



OpenGL<sup>®</sup> SC  
Version 2.0.0 (Full Specification)  
(April 19, 2016)

*Editors (version 2.0): Aidan Fabius, Steve Viggers*

Copyright © 2016 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 website 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, Vulkan, SPIR, SPIR-V, SYCL, WebGL, WebCL, OpenVX, OpenVG, EGL, COLLADA, glTF, OpenKODE, OpenKCAM, StreamInput, OpenWF, OpenSL ES, OpenMAX, OpenMAX AL, OpenMAX IL, OpenMAX DL and DevU are trademarks of the Khronos Group Inc. ASTC is a trademark of ARM Holdings PLC, OpenCL is a trademark of Apple Inc. and OpenGL and OpenML are registered trademarks and the OpenGL ES and OpenGL SC logos are trademarks of Silicon Graphics International used under license by Khronos. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	What is the OpenGL SC Graphics System? . . . . .	1
1.2	Suitability for Safety Critical applications? . . . . .	1
1.3	Programmer's View of OpenGL SC . . . . .	2
1.4	Implementer's View of OpenGL SC . . . . .	2
1.5	Our View . . . . .	3
1.6	Companion Documents . . . . .	3
1.6.1	Window System Bindings . . . . .	3
<b>2</b>	<b>OpenGL SC Operation</b>	<b>5</b>
2.1	OpenGL SC Fundamentals . . . . .	5
2.1.1	Numeric Computation . . . . .	7
2.1.2	Data Conversions . . . . .	8
2.2	GL State . . . . .	9
2.2.1	Shared Object State . . . . .	10
2.3	GL Command Syntax . . . . .	10
2.4	Basic GL Operation . . . . .	13
2.5	GL Errors . . . . .	14
2.6	Graphics Reset Recovery . . . . .	15
2.7	Primitives and Vertices . . . . .	17
2.7.1	Primitive Types . . . . .	19
2.8	Current Vertex State . . . . .	21
2.9	Vertex Arrays . . . . .	21
2.10	Buffer Objects . . . . .	24
2.10.1	Vertex Arrays in Buffer Objects . . . . .	26
2.10.2	Array Indices in Buffer Objects . . . . .	27
2.11	Vertex Shaders . . . . .	27
2.11.1	Program Objects . . . . .	28
2.11.2	Shader Variables . . . . .	29

2.11.3	Shader Execution . . . . .	34
2.11.4	Required State . . . . .	36
2.12	Primitive Assembly and Post-Shader Vertex Processing . . . . .	37
2.13	Coordinate Transformations . . . . .	37
2.13.1	Controlling the Viewport . . . . .	37
2.14	Primitive Clipping . . . . .	39
2.14.1	Clipping Varying Outputs . . . . .	40
<b>3</b>	<b>Rasterization</b>	<b>41</b>
3.1	Invariance . . . . .	42
3.2	Multisampling . . . . .	42
3.3	Points . . . . .	44
3.3.1	Point Multisample Rasterization . . . . .	44
3.4	Line Segments . . . . .	45
3.4.1	Basic Line Segment Rasterization . . . . .	45
3.4.2	Other Line Segment Features . . . . .	47
3.4.3	Line Rasterization State . . . . .	48
3.4.4	Line Multisample Rasterization . . . . .	48
3.5	Polygons . . . . .	49
3.5.1	Basic Polygon Rasterization . . . . .	50
3.5.2	Depth Offset . . . . .	51
3.5.3	Polygon Multisample Rasterization . . . . .	52
3.5.4	Polygon Rasterization State . . . . .	53
3.6	Pixel Rectangles . . . . .	53
3.6.1	Pixel Storage Modes . . . . .	53
3.6.2	Transfer of Pixel Rectangles . . . . .	54
3.7	Texturing . . . . .	58
3.7.1	Texture Image Specification . . . . .	59
3.7.2	Texture Image Specification Commands . . . . .	60
3.7.3	Compressed Texture Images . . . . .	62
3.7.4	Texture Parameters . . . . .	63
3.7.5	Texture Wrap Modes . . . . .	63
3.7.6	Texture Minification . . . . .	65
3.7.7	Texture Magnification . . . . .	69
3.7.8	Texture Framebuffer Attachment . . . . .	69
3.7.9	Texture Completeness and Non-Power-Of-Two Textures . . . . .	70
3.7.10	Mipmap Generation . . . . .	70
3.7.11	Texture State . . . . .	71
3.7.12	Texture Objects . . . . .	71
3.8	Fragment Shaders . . . . .	72

3.8.1	Shader Variables . . . . .	73
3.8.2	Shader Execution . . . . .	73
<b>4</b>	<b>Per-Fragment Operations and the Framebuffer</b>	<b>76</b>
4.1	Per-Fragment Operations . . . . .	77
4.1.1	Pixel Ownership Test . . . . .	77
4.1.2	Scissor Test . . . . .	79
4.1.3	Multisample Fragment Operations . . . . .	79
4.1.4	Stencil Test . . . . .	80
4.1.5	Depth Buffer Test . . . . .	82
4.1.6	Blending . . . . .	83
4.1.7	Dithering . . . . .	86
4.1.8	Additional Multisample Fragment Operations . . . . .	86
4.2	Whole Framebuffer Operations . . . . .	87
4.2.1	Selecting a Buffer for Writing . . . . .	87
4.2.2	Fine Control of Buffer Updates . . . . .	87
4.2.3	Clearing the Buffers . . . . .	89
4.3	Reading Pixels . . . . .	90
4.3.1	Reading Pixels . . . . .	90
4.3.2	Pixel Draw/Read State . . . . .	93
4.4	Framebuffer Objects . . . . .	93
4.4.1	Binding and Managing Framebuffer Objects . . . . .	93
4.4.2	Attaching Images to Framebuffer Objects . . . . .	95
4.4.3	Renderbuffer Objects . . . . .	96
4.4.4	Feedback Loops Between Textures and the Framebuffer . . . . .	99
4.4.5	Framebuffer Completeness . . . . .	100
4.4.6	Effects of Framebuffer State on Framebuffer Dependent Values . . . . .	104
4.4.7	Mapping between Pixel and Element in Attached Image . . . . .	104
4.4.8	Errors . . . . .	105
<b>5</b>	<b>Special Functions</b>	<b>106</b>
5.1	Flush and Finish . . . . .	106
5.2	Hints . . . . .	106
<b>6</b>	<b>State and State Requests</b>	<b>108</b>
6.1	Querying GL State . . . . .	108
6.1.1	Simple Queries . . . . .	108
6.1.2	Data Conversions . . . . .	108
6.1.3	Enumerated Queries . . . . .	109

