log_2 value of the number of bits of precision of the format is returned. If the smallest representable value greater than 1 is $1 + \epsilon$, then **precision* will contain $\lfloor -log_2(\epsilon) \rfloor$, and every value in the range

$$[-2^{range[0]}, 2^{range[1]}]$$

can be represented to at least one part in $2^{*precision}$. For example, an IEEE single-precision floating-point format would return $range[0] = 127$, $range[1] = 127$, and $*precision = 23$, while a 32-bit twos-complement integer format would return $range[0] = 31$, $range[1] = 30$, and $*precision = 0$.

The minimum required precision and range for formats corresponding to the different values of *precisiontype* are described in section 4.5 of the OpenGL ES Shading Language specification.

If high precision floating-point is not supported in fragment shaders, calling **GetShaderPrecisionFormat** with a *precisiontype* of HIGH_FLOAT will return zero for *range[0]*, *range[1]*, and **precision*.

If the value of SHADER_COMPILER is not TRUE, then the error INVALID_OPERATION is generated.

The commands

```
void GetVertexAttribfv( uint index, enum pname,
    float *params );
void GetVertexAttribiv( uint index, enum pname,
    int *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_ENABLED, VERTEX_ATTRIB_ARRAY_SIZE, VERTEX_ATTRIB_ARRAY_STRIDE, VERTEX_ATTRIB_ARRAY_TYPE, VERTEX_ATTRIB_ARRAY_NORMALIZED, VERTEX_ATTRIB_ARRAY_BUFFER_BINDING, or CURRENT_VERTEX_ATTRIB. Note that all the queries except CURRENT_VERTEX_ATTRIB return 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 and normalized flag are set by the command **VertexAttribPointer**. The query CURRENT_VERTEX_ATTRIB returns the current value for the generic attribute *index*.

The command

```
void GetVertexAttribPointerv( uint index, enum pname,
    void **pointer );
```

obtains the pointer named *pname* for vertex attribute numbered *index* and places the information in the array *pointer*. *pname* must be VERTEX_ATTRIB_ARRAY_POINTER. The INVALID_VALUE error is generated if *index* is greater than or equal to MAX_VERTEX_ATTRIBS.

The commands

```
void GetUniformfv( uint program, int location,
    float *params );
void GetUniformiv( uint program, int location,
    int *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.

6.2 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**, or **GetFloatv** 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**, and **GetFloatv**. State variables for which any other command is listed as the query command can be obtained only by using that command.

A type is also indicated for each variable. Table 6.1 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 clip planes, where only the selected clip plane is returned; and with textures, where only the selected texture or texture parameter is returned.

Type code	Explanation
B	Boolean
c	Character in a counted string
C	Color (floating-point R, G, B, and A values)
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
R_k	k-valued floating-point number
S	`NULL`-terminated string
Y	Pointer (data type unspecified)
$n \times type$	n copies of type $type$ ($n*$ indicates n is minimum)

Table 6.1: State Variable Types

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
VERTEX_ATTRIB_ARRAY_ENABLED	$8 * \times B$	**GetVertexAttrib**	*False*	Vertex attrib array enable	2.8
VERTEX_ATTRIB_ARRAY_SIZE	$8 * \times Z$	**GetVertexAttrib**	4	Vertex attrib array size	2.8
VERTEX_ATTRIB_ARRAY_STRIDE	$8 * \times Z^+$	**GetVertexAttrib**	0	Vertex attrib array stride	2.8
VERTEX_ATTRIB_ARRAY_TYPE	$8 * \times Z_4$	**GetVertexAttrib**	FLOAT	Vertex attrib array type	2.8
VERTEX_ATTRIB_ARRAY_NORMALIZED	$8 * \times B$	**GetVertexAttrib**	*False*	Vertex attrib array normalized	2.8
VERTEX_ATTRIB_ARRAY_POINTER	$8 * \times Y$	**GetVertex-AttribPointer**	NULL	Vertex attrib array pointer	2.8
ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	current buffer binding	2.9
ELEMENT_ARRAY_BUFFER_BINDING	Z^+	**GetIntegerv**	0	element array buffer binding	2.9.2
VERTEX_ATTRIB_ARRAY_BUFFER_BINDING	$8 * \times Z^+$	**GetVertexAttribiv**	0	Attribute array buffer binding	2.9

Table 6.2. Vertex Array Data

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
BUFFER_SIZE	$n \times Z^+$	**GetBufferParameteriv**	0	buffer data size	2.9
BUFFER_USAGE	$n \times Z_3$	**GetBufferParameteriv**	STATIC_DRAW	buffer usage pattern	2.9

Table 6.3. Buffer Object State

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
VIEWPORT	$4 \times Z$	**GetIntegerv**	see 2.12.1	Viewport origin & extent	2.12.1
DEPTH_RANGE	$2 \times R^{+}$	**GetFloatv**	0,1	Depth range near & far	2.12.1

Table 6.4. Transformation state

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
LINE_WIDTH	R^+	**GetFloatv**	1.0	Line width	3.4
CULL_FACE	B	**IsEnabled**	*False*	Polygon culling enabled	3.5.1
CULL_FACE_MODE	Z_3	**GetIntegerv**	BACK	Cull front/back facing polygons	3.5.1
FRONT_FACE	Z_2	**GetIntegerv**	CCW	Polygon frontface CW/CCW indicator	3.5.1
POLYGON_OFFSET_FACTOR	R	**GetFloatv**	0	Polygon offset factor	3.5.2
POLYGON_OFFSET_UNITS	R	**GetFloatv**	0	Polygon offset units	3.5.2
POLYGON_OFFSET_FILL	B	**IsEnabled**	*False*	Polygon offset enable	3.5.2

Table 6.5. Rasterization

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
SAMPLE_ALPHA_TO_COVERAGE	B	**IsEnabled**	*False*	Modify coverage from alpha	4.1.3
SAMPLE_COVERAGE	B	**IsEnabled**	*False*	Mask to modify coverage	4.1.3
SAMPLE_COVERAGE_VALUE	R^+	**GetFloatv**	1	Coverage mask value	4.1.3
SAMPLE_COVERAGE_INVERT	B	**GetBooleanv**	*False*	Invert coverage mask value	4.1.3

Table 6.6. Multisampling

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
TEXTURE_BINDING_2D	$8 * \times Z^+$	**GetIntegerv**	0	Texture object bound to `TEXTURE_2D`	3.7.13
TEXTURE_BINDING_CUBE_MAP	$8 * \times Z^+$	**GetIntegerv**	0	Texture object bound to `TEXTURE_CUBE_MAP`	3.7.12

Table 6.7. Textures (state per texture unit and binding point)

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
TEXTURE_MIN_FILTER	$n \times Z_6$	**GetTexParameter**	see 3.7	Texture minification function	3.7.7
TEXTURE_MAG_FILTER	$n \times Z_2$	**GetTexParameter**	see 3.7	Texture magnification function	3.7.8
TEXTURE_WRAP_S	$n \times Z_2$	**GetTexParameter**	REPEAT	Texcoord s wrap mode	3.7.6
TEXTURE_WRAP_T	$n \times Z_2$	**GetTexParameter**	REPEAT	Texcoord t wrap mode	3.7.6

Table 6.8. Textures (state per texture object)

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
ACTIVE_TEXTURE	Z_{8*}	**GetIntegerv**	TEXTURE0	Active texture unit selector	2.7