6.1.4	String Queries . . . . .	111
6.1.5	Program Queries . . . . .	112
6.2	State Tables . . . . .	113
<b>A</b>	<b>Invariance</b>	<b>138</b>
A.1	Repeatability . . . . .	138
A.2	Multi-pass Algorithms . . . . .	139
A.3	Invariance Rules . . . . .	139
A.4	What All This Means . . . . .	140
<b>B</b>	<b>Corollaries</b>	<b>141</b>
<b>C</b>	<b>Shared Objects and Multiple Contexts</b>	<b>143</b>
C.1	Sharing Contexts Between Different Versions of OpenGL SC . .	143
C.2	Sharing Objects Between Different Contexts in OpenGL SC . . .	143
C.3	Propagating Changes to Objects . . . . .	144
C.3.1	Determining Completion of Changes to an object . . . . .	144
C.3.2	Definitions . . . . .	144
C.3.3	Rules . . . . .	145
<b>D</b>	<b>Version 2.0</b>	<b>147</b>
<b>E</b>	<b>Extension Registry, Header Files, and Extension Naming Conventions</b>	<b>148</b>
E.1	Extension Registry . . . . .	148
E.2	Header Files . . . . .	148
E.3	OGLSC Extensions . . . . .	149
E.3.1	Naming Conventions . . . . .	149
E.4	Vendor and EXT Extensions . . . . .	149
<b>F</b>	<b>GLSL Limitations</b>	<b>151</b>
F.1	Overview . . . . .	151
F.2	Length of Shader Executable . . . . .	151
F.3	Usage of Temporary Variables . . . . .	151
F.4	Control Flow . . . . .	151
F.5	Indexing of Arrays, Vectors and Matrices . . . . .	152
F.6	Counting of Varyings and Uniforms . . . . .	154
F.7	Shader Parameters . . . . .	157

<b>G Packaging and Acknowledgements</b>	<b>158</b>
G.1 Header Files and Libraries . . . . .	158
G.2 Acknowledgements . . . . .	158
G.3 Document History . . . . .	159
G.3.1 Version 2.0.0, April 19, 2016 . . . . .	159

# List of Figures

2.1	Block diagram of the GL. . . . .	13
2.2	Vertex processing and primitive assembly. . . . .	17
2.3	Triangle strips, fans, and independent triangles. . . . .	20
2.4	Vertex transformation sequence. . . . .	37
3.1	Rasterization. . . . .	41
3.2	Visualization of Bresenham's algorithm. . . . .	46
3.3	Rasterization of non-antialiased wide lines. . . . .	47
3.4	The region used in rasterizing a multisampled line segment. . . . .	49
3.5	Transfer of pixel rectangles to the GL. . . . .	54
4.1	Per-fragment operations. . . . .	77
4.2	Operation of <b>ReadnPixels</b> . . . . .	90



# List of Tables

2.1	GL command suffixes . . . . .	11
2.2	GL data types . . . . .	12
2.3	Summary of GL errors . . . . .	17
2.4	Vertex array sizes (values per vertex) and data types . . . . .	22
2.5	Buffer object parameters and their values. . . . .	24
2.6	Buffer object initial state. . . . .	26
3.1	<b>PixelStore</b> parameters. . . . .	54
3.2	<b>TexSubImage2D</b> and <b>ReadnPixels</b> types. . . . .	55
3.3	<b>TexSubImage2D</b> and <b>ReadnPixels</b> formats. . . . .	55
3.4	Valid pixel format, type, and sized internal format. . . . .	56
3.5	Packed pixel formats. . . . .	57
3.6	UNSIGNED_SHORT formats . . . . .	57
3.7	Packed pixel field assignments. . . . .	58
3.8	Texture parameters and their values. . . . .	64
3.9	Correspondence of filtered texture components. . . . .	74
4.1	RGB and Alpha blend equations. . . . .	84
4.2	Blending functions. . . . .	85
4.3	<b>PixelStore</b> parameters. . . . .	91
4.4	<b>ReadnPixels</b> GL data types and reversed component conversion formulas. . . . .	92
4.5	Renderbuffer image internal formats. . . . .	101
6.1	State Variable Types . . . . .	114
6.2	Vertex Array Data . . . . .	115
6.3	Buffer Object State . . . . .	116
6.4	Transformation state . . . . .	117
6.5	Rasterization . . . . .	118
6.6	Multisampling . . . . .	119

6.7	Textures (state per texture unit and binding point) . . . . .	120
6.8	Textures (state per texture object) . . . . .	121
6.9	Texture Environment and Generation . . . . .	122
6.10	Pixel Operations . . . . .	123
6.11	Pixel Operations (cont.) . . . . .	124
6.12	Framebuffer Control . . . . .	125
6.13	Pixels . . . . .	126
6.14	Program Object State . . . . .	127
6.15	Vertex Shader State . . . . .	128
6.16	Hints . . . . .	129
6.17	Implementation Dependent Values . . . . .	130
6.18	Implementation Dependent Values (cont.) . . . . .	131
6.19	Implementation Dependent Values (cont.) . . . . .	132
6.20	Implementation Dependent Pixel Depths . . . . .	133
6.21	Miscellaneous . . . . .	134
6.22	Renderbuffer State . . . . .	135
6.23	Framebuffer State . . . . .	136
6.24	Robustness State . . . . .	137

# Chapter 1

## Introduction

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

### 1.1 What is the OpenGL SC Graphics System?

OpenGL SC 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 SC requires that the graphics hardware contain a framebuffer. Many OpenGL SC 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 SC is specifically concerned with framebuffer manipulation.