Table 6.9. Texture Environment and Generation

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
SCISSOR_TEST	B	**IsEnabled**	*False*	Scissoring enabled	4.1.2
SCISSOR_BOX	$4 \times Z$	**GetIntegerv**	see 4.1.2	Scissor box	4.1.2
STENCIL_TEST	B	**IsEnabled**	*False*	Stenciling enabled	4.1.4
STENCIL_FUNC	Z_8	**GetIntegerv**	ALWAYS	Front stencil function	4.1.4
STENCIL_VALUE_MASK	Z^+	**GetIntegerv**	1's	Front stencil mask	4.1.4
STENCIL_REF	Z^+	**GetIntegerv**	0	Front stencil reference value	4.1.4
STENCIL_FAIL	Z_8	**GetIntegerv**	KEEP	Front stencil fail action	4.1.4
STENCIL_PASS_DEPTH_FAIL	Z_8	**GetIntegerv**	KEEP	Front stencil depth buffer fail action	4.1.4
STENCIL_PASS_DEPTH_PASS	Z_8	**GetIntegerv**	KEEP	Front stencil depth buffer pass action	4.1.4
STENCIL_BACK_FUNC	Z_8	**GetIntegerv**	ALWAYS	Back stencil function	4.1.4
STENCIL_BACK_VALUE_MASK	Z^+	**GetIntegerv**	1's	Back stencil mask	4.1.4
STENCIL_BACK_REF	Z^+	**GetIntegerv**	0	Back stencil reference value	4.1.4
STENCIL_BACK_FAIL	Z_8	**GetIntegerv**	KEEP	Back stencil fail action	4.1.4
STENCIL_BACK_PASS_DEPTH_FAIL	Z_8	**GetIntegerv**	KEEP	Back stencil depth buffer fail action	4.1.4
STENCIL_BACK_PASS_DEPTH_PASS	Z_8	**GetIntegerv**	KEEP	Back stencil depth buffer pass action	4.1.4
DEPTH_TEST	B	**IsEnabled**	*False*	Depth buffer enabled	4.1.5
DEPTH_FUNC	Z_8	**GetIntegerv**	LESS	Depth buffer test function	4.1.5
BLEND	B	**IsEnabled**	*False*	Blending enabled	4.1.6
BLEND_SRC_RGB (v1.1:BLEND_SRC)	Z_{15}	**GetIntegerv**	ONE	Blending source RGB function	4.1.6
BLEND_SRC_ALPHA	Z_{15}	**GetIntegerv**	ONE	Blending source A function	4.1.6
BLEND_DST_RGB (v1.1:BLEND_DST)	Z_{14}	**GetIntegerv**	ZERO	Blending dest. RGB function	4.1.6

Table 6.10. Pixel Operations

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
COLOR_WRITEMASK	$4 \times B$	**GetBooleanv**	*True*	Color write enables; R, G, B, or A	4.2.2
DEPTH_WRITEMASK	B	**GetBooleanv**	*True*	Depth buffer enabled for writing	4.2.2
STENCIL_WRITEMASK	Z^+	**GetIntegerv**	1's	Front stencil buffer writemask	4.2.2
STENCIL_BACK_WRITEMASK	Z^+	**GetIntegerv**	1's	Back stencil buffer writemask	4.2.2
COLOR_CLEAR_VALUE	C	**GetFloatv**	0,0,0,0	Color buffer clear value (RGBA mode)	4.2.3
DEPTH_CLEAR_VALUE	R^+	**GetIntegerv**	1	Depth buffer clear value	4.2.3
STENCIL_CLEAR_VALUE	Z^+	**GetIntegerv**	0	Stencil clear value	4.2.3

Table 6.11. Framebuffer Control

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
UNPACK_ALIGNMENT	Z^+	**GetIntegerv**	4	Value of `UNPACK_ALIGNMENT`	3.6.1
PACK_ALIGNMENT	Z^+	**GetIntegerv**	4	Value of `PACK_ALIGNMENT`	4.3.1

Table 6.12. Pixels

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
SHADER_TYPE	Z_2	**GetShaderiv**	-	Type of shader (vertex or fragment)	2.10.1
DELETE_STATUS	B	**GetShaderiv**	*False*	Shader flagged for deletion	2.10.1
COMPILE_STATUS	B	**GetShaderiv**	*False*	Last compile succeeded	2.10.1
-	$0 * \times c$	**GetShaderInfoLog**	empty string	Info log for shader objects	6.1.8
INFO_LOG_LENGTH	Z^+	**GetShaderiv**	0	Length of info log	6.1.8
-	$0 * \times c$	**GetShaderSource**	empty string	Source code for a shader	2.10.1
SHADER_SOURCE_LENGTH	Z^+	**GetShaderiv**	0	Length of source code	6.1.8

Table 6.13. Shader Object State

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
CURRENT_PROGRAM	Z^+	**GetIntegerv**	0	Name of current program object	2.10.3
DELETE_STATUS	B	**GetProgramiv**	*False*	Program object deleted	2.10.3
LINK_STATUS	B	**GetProgramiv**	*False*	Last link attempt succeeded	2.10.3
VALIDATE_STATUS	B	**GetProgramiv**	*False*	Last validate attempt succeeded	2.10.3
ATTACHED_SHADERS	Z^+	**GetProgramiv**	0	Number of attached shader objects	6.1.8
-	$0 * \times Z$	**GetAttachedShaders**	empty	Shader objects attached	6.1.8
-	$0 * \times c$	**GetProgramInfoLog**	empty	Info log for program object	6.1.8
INFO_LOG_LENGTH	Z^+	**GetProgramiv**	0	Length of info log	2.10.4
ACTIVE_UNIFORMS	Z^+	**GetProgramiv**	0	Number of active uniforms	2.10.4
-	$0 * \times Z$	**GetUniformLocation**	–	Location of active uniforms	6.1.8
-	$0 * \times Z^+$	**GetActiveUniform**	–	Size of active uniform	2.10.4
-	$0 * \times Z^+$	**GetActiveUniform**	–	Type of active uniform	2.10.4
-	$0 * \times c$	**GetActiveUniform**	empty	Name of active uniform	2.10.4
	$512 * \times R$	**GetUniform**	0	Uniform value	2.10.4
ACTIVE_ATTRIBUTES	Z^+	**GetProgramiv**	0	Number of active attributes	2.10.4
-	$0 * \times Z$	**GetAttribLocation**	–	Location of active generic attribute	2.10.4
-	$0 * \times Z^+$	**GetActiveAttrib**	–	Size of active attribute	2.10.4
-	$0 * \times Z^+$	**GetActiveAttrib**	–	Type of active attribute	2.10.4
-	$0 * \times$ `char`	**GetActiveAttrib**	empty	Name of active attribute	2.10.4

Table 6.14. Program Object State

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
CURRENT_VERTEX_ATTRIB	$16 * \times R^4$	**GetVertexAttrib**	0,0,0,1	Generic vertex attribute	2.7

Table 6.15. Vertex Shader State

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
GENERATE_MIPMAP_HINT	Z_3	**GetIntegerv**	DONT_CARE	Mipmap generation hint	5.2

Table 6.16. Hints

Get value	Type	Get Cmnd	Minimum Value	Description	Sec.
SUBPIXEL_BITS	Z^+	**GetIntegerv**	4	Number of bits of subpixel precision in screen x_w and y_w	3
MAX_TEXTURE_SIZE	Z^+	**GetIntegerv**	64	Maximum texture image dimension	3.7.1
MAX_CUBE_MAP_TEXTURE_SIZE	Z^+	**GetIntegerv**	16	Maximum cube map texture image dimension	3.7.1
MAX_VIEWPORT_DIMS	$2 \times Z^+$	**GetIntegerv**	see 2.12.1	Maximum viewport dimensions	2.12.1
ALIASED_POINT_SIZE_RANGE	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of aliased point sizes	3.3
ALIASED_LINE_WIDTH_RANGE	$2 \times R^+$	**GetFloatv**	1,1	Range (lo to hi) of aliased line widths	3.4
SAMPLE_BUFFERS	Z^+	**GetIntegerv**	0	Number of multisample buffers	3.2
SAMPLES	Z^+	**GetIntegerv**	0	Coverage mask size	3.2
COMPRESSED_TEXTURE_FORMATS	$0 * \times Z_{0*}$	**GetIntegerv**	-	Enumerated compressed texture formats	3.7.3
NUM_COMPRESSED_TEXTURE_FORMATS	Z	**GetIntegerv**	0	Number of enumerated compressed texture formats	3.7.3

Table 6.17. Implementation Dependent Values

Get value	Type	Get Cmnd	Minimum Value	Description	Sec.
SHADER_BINARY_FORMATS	$nf \times Z$	**GetIntegerv**	-	Enumerated shader binary formats	2.10.2
NUM_SHADER_BINARY_FORMATS	Z	**GetIntegerv**	0	Number of shader binary formats	2.10.2
SHADER_COMPILER	B	**GetBooleanv**	-	Shader compiler supported	2.10
–	$6 \times 2 \times Z^+$	**GetShader-PrecisionFormat**	-	Fragment shader data type ranges	6.1.8
–	$6 \times Z^+$	**GetShader-PrecisionFormat**	-	Fragment shader data type precisions	6.1.8
EXTENSIONS	S	**GetString**	–	Supported extensions	6.1.5
RENDERER	S	**GetString**	–	Renderer string	6.1.5
SHADING_LANGUAGE_VERSION	S	**GetString**	–	Shading Language version supported	6.1.5
VENDOR	S	**GetString**	–	Vendor string	6.1.5
VERSION	S	**GetString**	–	OpenGL version supported	6.1.5

Table 6.18. Implementation Dependent Values (cont.)

Get value	Type	Get Cmnd	Minimum Value	Description	Sec.
MAX_VERTEX_ATTRIBS	Z^+	**GetIntegerv**	8	Number of active vertex attributes	2.7
MAX_VERTEX_UNIFORM_VECTORS	Z^+	**GetIntegerv**	128	Number of vectors for vertex shader uniform variables	2.10.4
MAX_VARYING_VECTORS	Z^+	**GetIntegerv**	8	Number of vectors for varying variables	2.10.4
MAX_COMBINED_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	8	Total number of texture units accessible by the GL	2.10.5
MAX_VERTEX_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	0	Number of texture image units accessible by a vertex shader	2.10.5
MAX_TEXTURE_IMAGE_UNITS	Z^+	**GetIntegerv**	8	Number of texture image units accessible by fragment processing	2.10.5
MAX_FRAGMENT_UNIFORM_VECTORS	Z^+	**GetIntegerv**	16	Number of vectors for frag. shader uniform variables	3.8.1
MAX_RENDERBUFFER_SIZE	Z^+	**GetIntegerv**	1	Maximum renderbuffer size	4.4.3

Table 6.19. Implementation Dependent Values (cont.)

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
x_BITS	Z^+	**GetIntegerv**	-	Number of bits in x color buffer component; x is one of RED, GREEN, BLUE, or ALPHA	4
DEPTH_BITS	Z^+	**GetIntegerv**	-	Number of depth buffer planes	4
STENCIL_BITS	Z^+	**GetIntegerv**	-	Number of stencil planes	4
IMPLEMENTATION_COLOR_READ_TYPE	Z^+	**GetIntegerv**	-	Implementation preferred pixel *type*	4.3.1
IMPLEMENTATION_COLOR_READ_FORMAT	Z^+	**GetIntegerv**	-	Implementation preferred pixel *format*	4.3.1

Table 6.20. Implementation Dependent Pixel Depths

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
-	$n \times Z_8$	**GetError**	NO_ERROR	Current error code(s)	2.5
-	$n \times B$	–	*False*	True if there is a corresponding error	2.5

Table 6.21. Miscellaneous

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
RENDERBUFFER_BINDING	Z^+	**GetIntegerv**	0	Renderbuffer binding	4.4.3
RENDERBUFFER_WIDTH	Z^+	**GetRenderbufferParameteriv**	0	Renderbuffer width	4.4.3
RENDERBUFFER_HEIGHT	Z^+	**GetRenderbufferParameteriv**	0	Renderbuffer height	4.4.3
RENDERBUFFER_INTERNAL_FORMAT	Z^+	**GetRenderbufferParameteriv**	RGBA4	Renderbuffer internal format	4.4.3
RENDERBUFFER_RED_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Renderbuffer red size	4.4.3
RENDERBUFFER_GREEN_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Renderbuffer green size	4.4.3
RENDERBUFFER_BLUE_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Renderbuffer blue size	4.4.3
RENDERBUFFER_ALPHA_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Renderbuffer alpha size	4.4.3
RENDERBUFFER_DEPTH_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Renderbuffer depth size	4.4.3
RENDERBUFFER_STENCIL_SIZE	Z^+	**GetRenderbufferParameteriv**	0	Renderbuffer stencil size	4.4.3

Table 6.22. Renderbuffer State

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
FRAMEBUFFER_BINDING	Z^+	**GetIntegerv**	0	Framebuffer binding	4.4.2
FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE	$n \times Z_3$	**GetFramebuffer-Attachment-Parameteriv**	NONE	Framebuffer object type	4.4.2
FRAMEBUFFER_ATTACHMENT_OBJECT_NAME	$n \times Z^+$	**GetFramebuffer-Attachment-Parameteriv**	0	Framebuffer object name	4.4.2
FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL	$n \times Z^+$	**GetFramebuffer-Attachment-Parameteriv**	0	Framebuffer texture level	4.4.2
FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE	$n \times Z_6$	**GetFramebuffer-Attachment-Parameteriv**	+ve X face	Framebuffer texture cubemap face	4.4.2

Table 6.23. Framebuffer State

Appendix A

Invariance

The OpenGL ES 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 ES .

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)*
- *Scissor parameters (other than enable)*
- *Writemasks (color, depth, stencil)*

- *Clear values (color, depth, stencil)*

Strongly suggested:

- *Stencil parameters (other than enable)*
- *Depth test parameters (other than enable)*
- *Blend parameters (other than enable)*
- *Pixel storage*
- *Polygon offset parameters (other than enables, and except as they affect the depth values of fragments)*

Corollary 1 *Fragment generation is invariant with respect to the state values marked with • in Rule 2.*

Rule 3 *The arithmetic of each per-fragment operation is invariant except with respect to parameters that directly control it (the parameters that control the depth test, for instance, are the depth test enable and the depth comparison function).*

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 ES 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 ES state values may change subtly when renderers are swapped. This is the type of state value change that Rule 1 seeks to avoid.

Appendix B

Corollaries

The following observations are derived from the body and the other appendixes of the specification. Absence of an observation from this list in no way impugns its veracity.

1. The error semantics of upward compatible OpenGL ES revisions may change. Otherwise, only additions can be made to upward compatible revisions.

2. GL query commands are not required to satisfy the semantics of the **Flush** or the **Finish** commands. All that is required is that the queried state be consistent with complete execution of all previously executed GL commands.

3. Application specified point size and line width must be returned as specified when queried. Implementation-dependent clamping affects the values only while they are in use.

4. The mask specified as the third argument to **StencilFunc** affects the operands of the stencil comparison function, but has no direct effect on the update of the stencil buffer. The mask specified by **StencilMask** has no effect on the stencil comparison function; it limits the effect of the update of the stencil buffer.

5. There is no atomicity requirement for OpenGL ES rendering commands, even at the fragment level.

6. Because rasterization of non-antialiased polygons is point sampled, polygons that have no area generate no fragments when they are rasterized, and the fragments generated by the rasterization of "narrow" polygons may not form a continuous array.

7. The GL does not force left- or right-handedness on any of its coordinate systems,

8. (No pixel dropouts or duplicates.) Let two polygons share an identical edge (that is, there exist vertices A and B of an edge of one polygon, and vertices C and D of an edge of the other polygon, and the coordinates of vertex A (resp. B) are identical to those of vertex C (resp. D), and the state of the the coordinate transfomations is identical when A, B, C, and D are specified). Then, when the fragments produced by rasterization of both polygons are taken together, each fragment intersecting the interior of the shared edge is produced exactly once.

9. Dithering algorithms may be different for different components. In particular, alpha may be dithered differently from red, green, or blue, and an implementation may choose to not dither alpha at all.

Appendix C

Deleting Shared Objects

This section clarifies how object deletion works for texture, buffer, framebuffer and renderbuffer objects. This section will only refer to texture objects but the same rule applies to buffer, renderbuffer and framebuffer objects.

The OpenGL 2.0 specification states that *after a texture object is deleted, it has no contents or dimensionality, and its name is again unused*. However, the EGL specification states that *all modifications to shared context state as a result of executing* **glBindTexture** *are atomic. Also, a texture object will not be deleted until it is no longer bound to any rendering context*.

An ambiguity arises because the OpenGL ES specification indicates that texture names are immediately unused after deletion, but the EGL specification indicates deletion does not actually occur until the texture is unbound from all contexts. Reading only the OpenGL and OpenGL ES specifications may lead one to conclude that a call to **DeleteTextures** marks a name as *unused* immediately, regardless of whether a texture object is bound in other contexts, but reading the EGL specification in conjunction with the OpenGL ES specification may lead one to believe that the name remains *used* until the texture is explicitly unbound in all contexts.

In the remainder of appendix C, it is understood that commands causing state changes in a shared object must complete before the described consequences of those state changes in another context hold true.

C.1 Effect of Shared Object Deletion on Object Namespace

After **DeleteTexture** is called, the object names are immediately marked unused. Note that the actual underlying object state and data are retained (i.e. the object

state is *orphaned*) and the object remains bound to all contexts until that object is explicitly unbound by a given context, or the context itself calls **DeleteTexture** which does an implicit Bind to object 0.

Let us review the effect on **IsTexture** in the following scenario:

1. Context 1 binds texture T to the 2D texture target
2. Context 2 binds texture T to the 2D texture target
3. Context 1 deletes texture T
4. Context 1 (or context 2) asks IsTexture(T)

IsTexture will return `FALSE`.

The *used* vs. *unused* state of the texture name T affects whether or not a new object is created on a bind of texture T. To see this, consider another scenario:

1. Context 1 binds texture T to the 2D texture target
2. Context 2 binds texture T to the 2D texture target
3. Context 1 deletes texture T
4. Context 2 attempts to rebind to texture T to the 2D texture target

When context 2 attempts to rebind to texture T in step 4, context 2 successfully creates and binds a **new** uninitialized 2D texture with the name T.