### 1.2 Suitability for Safety Critical applications?

OpenGL SC 2.0 is based on the OpenGL ES 2.0 graphics system and is designed primarily for safety critical graphics hardware running on embedded devices. It removes aspects of OpenGL ES 2.0 that are not aligned with deterministic safety critical software applications. For example, the ability to delete objects (textures, etc.) has been removed to avoid memory fragmentation and garbage collection. In

addition, portions of the `GL_EXT_texture_storage` and `GL_KHR_robustness` extensions have been adopted as part of the core OpenGL SC 2.0 specification.

The OpenGL SC 2.0 API has been designed to be aligned with avionics and automotive requirements for safety critical software. In order to meet the avionics market needs, OpenGL SC has been defined to enable implementations and applications to be certifiable to DO-178 Level A. In order to meet the automotive market needs, OpenGL SC has been defined to enable implementations and applications to be certifiable to ISO26262 ASIL D.

### 1.3 Programmer's View of OpenGL SC

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

A typical program that uses OpenGL SC begins with calls to open a window into the framebuffer into which the program will draw. Then, calls are made to allocate an OpenGL SC 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 SC 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 Implementer's View of OpenGL SC

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

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

## 1.5 Our View

We view OpenGL SC 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 implementers. 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 2.11 and 3.8). These sections may include references to concepts and terms (such as shading language variable types) defined in the companion document.

OpenGL SC 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 6.1.4.

### 1.6.1 Window System Bindings

OpenGL SC requires a companion API to create and manage graphics contexts, windows to render into, and other resources beyond the scope of this Specification.

The *Khronos Native Platform Graphics Interface* or “EGL Specification” describes the EGL API for use of OpenGL ES on mobile and embedded devices. The EGL Specification is available in the Khronos Extension Registry at URL

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

Khronos strongly encourages OpenGL SC implementations to also support EGL, but some implementations may provide alternate, platform- or vendor-specific APIs with similar functionality.

Specifically, OpenGL SC is defined targeting EGL 1.4 with the *EGL\_EXT\_create\_context\_robustness* and *EGL\_KHR\_surfaceless\_context* extensions.

It is expected that the behavior of the *EGL\_EXT\_create\_context\_robustness* extension is built in to the OpenGL SC version of EGL. In particular, *EGL\_CONTEXT\_OPENGL\_ROBUST\_ACCESS\_EXT* is implied to be TRUE, and *EGL\_CONTEXT\_OPENGL\_RESET\_NOTIFICATION\_STRATEGY\_EXT* is implied to be *EGL\_LOSE\_CONTEXT\_ON\_RESET\_EXT*. This ensures that robust behavior is always enabled.

It is also expected that the behavior of the *EGL\_KHR\_surfaceless\_context* extension is built into the OpenGL SC version of EGL. In particular, *eglMakeCurrent* is relaxed to allow *EGL\_NO\_SURFACE* for draw and read surfaces.

## Chapter 2

# OpenGL SC Operation

### 2.1 OpenGL SC Fundamentals

OpenGL SC (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 the commands being ignored.

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

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

Allocation and configuration of the default framebuffer occurs outside of the GL in conjunction with the window system, using companion APIs, such as EGL. Allocation and initialization of GL contexts is also done using these companion APIs. GL contexts can typically be associated with different default framebuffers,



and some context state is determined at the time this association is performed.

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

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

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

### 2.1.1 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 2.11 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.

### 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

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

### 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  $k \in \{0, 1, \dots, 2^m - 1\}$ , as  $k$  (e.g. 1.0 is represented in binary as a string of all ones).  $m$  must be at least as large as the number of bits in the corresponding component of the framebuffer.  $m$  must be at least 2 for A if the framebuffer does not contain an A component, or if there is only 1 bit of A in the framebuffer.

## 2.2 GL State

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

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

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

### 2.2.1 Shared Object State

It is possible for groups of contexts to share certain state. Enabling such sharing between contexts is done through window system binding APIs such as those described in section 1.6.1. These APIs are responsible for creation and management of contexts, and not discussed further here. More detailed discussion of the behavior of shared objects is included in appendix C. Except as defined in this appendix, all state in a context is specific to that context only.

## 2.3 GL Command Syntax

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

GL commands are formed from a *name* followed, depending on the particular command, by up to 4 characters. The first character indicates the number of values of the indicated type that must be presented to the command. The second character or character pair indicates the specific type of the arguments: 32-bit integer 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<sup>1</sup>

---

<sup>1</sup>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
<b>i</b>	int
<b>f</b>	float

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

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

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

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.

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
sizei	32	Non-negative binary integer size
enum	32	Enumerated binary integer value
intptr	<i>ptrbits</i>	Signed 2's complement binary integer
sizeiptr	<i>ptrbits</i>	Non-negative binary integer size
bitfield	32	Bit field
float	32	Floating-point value

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.

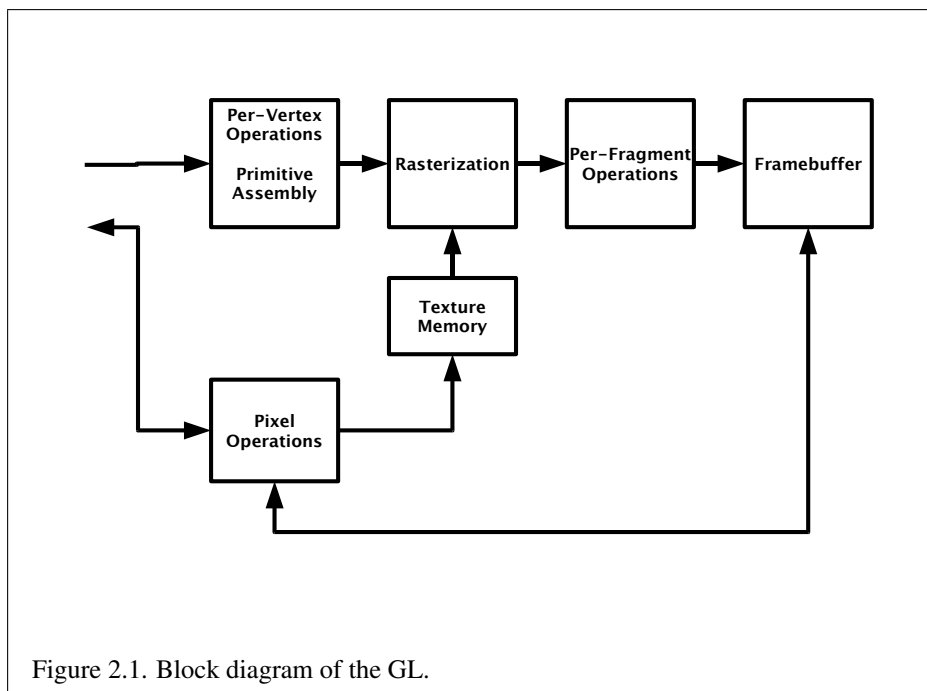


Figure 2.1. Block diagram of the GL.

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.

## 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 individ-

ual 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.

From a safety critical point of view there are two types of error condition that can occur; deterministic or non-deterministic. It is advised for non-deterministic errors, those that leave the run-time in an unknown state or unstable condition, to not continue working with the current runtime instance and abort.

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, 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, including `OUT_OF_MEMORY`, but not to system errors such as memory access errors. Extensions may change behavior that would otherwise generate errors in an unextended GL implementation.

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.
- If the GL context has been reset as a result of a previous GL command, or if the context is reset as a side effect of execution of a command, a `CONTEXT_LOST` error is generated.

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

## 2.6 Graphics Reset Recovery

Certain events can result in a reset of the GL context. After such an event, it is referred to as a *lost context* and is unusable for almost all purposes. Recovery requires creating a new context and recreating all relevant state from the lost context. The current status of the graphics reset state is returned by

```
enum GetGraphicsResetStatus( void );
```

The value returned indicates if the GL context has been in a reset state at any point since the last call to `GetGraphicsResetStatus`:

- `NO_ERROR` indicates that the GL context has not been in a reset state since the last call

- `GUILTY_CONTEXT_RESET` indicates that a reset has been detected that is attributable to the current GL context
- `INNOCENT_CONTEXT_RESET` indicates a reset has been detected that is not attributable to the current GL context
- `UNKNOWN_CONTEXT_RESET` indicates a detected graphics reset whose cause is unknown

If a reset status other than `NO_ERROR` is returned and subsequent calls return `NO_ERROR`, the context reset was encountered and completed. If a reset status is repeatedly returned, the context may be in the process of resetting.

Reset notification behavior is determined at context creation time, and may be queried by calling `GetIntegerv` with the symbolic constant `RESET_NOTIFICATION_STRATEGY`.

If the reset notification behavior is `NO_RESET_NOTIFICATION`, then the implementation will never deliver notification of reset events, and **`GetGraphicsResetStatus`** will always return `NO_ERROR`<sup>2</sup>.

If the behavior is `LOSE_CONTEXT_ON_RESET`, a graphics reset will result in a lost context and require creating a new context as described above. In this case **`GetGraphicsResetStatus`** will return an appropriate value from those described above.

If a graphics reset notification occurs in a context, a notification must also occur in all other contexts which share objects with that context<sup>3</sup>.

After a graphics reset has occurred on a context, subsequent GL commands on that context (or any context which shares with that context) will generate a `CONTEXT_LOST` error. Such commands will not have side effects (in particular, they will not modify memory passed by pointer for query results), and may not block indefinitely or cause termination of the application. Exceptions to this behavior include:

- **`GetError`** and **`GetGraphicsResetStatus`** behave normally following a graphics reset, so that the application can determine a reset has occurred, and when it is safe to destroy and recreate the context
- Any commands which might cause a polling application to block indefinitely will generate a `CONTEXT_LOST` error, but will also return a value indicating completion to the application.

---

<sup>2</sup> In this case it is recommended that implementations should not allow loss of context state no matter what events occur. However, this is only a recommendation, and cannot be relied upon by applications.

<sup>3</sup> The values returned by **`GetGraphicsResetStatus`** in the different contexts may differ.

Error	Description	Offending command ignored?	Advisory Action
INVALID_ENUM	enum argument out of range	Yes	continue
INVALID_FRAMEBUFFER_OPERATION	Framebuffer is incomplete	Yes	continue
INVALID_VALUE	Numeric argument out of range	Yes	continue
INVALID_OPERATION	Operation illegal in current state	Yes	continue
OUT_OF_MEMORY	Not enough memory left to execute command	Unknown	abort
CONTEXT_LOST	Context has been lost and reset by the driver	Except as noted for specific command	continue

Table 2.3: Summary of GL errors

## 2.7 Primitives and Vertices

In the GL, geometric objects are drawn by specifying a series of generic attribute sets using vertex arrays (see section 2.9). 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 2.11) and used to compute values for consumption by later processing stages.