The choice of behaviors also affects the generation of OpenGL ES errors. The OpenGL ES specification states that it is an error to bind a texture to a target if that texture has already been bound to a target of a different type. So, consider the following scenario:

1. Context 1 binds texture T as a 2D texture
2. Context 2 binds texture T as a 2D texture
3. Context 1 deletes texture T
4. Context 1 attempts to rebind to texture T as a 3D texture

Context 1 successfully binds texture T to the 3D texture target creating a new uninitialized 3D texture in the process. Additionally, after step 4, if context 2 were to query its current texture bindings, it would find out that it is bound to a 2D texture named T, even though the texture named T is now a 3D texture. Further, if context 2 attempted to *rebind* texture T to the 2D target, then it would get an error.

C.2 Sharing objects across multiple OpenGL ES contexts

This section defines some of the behavior of OpenGL ES objects which can be shared by multiple contexts. Objects which can be shared in this manner include:

- textures
- buffer objects
- renderbuffers
- framebuffers
- shader and program objects

Traditionally, the specification of shared object and multi-context behavior was left to the window-system. However, this section describes the portions of behavior of shared objects that apply to all OpenGL ES implementations, regardless of window-system.

C.2.1 Updates to the state of shared objects

When multiple contexts are bound to a shared object, such as a texture, vertex buffer object etc., care should be taken to ensure that multiple contexts do not change the state of the shared object simultaneously. Otherwise, it is undefined which set of state changes made by the various contexts will take precedence.

Further, even if only one context at a time changes the state of the shared object, it is possible that the other context's might not observe the changed state until the next time those contexts bind the shared object again.

Finally, it is guaranteed that outstanding changes to the state of a shared object must be observable by any context which binds that object.

In other words, if context A and context B, are bound to shared object O, and context A modifies the state of shared object O, then it is possible that context B will still be using an *out of date* version of object O that does not reflect the state changes until context B rebinds object O. Further, it is guaranteed that the state changes made by context A will be observable by context B when context B binds object O.

C.2.2 The effect of shared object deletion on object namespace

If multiple contexts are bound to a shared object, such as a texture, vertex buffer object etc., then care should be taken when deleting that shared object. This is

because after deletion of a shared object bound to multiple contexts the name of the deleted object is immediately marked unused. This in turn means that the name can immediately be used by any context to create a new object with that name and the name might be returned by OpenGL ES routines which generate ranges of object names. However, the underlying object state and data may still be *in use* by contexts other than the context which deleted the object. These other contexts might continue using the underlying object, and these other contexts might still contain state which identifies the named object as being currently bound, until those other contexts make another attempt to bind the object with that name. Since the name is immediately unused, attempts to bind an object of that name by any context will create a new object with the specified name.

Appendix D

Version 2.0

OpenGL ES 2.0 is **not** upward compatible with prior versions (OpenGL ES 1.0 and 1.1). It introduces programmable vertex and fragment shaders, but removes the corresponding fixed-function pipeline functionality.

As of version 2.0.22, OpenGL ES 2.0 includes this *Full Specification* document, which is the authoritative definition of the API.

OpenGL ES 2.0 also includes a Difference Specification document written relative to the OpenGL 2.0 Specification. The Difference Specification is no longer authoritative, but is maintained as a reference for those familiar with desktop OpenGL, and to summarize the changes between the two APIs.

Appendix E

OES Extensions

OpenGL ES extensions that have been approved by the Khronos OpenGL ES working group are described in this chapter. These extensions are not required to be supported by a conformant OpenGL ES 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, OES extensions are not integrated into the core language; instead, they are made available online in the OpenGL ES Extension Registry. Extensions are documented as changes to the Specification. The Registry is available on the World Wide Web at URL

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

E.1 Naming Conventions

To distinguish OES extensions from core OpenGL ES features and from vendor specific extensions, the following naming conventions are used:

- A unique *name string* of the form "`GL_OES_`*`name`*" is associated with each extension. If the extension is supported by an implementation, this string will be present in the `EXTENSIONS` string.
- All functions defined by the extension will have names of the form ***Function*OES**
- All enumerants defined by the extension will have names of the form *`NAME`*`_OES`.

E.2 Promoting Extensions to Core Features

`OES` extensions can be *promoted* to required core features in later revisions of OpenGL ES . 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 **OES** affix removed.

OpenGL ES implementations of such later revisions should continue to export the name strings of promoted extensions in the `EXTENSIONS` string, and continue to support the **OES**-affixed versions of functions and enumerants as a transition aid.

Appendix F

Packaging and Acknowledgements

F.1 Header Files and Libraries

The Khronos Implementer's Guidelines, a separate document linked from the Khronos Extension Registry at

https://www.khronos.org/registry/

describes recommended and required practice for implementing OpenGL ES , including names of header files and libraries making up the implementation, and links to standard versions of the header files defining interfaces for the core OpenGL ES API (`gl2.h` and `gl2platform.h`) as well as a separate header (`gl2ext.h`) defining interfaces for Khronos-approved and vendor extensions.

Preprocessor tokens `GL_ES_VERSION_n_m`, where `n` and `m` are the major and minor version numbers as described in section 6.1.5, are included in `gl2.h`. These tokens indicate the OpenGL ES versions supported at compile-time.

F.2 Acknowledgements

The OpenGL ES 2.0 specification is the result of the contributions of many people, representing a cross section of the desktop, hand-held, and embedded computer industry. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:

Aaftab Munshi, ATI

Akira Uesaki, Panasonic

Aleksandra Krstic, Qualcomm

Andy Methley, Panasonic

Axel Mamode, Sony Computer Entertainment

Barthold Lichtenbelt, 3Dlabs

Benji Bowman, Imagination Technologies

Bill Marshall, Alt Software

Borgar Ljosland, Falanx

Brian Murray, Freescale

Chris Grimm, ATI

Daniel Rice, Sun

David Garcia, AMD

Ed Plowman, ARM

Edvard Sorgard, Falanx

Eisaku Ohbuch, DMP

Eric Fausett, DMP

Gary King, Nvidia

Gordon Grigor, ATI

Graham Connor, Imagination Technologies

Hans-Martin Will, Vincent

Hiroyasu Negishi, Mitsubishi

James McCarthy, Imagination Technologies

Jasin Bushnaief, Hybrid

Jitaek Lim, Samsung

John Howson, Imagination Technologies

John Kessenich, 3Dlabs

Jacob Ström, Ericsson

Jani Vaarala, Nokia

Jarkko Kemppainen, Nokia

John Boal, Alt Software

John Jarvis, Alt Software

Jon Leech, Silicon Graphics / Independent

Joonas Itaranta, Nokia

Jorn Nystad, Falanx

Justin Radeka, Falanx

Kari Pulli, Nokia

Katzutaka Nishio, Panasonic

Kee Chang Lee, Samsung

Keisuke Kirii, DMP

Lane Roberts, Symbian

Mario Blazevic, Falanx

Mark Callow, HI

Max Kazakov, DMP

Neil Trevett, 3Dlabs

Nicolas Thibieroz, Imagination Technologies

Petri Kero, Hybrid

Petri Nordlund, Bitboys

Phil Huxley, Tao Group

Robin Green, Sony Computer Entertainment

Remi Arnaud, Sony Computer Entertainment

Robert Simpson, Bitboys

Stanley Kao, HI

Stefan von Cavallar, Symbian

Steve Lee, SIS

Tero Pihlajakoski, Nokia

Tero Sarkinnen, Futuremark

Timo Suoranta, Futuremark

Thomas Tannert, Silicon Graphics

Tom McReynolds, Nvidia

Tom Olson, Texas Instruments

Tomi Aarnio, Nokia

Ville Miettinen, Hybrid Graphics

Woo Sedo Kim, LG Electronics

Yong Moo Kim, LG Electronics

Yoshihiko Kuwahara, DMP

Yoshiyuki Kato, Mitsubishi

Young Seok Kim, ETRI

Yukitaka Takemuta, DMP

F.3 Document History

F.3.1 Version 2.0.24, updated 2009/04/22

- Use "level zero" terminology consistently, replacing "level 0", "zero level", and "base level", in sections 3.7.1, 3.7.10, 4.4.4, and 4.4.7 (bug 4406).
- Replace redundant language in section 3.7.7 with a reference to section 4.4.4 (bug 4406).