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

Before vertex shader execution, the state required by a vertex is its 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,

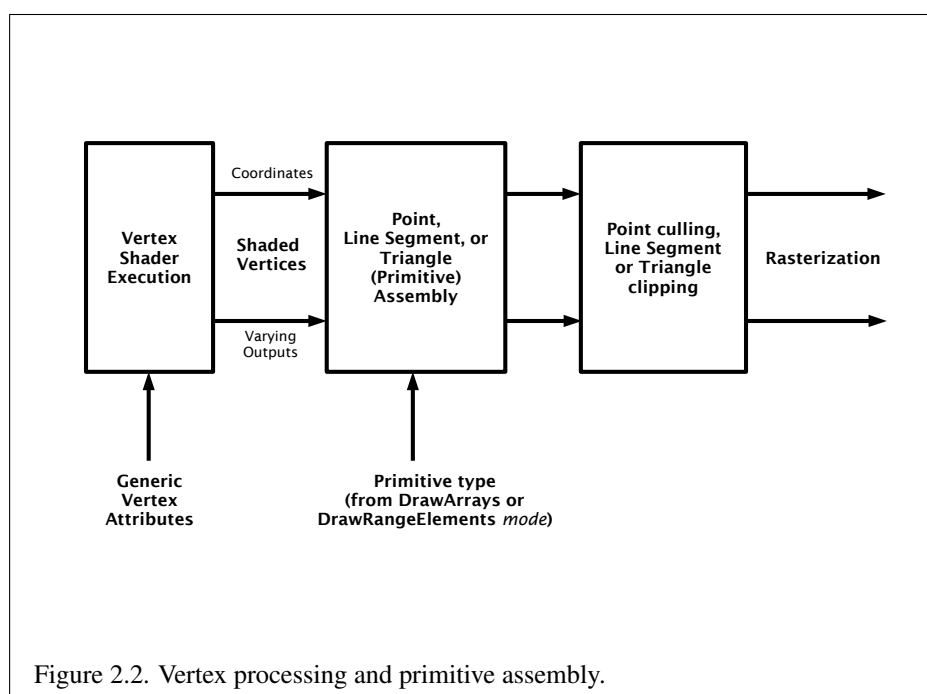


Figure 2.2. Vertex processing and primitive assembly.

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.

### 2.7.1 Primitive Types

A sequence of vertices is passed to the GL using the commands **DrawArrays** or **DrawRangeElements** (see section 2.9). 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  $i$ th vertex (for  $i > 1$ ) specifies the beginning of the  $i$ th segment and the end of the  $i - 1$ st. The last vertex specifies the end of the last segment. If only one vertex is specified, 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

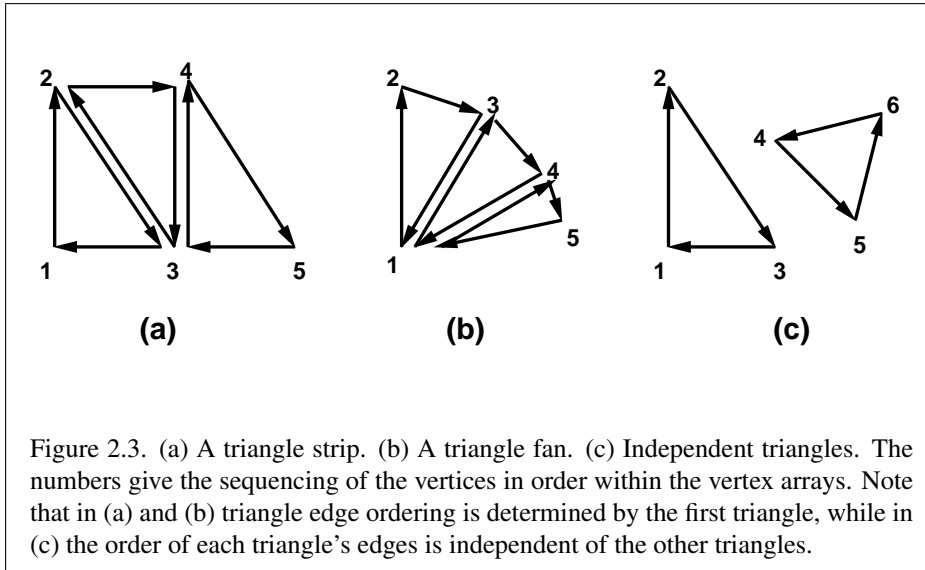


Figure 2.3. (a) A triangle strip. (b) A triangle fan. (c) Independent triangles. The numbers give the sequencing of the vertices in order within the vertex arrays. Note that in (a) and (b) triangle edge ordering is determined by the first triangle, while in (c) the order of each triangle's edges is independent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

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 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  $i = 0, 1, \dots, n - 1$ , where there are  $3n + k$  vertices drawn.  $k$  is either 0, 1, or 2; if  $k$  is not zero, the final  $k$  vertices are ignored. For each triangle, vertex A is vertex  $3i$  and vertex B is vertex  $3i + 1$ . Otherwise, separate triangles are the same as a triangle strip.

The order of the vertices in a triangle generated from a triangle strip, trian-

gle fan, or separate triangles is significant in polygon rasterization and fragment shading (see sections 3.5.1 and 3.8.2).

## 2.8 Current Vertex State

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

Current generic attribute values define generic attributes for a vertex when a vertex array defining that data is not enabled, as described in section 2.9. 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  $2 \times 2$ ,  $3 \times 3$  or  $4 \times 4$  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).

## 2.9 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 2.10). 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

Command	Sizes	Normalized	Types
<b>VertexAttribPointer</b>	1,2,3,4	<i>flag</i>	byte, ubyte, short, ushort, 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.

```
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`, and `FLOAT`, indicate types `byte`, `ubyte`, `short`, `ushort`, and `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 *i*th and (*i* + 1)st elements of an array differ by *stride* basic machine units (typically unsigned bytes), the pointer to the (*i* + 1)st element being greater. For each command, *pointer* specifies the location in memory of the first value of the first element of the array being specified.

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

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



```
void DisableVertexArray( 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 **DrawRangeElements** 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, size_t 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 2.7.1. 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 2.8).

Specifying *first* < 0 results in generating the error `INVALID_VALUE`.

The command

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

constructs a sequence of geometric primitives by successively transferring the *count* elements whose indices are stored in *indices* to the GL. The *i*th element transferred by **DrawRangeElements** will be taken from element *indices*[*i*] of each enabled array. *type* must be `UNSIGNED_SHORT`, indicating that the values in *indices* are indices of GL type `ushort`. *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 2.8).

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

Table 2.5: Buffer object parameters and their values.

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 `GLfloat`, and the integers representing values per element are each four.

## 2.10 Buffer Objects

The vertex data arrays described in section 2.9 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 a buffer object name obtained by calling **GenBuffers** 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 object buffer name obtained from **GenBuffers**. 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. Calling **BindBuffer** with a buffer object name not obtained by **GenBuffers** will result in a `INVALID_OPERATION` error.

**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 cannot be deleted.

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.

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 uninitialized.

*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** sets the values of the buffer object's state variables as shown in table 2.6. Once established these values are considered immutable and cannot be changed. Attempting to change these state variables of a buffer object once established will result in an `INVALID_OPERATION` error.

Name	Value
BUFFER_SIZE	<i>size</i>
BUFFER_USAGE	<i>usage</i>

Table 2.6: Buffer object initial state.

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

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

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

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

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

### 2.10.1 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 **DrawRangeElements** 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<sup>4</sup>.

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.

### 2.10.2 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 **DrawRangeElements** 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 2.10.

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 2.10, 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`, **DrawRangeElements** 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 2.10.1.

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.

## 2.11 Vertex Shaders

Vertices specified with **DrawArrays** or **DrawRangeElements** are processed by the *vertex shader*. Each vertex attribute consumed by the vertex shader (see sec-

---

<sup>4</sup> 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 2.9.

tion 2.11.2) is set to the corresponding generic vertex attribute value from the array element being 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 2.12).

A vertex shader contains 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 *compiled* off-line into a *shader object*. Shader objects are linked off-line into a *program binary object* which generates executable code from the specified compiled shader objects. When a linked program binary 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* are also compiled off-line into shader objects. Fragment shaders affect the processing of fragments during rasterization, and are described in section 3.8.

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

### 2.11.1 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. Program objects cannot be deleted and once a program binary object is loaded into program object, it cannot be replaced.

To load a pre-compiled program binary into a program object, use the command

```
void ProgramBinary( uint program, enum binaryformat,  
                    const void *binary, size_t length );
```

This command will copy the pre-compiled program binary object of size *length* from *binary* into the program object *program*. Calling **ProgramBinary** with a

*program* that already has a program binary object will generate an `INVALID_OPERATION` error and the existing program binary object is not modified.

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

When a program object has a program binary object loaded, 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 a successfully loaded program binary. If **UseProgram** is called with *program* set to zero, then the current rendering state refers to an *invalid* program object, and no vertex or fragment shader executions due to any **DrawArrays** or **DrawRangeElements** commands are performed. However, this is not an error. If *program* does not contain a successfully loaded program binary, the error `INVALID_OPERATION` is generated and the current rendering state is not modified.

### 2.11.2 Shader Variables

A vertex shader can reference a number of variables as it executes. *Vertex attributes* are the per-vertex values specified in section 2.8. *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 is 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`.

The bindings of attribute variable names to indices can be queried using 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 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* was not successfully loaded, the error `INVALID_OPERATION` is generated. If *name* is not an active attribute, or if an error occurs, -1 will be returned.

### 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. A uniform is considered *active* if it is determined by the compiler and linker that the uniform will actually be accessed when the 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 loaded, all active uniforms belonging to the program object are initialized to zero (FALSE for booleans). A successful link will 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).

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 loaded, the error INVALID\_OPERATION is generated.

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 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*  $2 \times 2$ ,  $3 \times 3$ , or  $4 \times 4$  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  $3 \times 3$  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 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 off-line linker determines if a sampler is active or not. The off-line linker 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.

### 2.11.3 Shader Execution

If a successfully loaded 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 implementation does not support texture accesses in vertex shaders. The maximum number of texture image units available to the fragment stage of the GL is `MAX_TEXTURE_IMAGE_UNITS`. Both the vertex shader and fragment processing combined cannot use more than `MAX_COMBINED_TEXTURE_IMAGE_UNITS` texture image units. If both the vertex shader and the fragment processing stage access the same texture image unit, then that counts as using two texture image units against the `MAX_COMBINED_TEXTURE_IMAGE_UNITS` limit.

When a texture lookup is performed in a vertex shader, the filtered texture value  $\tau$  is computed in the manner described in sections 3.7.6 and 3.7.7, and converted to

a texture source color  $C_s$  according to table 3.9 (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 3.7.6. 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.

Using a sampler in 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.

### Out-of-Bounds Robustness Behavior

GL commands operating on buffer objects will detect attempts to read from or write to a location in a bound buffer object at an offset less than zero, or greater than or equal to the buffers size. When such an attempt is detected, a GL error is generated. Any command unable to generate a GL error, such as buffer object accesses from the active program, will not read or modify memory outside of the data store of the buffer object and will not result in GL interruption or termination.

Out-of-bounds accesses will be bounded within the working memory of the active program, cannot access memory owned by other GL contexts, and will not result in abnormal program termination. Out-of-bounds access to local and global variables cannot read values from other program invocations.

Out-of-bounds accesses to arrays of resources, such as an array of textures, can only access the data of bound resources. Reads from unbound resources return zero and writes are discarded. It is not possible to access data owned by other GL contexts unless context sharing is enabled.

Out-of-bounds buffer object reads may return any of the following values:

- Values from anywhere within the buffer object.
- Zero values, or  $(0,0,0,x)$  vectors for vector reads where  $x$  is a valid value represented in the type of the vector components and may be any of
  - 0, 1, or the maximum representable positive integer value, for signed or unsigned integer components

0.0 or 1.0, for floating-point components

However, indices within the element array that reference vertex data that lies outside the enabled attribute's vertex buffer object result in reading zero.