F.3.2 Version 2.0.24, draft of 2009/04/01

- Removed `INVALID_OPERATION` error from the end of **ShaderBinary** specification in section 2.10.2 (bug 3673).
- Added forward references from vertex shader "Texture Access" language in section 2.10.5, and from completeness language in section 3.7.10, to fragment shader "Texture Access" language in section 3.8.2. Replaced reference to enabled cube map textures in section 3.7.5 and removed reference to texture mapping being enabled in section 3.7.13 (bug 3499).
- Dropped language about null textures from section 3.7.1; such a texture is by definition incomplete and therefore will be sampled as described in section 3.8.2 (bugs 3499,4176)
- Corrected references and descriptions in figure 3.6 and section 3.7.7 to account for NPOT textures (bug 4405).
- Updated completeness requirements in section 3.7.10 to include matching format and type as well as internal format (bug 3770).

- Updated sampling language in section 3.8.2 to not require texture completeness when non-mipmapped access to a two-dimensional texture is being done (bug 4282).
- Updated rendering feedback loop language, and added texture copying feedback loop language in section 4.4.4. Updated sections 3.7.2 and 3.7.7 as well, all corresponding to the language in the GL 3.1 Specification (bug 4406).
- Removed "one or more attachment points in" clauses in discussions of deleting a texture or renderbuffer while it's attached to an FBO with **FramebufferRenderbuffer** or **FramebufferTexture2D**, in section 4.4.3 (bug 3693).
- Enumerated valid **GetFramebufferAttachmentParameteriv** parameters in section 6.1.3 (bug 4408).
- Added errors in section 6.1.8 for **GetShaderInfoLog** and **GetShaderiv** when `SHADER_COMPILER` is `FALSE` (bug 3753).

F.3.3 Version 2.0.23, updated 2008/08/27

- Removed no-longer-relevant SGI copyright.
- Bump release number to 2.0.23 for public release.
- Flip sign of $log_2(\epsilon)$ in computation of *precision* for **GetShaderPrecisionFormat** in section 6.1.8 (bug 3667).

F.3.4 Version 2.0.22, updated 2008/08/06

- Remove `BUFFER_ACCESS` and `BUFFER_MAPPED` state from tables 2.5 and 6.3, since they cannot be changed (bug 3449, also much older WG minutes).
- Minor changes to prototypes and error conditions of **ShaderBinary** in section 2.10.2 (bug 3673).
- Removed luminance color buffer formats from **CopyTexImage** conversion table 3.9 (bug 3695).
- Minor fixes to prototypes and error conditions of framebuffer object commands including **BindFramebuffer**, **GenFramebuffers**, **GetFramebufferAttachmentParameteriv**, and **GetRenderbufferAttachmentParameteriv** in sections 4.4.1, 4.4.3, 4.4.6, 4.4.8, and 6.1.3. Change initial value of `RENDERBUFFER_INTERNAL_FORMAT` to `RGBA4` in table 6.22 (bug 3693).

- Specify error behavior when querying a NONE framebuffer attachment for a parameter other than the attachment's type in section 6.1.3.
- Clarify meaning of returned values from **GetShaderPrecisionFormat** in section 6.1.8 (bug 3667).
- Remove legacy attribute group column from state tables in section 6.2 (bug 3694).
- Added new implementation-dependent state table 6.19 to prevent tables overflowing the page width.

F.3.5 Version 2.0.22, updated 2008/07/17

- Remove references to enabling/disabling texture units from sections 3.7 and 3.7.10, and move language about as-if disabled behavior to section 3.8.2. This language may need to be replicated or referred to from the corresponding vertex shader language in section 2.10.5 as well (bug 3499).
- Removed leftover references to alpha test in figure 4.1 and section 4.1.8 (bug 3476).
- Define error for unsupported **ReadPixels** format/type combinations in section 4.3.1, disallow LUMINANCE as an implementation-dependent read format, remove language talking about luminance formats from **ReadPixels** (bug 3637).
- Rewrote description of **GetShaderPrecisionFormat** in section 6.1.8 to clarify meaning of range and precision values returned (bug 3667).
- Removed leftover references to fixed-function matrix and light state in section 6.2 (bug 3413).

F.3.6 Version 2.0.22, draft of 2008/04/30

- Updated figures 2.2, 2.4, and 3.1.
- Added appendix D, briefly summarizing OpenGL ES 2.0.

F.3.7 Version 2.0.22, draft of 2008/04/24

- Moved description of **GetShaderPrecisionFormat** from section 2.10.1 to 6.1.8 and removed FRAGMENT_PRECISION_HIGH query (bug 3359). Added state table entries for shader range and precision to table 6.18.

F.3.8 Version 2.0.22, draft of 2008/04/08

- Added boolean state `FRAGMENT_PRECISION_HIGH` in section 2.10.1 and table 6.18, indicating whether or not `highp` floating-point is supported in fragment shaders, and defined behavior of **GetShaderPrecisionFormat** when it is not supported (bug 3296).
- Moved section 3.2 up one logical level (the surrounding section was removed earlier in editing) (bug 3277).
- Removed incorrect `INVALID_VALUE` error for **AttachShader** in section 2.10.3 (bug 3186).
- Require that programs contain both vertex and fragment shaders, and fail linking otherwise, in sections 2.10, 2.10.3, and 3.8 (bug 3038).
- Removed leftover reference to fixed-point version of **TexParameter** in section 3.7.4 (bug 3187).
- Added table 4.5 describing valid renderbuffer image formats (bug 3070).
- Fixed typo in description of **GetShaderInfoLog** (section 6.1.8) (bug 3225).
- Fixed several small typos and errors in the state tables (bug 3195).
- Added a note that commands causing state changes to shared objects must complete before the other language in appendix C holds true (bug 3297).
- Changed section F.1 to point to the Khronos Implementer's Guidelines instead of duplicating header information here (bug 3184).

F.3.9 Version 2.0.22, draft of 2008/03/12

- Generalize references to vertex attributes in section 2.1 (bug 2866).
- Generalize description of per-fragment operations in section 2.4 (bug 2866).
- Clarify that error behavior may be changed by GL extensions in section 2.5 (bug 2866).
- Note that a program object must contain both vertex and fragment shaders in section 2.10 (bug 2866).
- Note in section 2.10.1 that when a string length specified to **ShaderSource** is negative, only its sign matters, not its value (bug 2866).

- Clarified the meaning of *range* and *precision* in **GetShaderPrecisionFormat** (section 2.10.1), including documenting that *range* is a two-element array (bug 2866). I'm not entirely clear on what these mean myself - please double-check.
- Removed forward reference to "the limit" in the discussion of vertex attributes in section 2.10.4 (bug 2866).
- Changed `MAX_VERTEX_UNIFORM_COMPONENTS` to `MAX_VERTEX_UNIFORM_VECTORS` in section 2.10.4, and changed `MAX_VARYING_FLOATS` to `MAX_VARYING_VECTORS` in section 2.10.4 (bug 2866).
- Removed dangling reference to fixed-function processing in section 2.10.5 (bug 2866).
- Note that an empty program will always fail validation (per conformance tests) in section 2.10.5 (bug 3038).
- Refer to "disabled" textures as well as incomplete textures in sections 2.10.5 and 3.8.2 (bug 3038).
- Fixed typos in sections 2.9, 2.9.1, 2.9.2, 2.10, 2.10.4, and 2.10.6 (bug 2866).
- Changed "between" to "outside" in discussion of line clipping (section 2.13) (bug 2866).
- Fixed constraints on texture mipmap level dimensions in section 3.7.1 (bug 2891).
- Added an `INVALID_VALUE` error to **TexImage2D** in section 3.7.1 when specifying a non-power-of-two image for a level other than zero, and an `INVALID_OPERATION` error to **GenerateMipmaps** in section 3.7.11 when called for a texture whose level zero array dimensions are non-power-of-two (bug 2807).
- Proposed that **GenerateMipmap** on a compressed texture image generate an error (section 3.7.11). This is based off of Remi's comments in bug 2893, but I'm not sure it's the right resolution - what is existing practice? The group should make a call on this.
- Removed most references to initial values of texture object state from section 3.7.12, as none of this state is queriable in GLES anyway. All that's relevant is that enough of it be defined to ensure the initial texture is null for enabling / completeness purposes (bug 3105). Could instead restore the texture object state table with no query commands, though.

- Dropped _OES suffix from IMPLEMENTATION_COLOR_READ_FORMAT and IMPLEMENTATION_COLOR_READ_TYPE in section 4.3.1 (bug 2884).
- Restored INT entry to reversed component conversion table 4.4, since this is referred to from section 6.1.2 (bug 3082).
- Noted the meaning of RGB565 in section 4.4.5, since we do not have a sized internal format table to refer to. Actually this section need to be cleaned up further for this reason (bug 3070).
- Removed INVALID_OPERATION error when querying CURRENT_VERTEX_ATTRIB for attribute zero (section 6.1.8) (bug 3038).
- Added missing state tables 6.13, 6.14, and 6.15 describing shader object, program object, and vertex shader state (bug 2886).
- Corrected names and descriptions of framebuffer attachment state and query commands in table 6.23 (bug 3038). Unfortunately there's really no way to cram in the full cubemap face name and still make the table fit on the page.
- Added shader and program objects to list of shared object types in appendix C.2 (bug 2885).