Applications that require defined behavior for out-of-bounds accesses should range check all computed indices before dereferencing the array, vector or matrix.

#### 2.11.4 Required State

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

The state required per program object consists of:

- An unsigned integer indicating the program object name.
- A boolean holding the status of the last load attempt, initially `FALSE`.
- A boolean holding the status of the last validation attempt, initially `FALSE`.
- 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.

## 2.12 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 2.7.1 and 2.9). The steps of primitive assembly are described in the remaining sections of this chapter and include

- Perspective division on clip coordinates (section 2.13).
- Viewport mapping, including depth range scaling (section 2.13.1).
- Primitive clipping (section 2.14).
- Clipping varying outputs (section 2.14.1).

## 2.13 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

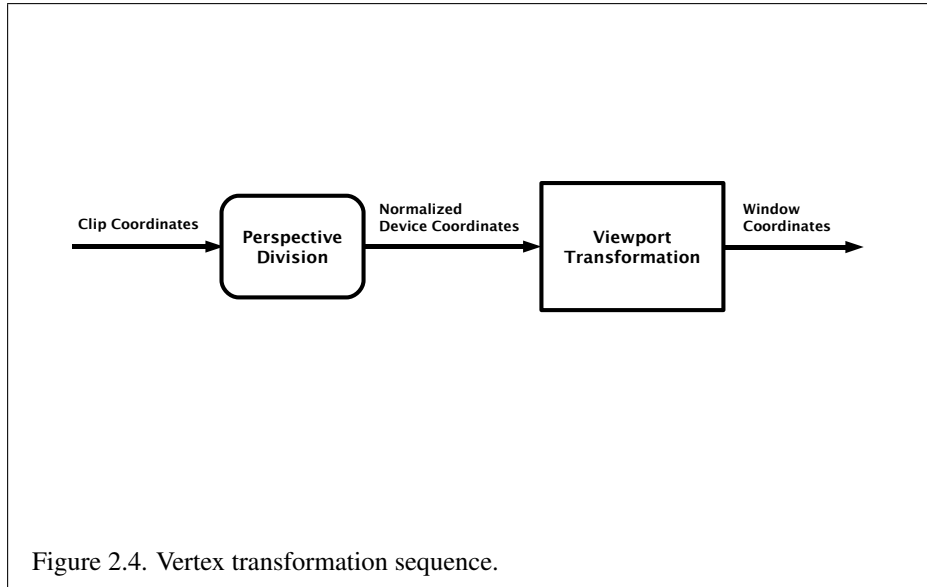
$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix}$$

then the vertex's normalized device coordinates are

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

### 2.13.1 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( float  $n$ , float  $f$  );
```

Each of  $n$  and  $f$  are clamped to lie within  $[0, 1]$ , as are all arguments of type `float`.  $z_w$  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  $y$ , size_t  $w$ , size_t  $h$  );
```

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. If no default framebuffer is associated with the GL context (see chapter 4), then  $w$  and  $h$  are initially set to zero.  $o_x$  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.

## 2.14 Primitive Clipping

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

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

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

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

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.

### 2.14.1 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 2.14) for a clipped point  $\mathbf{P}$  is used to obtain the value associated with  $\mathbf{P}$  as<sup>5</sup>

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

(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.

---

<sup>5</sup> Since this computation is performed in clip space before division by  $w_c$ , clipped varying values are perspective-correct.

## Chapter 3

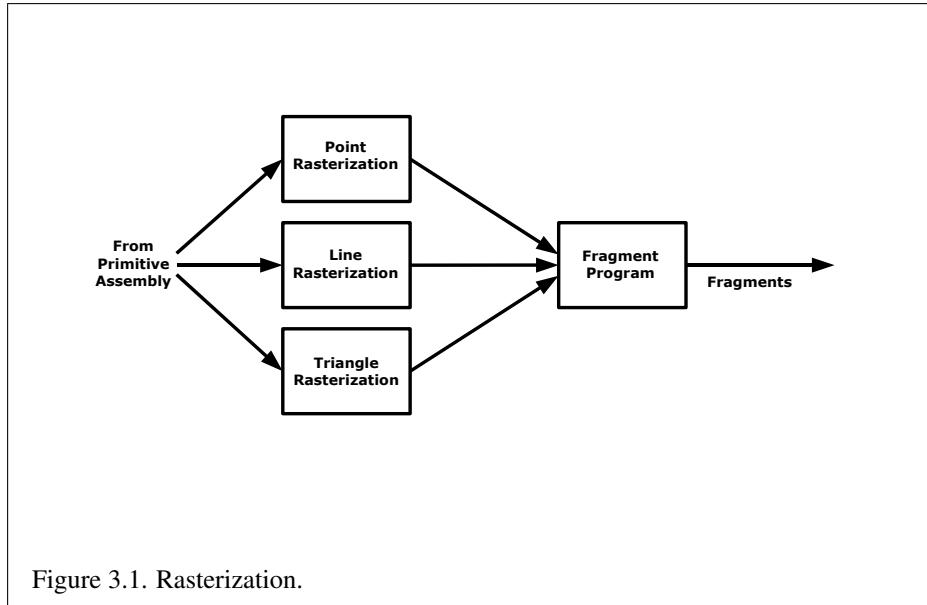
# Rasterization

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).



### 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 **GetIntegeriv** 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 **GetIntegeriv** 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 <sup>1</sup>.

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.

---

<sup>1</sup>When using EGL to create OpenGL SC 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. 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  $R_f$ .  $\epsilon$  is chosen to be so small that rasterizing the line segment produces the same fragments when  $\delta$  is substituted for  $\epsilon$  for any  $0 < \delta \leq \epsilon$ .