F.3.10 Version 2.0.22, draft of 2008/01/20

Fixes from David Garcia (**https://cvs.khronos.org/bugzilla/show_bug.cgi?id=2813**):

- Removed redundancy of "point" and "point sprite" in sections 2.6, 2.6.1, and 3.7.7 - all points are now point sprites and the term "point sprite" is no longer used.
- Changed description of program object contents in section 2.10, since they cannot contain multiple vertex shaders or multiple fragment shaders.
- Removed references to non-multisample antialiasing in chapter 3, including point/line antialiasing in the fourth paragraph of the chapter; all of the old "Antialiasing" and "Antialiasing Application" sections 3.2 and 3.9; and the description of mixing multisample and smooth rendering in the third paragraph of section 3.2.
- Removed references to shading fragments from pixel rectangles or bitmaps in section 3.8.

- Changed possible values of `gl_FrontFacing` in section 3.8.2 from `TRUE` / `FALSE` to `true` / `false`, since those are the GLSL values.

Other changes:

- Bumped release number on difference and full specs to 22.
- Mandated that **GetActiveUniform** for an active uniform array variable will always return the name of the uniform array appended with `"[0]"`, in section 2.10.4 and in section 2.15.3 of the difference specification (bug 1832).
- Remove $_c$ notation in table 4.1, and fixed blend equations for `FUNC_REVERSE_SUBTRACT`, per WG discussion.
- Noted that all varying attributes are interpolated perspective-correct in sections 3.4.1 and 3.5.1 of the difference specification (bug 2508).

F.3.11 Version 2.0.21, draft of 2008/01/11

Fixes from Ben Bowman (**https://cvs.khronos.org/bugzilla/show_bug.cgi?id=2807**):

- Noted that only a single vertex shader object can be attached to a program object in section 2.10.
- Dropped reference to a program object containing only a vertex shader as an example of sampler non-determinism at link time, from the end of the "Samplers" subsection of section 2.10.4.
- Dropped validation failures due to interactions with fixed-function state from the "Validation" subsection of section 2.10.5.
- Changed references to `gl_FragData[n]` to `gl_FragData[0]` in section 3.8.2, since OpenGL ES only supports a single fragment color.

Fixes from Georg Kolling (**https://cvs.khronos.org/bugzilla/show_bug.cgi?id=2770#c2**):

- Removed corollary 2 in appendix A.3, since there is no fixed-function state remaining for generated window coordinates to be invariant with respect to. Changed example for invariance rule 3 to use the depth test instead of the no longer existent alpha test. Removed corollary 3 since neither **DrawBuffers** nor multiple color buffer attachments are supported.

Other changes:

- Removed unused `clampx` type from table 2.2.
- Added `INVALID_OPERATION` errors in sections 2.10.1, 2.10.2, and 6.1.8 when calling unsupported shader compiler or shader binary commands.
- Simplified corollary 7 in appendix B to just say that OpenGL ES does not force left- or right-handedness on any of its coordinate systems, and not talk about **DepthRangef**.
- Removed "Libraries" section from appendix F, since library naming is defined in the separate OpenGL ES Implementer's Guidelines document.

F.3.12 Version 2.0.21, draft of 2008/01/10

Fixes from Steve Hill (**https://cvs.khronos.org/bugzilla/show_bug.cgi?id=2800**):

- Updated or removed dangling references to fixed-function pipeline and/or state in sections 2.1, 2.6, 2.6.1, and 3.8.2.
- Updated references to commands which cause program validation in section 2.10.5.
- Fixed typos in sections 1.4 and 2.5.

Fixes from Georg Kolling (**https://cvs.khronos.org/bugzilla/show_bug.cgi?id=2770**):

- Clarified that the range and precision requirements in section 2.1.1 do not apply during shader execution.
- Replaced example declaration of **Color4f** and fixed example declarations of **Uniform[1234][if]** in section 2.3.
- Removed "Position Invariance" language preceding section 2.10.5, since GLSL ES has neither a fixed-function pipeline nor an **ftransform** builtin.
- Removed legacy operations following section 2.11, including client-defined clip planes and the sections "Two-Sided Mode", "Clamping", "Flatshading", and "Final Color Processing" following section 2.13, since these operations do not apply to primitive assembly in OpenGL ES . Moved equation 3.4

and the definition of **FrontFace** into section 3.5.1, where it's still used, and removed references to `VERTEX_PROGRAM_TWO_SIDE` from section 2.10.6. Changed section 3.8.2 to refer to the new location of equation 3.4, and removed legacy language about quads and point/line **PolygonMode** rendering.

- Changed description of normalized "fixed-point types" to "integer types" in caption to table 2.4.
- Noted how to determine if binary shader loading is supported in section 2.10.
- Removed references to fixed-function / built-in / "conventional" attributes and uniforms from sections 2.10.4 and 2.10.4.
- Removed references to non-square matrices and 1D / 3D samplers from sections 2.10.4 and 2.10.4, including the **UniformMatrix**{**2x3,3x2,2x4,4x2,3x4,4x3**}**fv** commands.
- Clarified that fragment shaders use varying input values generated by rasterization to generate fragment color outputs in section 3, and replaced references to associated "colors" and "texture coordinates" of a fragment with "varying data".
- Removed "potentially clipped' language in section 3.3 referring to `gl_PointSize`.
- Removed references to fixed-function color varyings and to vertex shaders being disabled in section 3.8.1.
- Removed references to built-in variables `gl_Color` and `gl_SecondaryColor` in section 3.8.2.
- Removed references to built-in output `gl_FragDepth` in section 3.8.2.
- Removed reference to two-sided lighting in section 4.1.4.
- Removed reference to multiple color buffers in section 4.1.6.
- Removed EXT suffix from **GetFramebufferAttachmentParameteriv** and **GetRenderbufferParameteriv**, and removed `FRAMEBUFFER_ATTACHMENT_TEXTURE_3D_ZOFFSET` in section 6.1.3.
- Fixed format of `SHADING_LANGUAGE_VERSION` string in section 6.1.5.
- Dropped no-longer-used state variable types from table 6.1.

- Changed minimum number of vertex attributes to 8 in table 6.2.
- Changed minimum number of texture binding points to 8 in tables 6.9 and 6.7.
- Removed fixed-function state from invariance rules in appendix A.3.
- Removed corollaries involving fixed-function state in appendix B, and simplified the corollary regarding coordinate systems.
- Corrected name of `<GLES2/gl2platform.h>` in appendix F.1.

Other changes:

- Removed duplicated language from the start of section 4.1.6.
- Added a description of conversion from floating-point to framebuffer fixed-point in section 2.1.2, using language formerly in the removed "Final Conversion" section.

F.3.13 Version 2.0.21, draft of 2008/01/03

Initial revision of the full specification, based on the 2.0.21 diff specification.

Index

*GetString, 126

ACTIVE_ATTRIBUTE_MAX_-LENGTH, 32, 128
ACTIVE_ATTRIBUTES, 31, 128
ACTIVE_TEXTURE, 64, 84, 123
ACTIVE_UNIFORM_MAX_LENGTH, 35, 128
ACTIVE_UNIFORMS, 34, 35, 128
ActiveTexture, 37, 64
ALIASED_POINT_SIZE_RANGE, 49
ALPHA, 60, 61, 66, 69, 85, 97, 104, 152
ALWAYS, 93, 94, 142
ARRAY_BUFFER, 21–24, 124
ARRAY_BUFFER_BINDING, 23
ATTACHED_SHADERS, 128, 129
AttachShader, 28, 174

BACK, 55, 93, 100, 137
BindAttribLocation, 32, 33
BindBuffer, 21, 24
BindFramebuffer, 106, 107, 116, 172
BindRenderbuffer, 108, 109
BindTexture, 37, 64, 83
BLEND, 95
BlendColor, 96
BlendEquation, 95
BlendEquationSeparate, 95
BlendFunc, 96
BlendFuncSeparate, 96
BLUE, 152
BOOL, 35
BOOL_VEC2, 35
BOOL_VEC3, 35
BOOL_VEC4, 35
BUFFER_SIZE, 21, 23
BUFFER_USAGE, 21, 23
BufferData, 22, 24
BufferSubData, 23, 24
bvec2, 36
BYTE, 18

CCW, 55, 137
CheckFramebufferStatus, 116–119
CLAMP_TO_EDGE, 74–76, 86
Clear, 101, 102
ClearColor, 101
ClearDepthf, 101
ClearStencil, 101
COLOR_ATTACHMENT0, 89, 103, 107, 110, 115, 124
COLOR_BUFFER_BIT, 101, 102
ColorMask, 100, 101, 118
COMPILE_STATUS, 26, 128
CompileShader, 26, 27
COMPRESSED_TEXTURE_FOR-MATS, 71
CompressedTexImage, 73
CompressedTexImage2D, 71, 72, 116
CompressedTexSubImage2D, 72, 73
CONSTANT_ALPHA, 97
CONSTANT_COLOR, 97
CopyTexImage, 69, 172

CopyTexImage2D, 67, 69–71, 79, 102, 111, 114, 116
CopyTexSubImage, 117
CopyTexSubImage2D, 70, 71, 111
CreateProgram, 28, 29
CreateShader, 26
CULL_FACE, 55
CullFace, 55, 57
CURRENT_VERTEX_ATTRIB, 131, 176

DECR, 93
DECR_WRAP, 93
DELETE_STATUS, 27, 127, 128
DeleteBuffers, 21, 22
DeleteFramebuffers, 107
DeleteProgram, 30
DeleteRenderbuffers, 109, 116
DeleteShader, 27
DeleteTexture, 161, 162
DeleteTextures, 83, 116, 161
DEPTH_ATTACHMENT, 89, 107, 110, 115, 124
DEPTH_BUFFER_BIT, 101, 102
DEPTH_COMPONENT16, 115
DEPTH_TEST, 94
DepthFunc, 94
DepthMask, 100, 101, 118
DepthRangef, 43, 123, 178
DetachShader, 29
Disable, 55, 57, 91, 92, 94, 95, 98
DisableVertexAttribArray, 19, 131
DITHER, 98
DONT_CARE, 121, 148
DrawArrays, 15, 20, 23, 24, 30, 39, 117
DrawElements, 15, 20, 23, 24, 30, 39, 117
DST_ALPHA, 97
DST_COLOR, 97
DYNAMIC_DRAW, 21, 22

ELEMENT_ARRAY_BUFFER, 24, 124
Enable, 55, 57, 91, 92, 94, 95, 98, 122
EnableVertexAttribArray, 19, 131
EQUAL, 93, 94
EXTENSIONS, 126, 166, 167

FALSE, 26, 29, 30, 36, 37, 40, 41, 92, 123, 125–128, 162, 172, 177
false, 86, 177
FASTEST, 121
Finish, 120, 159
FIXED, 18
FLOAT, 18, 20, 32, 35, 134
float, 31
FLOAT_MAT2, 32, 35
FLOAT_MAT3, 32, 35
FLOAT_MAT4, 32, 35
FLOAT_VEC2, 32, 35
FLOAT_VEC3, 32, 35
FLOAT_VEC4, 32, 35
Flush, 120, 159
FRAGMENT_PRECISION_HIGH, 173, 174
FRAGMENT_SHADER, 84, 127, 130
FRAMEBUFFER, 89, 105–107, 110, 111, 117, 119, 124
FRAMEBUFFER_ATTACHMENT_OBJECT_NAME, 110, 112, 115, 124
FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE, 110, 112, 115, 124, 125
FRAMEBUFFER_ATTACHMENT_TEXTURE_3D_ZOFFSET, 179
FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE, 112, 124
FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL, 80, 112,

113, 124
FRAMEBUFFER_BINDING, 80, 103, 106, 110, 111, 117, 118
FRAMEBUFFER_COMPLETE, 117, 118
FRAMEBUFFER_INCOMPLETE_ATTACHMENT, 116
FRAMEBUFFER_INCOMPLETE_DIMENSIONS, 116
FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT, 116
FRAMEBUFFER_UNSUPPORTED, 116, 117
FramebufferRenderbuffer, 110, 111, 116, 118, 119, 172
FramebufferTexture2D, 111, 112, 116, 118, 119, 172
FRONT, 55, 93, 100
FRONT_AND_BACK, 55, 93, 100
FrontFace, 55, 86, 179
FUNC_ADD, 95, 96, 98, 142
FUNC_REVERSE_SUBTRACT, 95, 96, 177
FUNC_SUBTRACT, 95, 96

GenBuffers, 22
GENERATE_MIPMAP_HINT, 121
GenerateMipmap, 81, 82, 119, 121, 175
GenerateMipmaps, 175
GenFramebuffers, 107, 172
GenRenderbuffers, 109
GenTextures, 83, 125
GEQUAL, 93, 94
Get, 44, 122, 123
GetActiveAttrib, 31, 32
GetActiveUniform, 34, 35, 37, 177
GetAttachedShaders, 129
GetAttribLocation, 32, 33
GetBooleanv, 92, 122, 123, 132
GetBufferParameteriv, 123
GetError, 13
GetFloatv, 9, 92, 122, 123, 132
GetFramebufferAttachmentParameteriv, 118, 124, 172, 179
GetIntegerv, 48, 103, 109, 122, 123, 132
GetProgramInfoLog, 29, 129
GetProgramiv, 29, 31, 32, 35, 40, 128, 129
GetRenderbufferAttachmentParameteriv, 172
GetRenderbufferParameteriv, 110, 125, 179
GetShaderInfoLog, 26, 129, 172, 174
GetShaderiv, 26, 27, 127, 129, 130, 172
GetShaderPrecisionFormat, 27, 130, 172–175
GetShaderSource, 129
GetString, 126
GetTexParameter, 123
GetUniform*, 132
GetUniformfv, 131
GetUniformiv, 131
GetUniformLocation, 34, 35, 37
GetVertexAttribfv, 131
GetVertexAttribiv, 131
GetVertexAttribPointerv, 131
gl_Color, 179
GL_ES_VERSION_n_m, 168
gl_FragColor, 87
gl_FragCoord, 86
gl_FragData, 87
gl_FragData[0], 87, 177
gl_FragData[n], 177
gl_FragDepth, 179
gl_FrontFacing, 86, 177
gl_PointCoord, 49
gl_PointSize, 49, 179
gl_Position, 38, 42
gl_SecondaryColor, 179

glBindTexture, 161
GREATER, 93, 94
GREEN, 152

HIGH_FLOAT, 130
HIGH_INT, 130
Hint, 120

IMPLEMENTATION_COLOR_-READ_FORMAT, 103, 176
IMPLEMENTATION_COLOR_-READ_TYPE, 103, 176
INCR, 93
INCR_WRAP, 93
IndexMask, 100
INFO_LOG_LENGTH, 128, 129
INT, 35, 104, 176
INT_VEC2, 35
INT_VEC3, 35
INT_VEC4, 35
INVALID_ENUM, 14, 28, 64, 117, 119, 124, 125
INVALID_FRAMEBUFFER_OPERATION, 14, 71, 117, 118
INVALID_OPERATION, 14, 25, 27–30, 32, 34, 36–40, 61, 64, 65, 69, 72, 73, 82, 83, 103, 110, 111, 118, 119, 123–125, 127–132, 171, 175, 176, 178
INVALID_VALUE, 14, 18–20, 23, 25, 28, 29, 31, 32, 35, 37, 44, 50, 58, 65–67, 69, 70, 72, 73, 91, 101, 109, 111, 119, 127, 131, 174, 175
INVERT, 93
IsBuffer, 126
IsEnabled, 91, 122, 132
IsFramebuffer, 127
IsProgram, 128
IsRenderbuffer, 127
IsShader, 127
IsTexture, 125, 162

KEEP, 93, 142

LEQUAL, 93, 94
LESS, 93, 94, 142
level zero array, 66
LINE_LOOP, 16
LINE_STRIP, 16
LINEAR, 74, 78–80, 82, 85, 86, 113
LINEAR_MIPMAP_LINEAR, 74, 78–80, 113
LINEAR_MIPMAP_NEAREST, 74, 78–80, 113
LINES, 16
LineWidth, 50
LINK_STATUS, 29, 128
LinkProgram, 28–31, 33, 35, 38
LOW_FLOAT, 130
LOW_INT, 130
LUMINANCE, 60, 61, 63, 66, 69, 85, 103, 173
LUMINANCE_ALPHA, 60, 61, 63, 66, 69, 85, 103

m, 168
mat2, 31
mat3, 31
mat4, 31
MAX_COMBINED_TEXTURE_IMAGE_UNITS, 39, 64, 123
MAX_CUBE_MAP_TEXTURE_SIZE, 67
MAX_FRAGMENT_UNIFORM_VECTORS, 84
MAX_RENDERBUFFER_SIZE, 109, 119
MAX_TEXTURE_IMAGE_UNITS, 39, 64, 86
MAX_TEXTURE_SIZE, 67

MAX_VARYING_FLOATS, 175
MAX_VARYING_VECTORS, 38, 175
MAX_VERTEX_ATTRIBS, 17–20, 31–33, 131
MAX_VERTEX_TEXTURE_IMAGE_UNITS, 39, 64
MAX_VERTEX_UNIFORM_COMPONENTS, 175
MAX_VERTEX_UNIFORM_VECTORS, 33, 175
MEDIUM_FLOAT, 130
MEDIUM_INT, 130
MIRRORED_REPEAT, 74, 76

n, 168
NEAREST, 74, 77, 79, 80, 85, 86, 113
NEAREST_MIPMAP_LINEAR, 74, 78–80, 82, 113
NEAREST_MIPMAP_NEAREST, 74, 78–80, 113
NEVER, 93, 94
NICEST, 121
NO_ERROR, 13, 153
NONE, 115, 124, 125, 155, 173
NOTEQUAL, 93, 94
NULL, 20, 26, 31, 35, 129, 130, 133, 134
NUM_COMPRESSED_TEXTURE_FORMATS, 71
NUM_SHADER_BINARY_FORMATS, 25, 27

ONE, 97, 98, 142
ONE_MINUS_CONSTANT_ALPHA, 97
ONE_MINUS_CONSTANT_COLOR, 97
ONE_MINUS_DST_ALPHA, 97
ONE_MINUS_DST_COLOR, 97
ONE_MINUS_SRC_ALPHA, 97
ONE_MINUS_SRC_COLOR, 97
OUT_OF_MEMORY, 13, 14, 23, 109, 119

PACK_ALIGNMENT, 103, 144
PixelStore, 59, 103, 105
PixelStorei, 58
POINTS, 16
POLYGON_OFFSET_FILL, 57
PolygonOffset, 56

READ_FRAMEBUFFER_BINDING, 71
ReadPixels, 58, 60, 69, 102, 103, 106, 117, 173
RED, 152
ReleaseShaderCompiler, 27
RENDERBUFFER, 108–110, 124, 125
RENDERBUFFER_ALPHA_SIZE, 125
RENDERBUFFER_BINDING, 109, 119, 125
RENDERBUFFER_BLUE_SIZE, 125
RENDERBUFFER_DEPTH_SIZE, 125
RENDERBUFFER_GREEN_SIZE, 125
RENDERBUFFER_HEIGHT, 125
RENDERBUFFER_INTERNAL_FORMAT, 125, 172
RENDERBUFFER_RED_SIZE, 125
RENDERBUFFER_STENCIL_SIZE, 125
RENDERBUFFER_WIDTH, 125
RenderbufferStorage, 109, 110, 116, 119
RENDERER, 126
REPEAT, 74, 75, 77, 78, 82, 140
REPLACE, 93
RGB, 60–63, 66, 69, 85, 97
RGB565, 115, 176
RGB5_A1, 115
RGBA, 60–63, 66, 69, 85, 103

RGBA4, 115, 154, 172

SAMPLE_ALPHA_TO_COVERAGE, 91
SAMPLE_BUFFERS, 48, 49, 53, 57, 91, 98, 99, 101, 107
SAMPLE_COVERAGE, 91, 92
SAMPLE_COVERAGE_INVERT, 91, 92
SAMPLE_COVERAGE_VALUE, 91, 92
SampleCoverage, 92
sampler2D, 37
SAMPLER_2D, 35
SAMPLER_CUBE, 35
SAMPLES, 48, 107
Scissor, 91
SCISSOR_TEST, 91
SHADER_BINARY_FORMATS, 27, 28
SHADER_COMPILER, 25, 27, 128–131, 172
SHADER_SOURCE_LENGTH, 128, 130
SHADER_TYPE, 41, 127
ShaderBinary, 27, 28, 171, 172
ShaderSource, 26, 27, 130, 174
SHADING_LANGUAGE_VERSION, 126, 179
SHORT, 18
SRC_ALPHA, 97
SRC_ALPHA_SATURATE, 97
SRC_COLOR, 97
STATIC_DRAW, 21, 22, 135
STENCIL_ATTACHMENT, 89, 107, 110, 115, 124
STENCIL_BUFFER_BIT, 101, 102
STENCIL_INDEX8, 115
STENCIL_TEST, 92
StencilFunc, 92–94, 159
StencilFuncSeparate, 92, 93
StencilMask, 100, 101, 118, 159
StencilMaskSeparate, 100, 118
StencilOp, 92, 93
StencilOpSeparate, 92, 93
STREAM_DRAW, 21, 22

TexImage, 64, 70
TexImage2D, 58–61, 64–67, 69, 70, 72, 79, 102, 104, 116, 175
TexParameter, 73, 174
TexSubImage, 70
TexSubImage2D, 59, 70, 72
TEXTURE, 112, 124
TEXTUREi, 64
TEXTURE0, 64, 141
TEXTURE_2D, 37, 65, 69, 70, 73, 82, 83, 111, 123, 139
TEXTURE_CUBE_MAP, 65, 73, 82, 83, 119, 123, 139
TEXTURE_CUBE_MAP_*, 65
TEXTURE_CUBE_MAP_-NEGATIVE_X, 65, 69, 70, 75, 111
TEXTURE_CUBE_MAP_-NEGATIVE_Y, 65, 69, 70, 75, 111
TEXTURE_CUBE_MAP_-NEGATIVE_Z, 65, 69, 70, 75, 111
TEXTURE_CUBE_MAP_-POSITIVE_X, 65, 69, 70, 75, 111
TEXTURE_CUBE_MAP_-POSITIVE_Y, 65, 69, 70, 75, 111
TEXTURE_CUBE_MAP_-POSITIVE_Z, 65, 69, 70, 75, 111
TEXTURE_MAG_FILTER, 74, 80, 82
TEXTURE_MIN_FILTER, 74, 77, 78,

80, 82, 86, 113
TEXTURE_WRAP_S, 74, 75, 77, 78
TEXTURE_WRAP_T, 74, 75, 78
TRIANGLE_FAN, 17
TRIANGLE_STRIP, 16
TRIANGLES, 17
TRUE, 19, 25–27, 29, 36, 40, 92, 100, 123, 125–131, 177
true, 86, 177

Uniform, 10, 36
Uniform*, 34, 36, 37
Uniform*f{v}, 36
Uniform*i{v}, 36
Uniform1f, 10
Uniform1i, 10
Uniform1i{v}, 36, 37
Uniform1iv, 36
Uniform2f, 10
Uniform2f{v}, 36
Uniform2i, 10
Uniform2i{v}, 36
Uniform3f, 10
Uniform3i, 10
Uniform4f, 9, 10
Uniform4f{v}, 36
Uniform4i, 10
Uniform4i{v}, 36
UniformMatrix*, 37
UniformMatrix3fv, 36
UniformMatrix{234}fv, 36
UNPACK_ALIGNMENT, 59, 61, 144
UNSIGNED_BYTE, 18, 20, 60, 61, 103, 104
UNSIGNED_SHORT, 18, 20, 62
UNSIGNED_SHORT_4_4_4_4, 60–62, 104
UNSIGNED_SHORT_5_5_5_1, 60–62, 104
UNSIGNED_SHORT_5_6_5, 60–62, 104
UseProgram, 29, 30, 38, 42

VALIDATE_STATUS, 40, 128
ValidateProgram, 40, 128
vec2, 31
vec3, 31
vec4, 31, 36
VENDOR, 126
VERSION, 126
VERTEX_ATTRIB_ARRAY_BUFFER_BINDING, 23, 131
VERTEX_ATTRIB_ARRAY_ENABLED, 131
VERTEX_ATTRIB_ARRAY_NORMALIZED, 131
VERTEX_ATTRIB_ARRAY_POINTER, 131
VERTEX_ATTRIB_ARRAY_SIZE, 131
VERTEX_ATTRIB_ARRAY_STRIDE, 131
VERTEX_ATTRIB_ARRAY_TYPE, 131
VERTEX_PROGRAM_TWO_SIDE, 179
VERTEX_SHADER, 26, 127, 130
VertexAttrib, 18
VertexAttrib*, 18, 30
VertexAttrib1*, 18
VertexAttrib2*, 18
VertexAttrib3*, 18
VertexAttrib4*, 18
VertexAttribPointer, 18, 19, 23, 131
Viewport, 43

ZERO, 93, 97, 98, 142