When  $\mathbf{p}_a$  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,

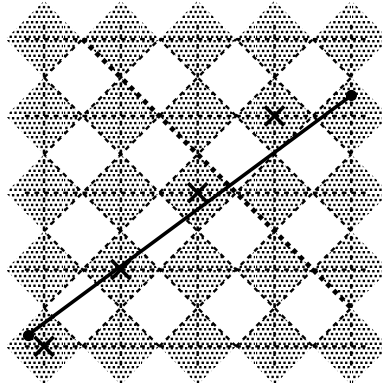


Figure 3.2. Visualization of Bresenham's algorithm. A portion of a line segment is shown. A diamond shaped region of height 1 is placed around each fragment center; those regions that the line segment exits cause rasterization to produce corresponding fragments.

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

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

(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

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

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:

$$f = (1-t)f_a + tf_b. \quad (3.3)$$

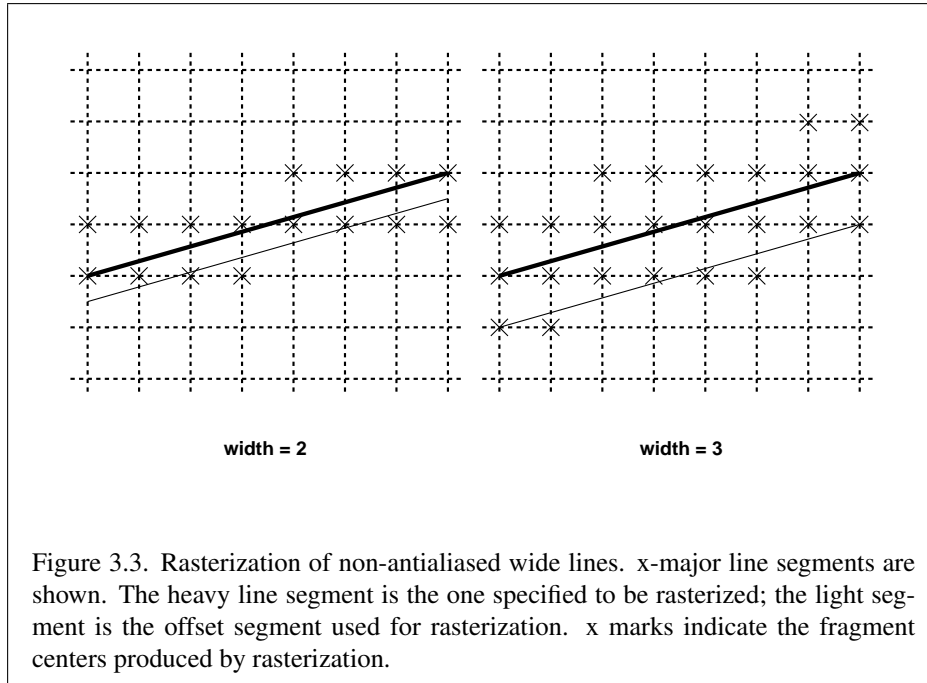
### 3.4.2 Other Line Segment Features

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



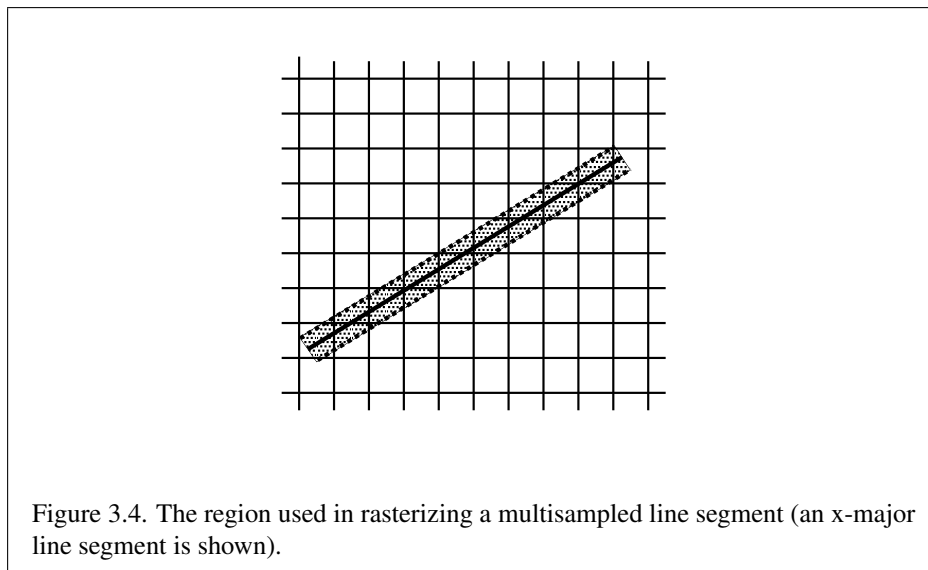
given by  $(x_0, y_0)$  and  $(x_1, y_1)$  in window coordinates, the segment with endpoints  $(x_0, y_0 - (w - 1)/2)$  and  $(x_1, y_1 - (w - 1)/2)$  is rasterized, but instead of a single fragment, a column of fragments of height  $w$  (a row of fragments of length  $w$  for a  $y$ -major segment) 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



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

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

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

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

Setting *dir* to CCW (corresponding to counter-clockwise orientation of the projected polygon in window coordinates) indicates that the sign of  $a$  should be reversed prior to use. Setting *dir* to CW (corresponding to clockwise orientation) uses the sign of  $a$  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

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

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

Denote 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

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

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

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

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

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

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

$$o = m * factor + r * units. \quad (3.8)$$

$m$  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 **TexSubImage2D** and related commands described in section 3.7.1. Some of the parameters and operations governing the operation of **TexSubImage2D** are shared by **ReadnPixels** (used to obtain pixel values from the framebuffer); the discussion of **ReadnPixels**, 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 **TexSubImage2D** also pertain to **ReadnPixels**.

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 **TexSubImage2D** and **ReadnPixels** (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

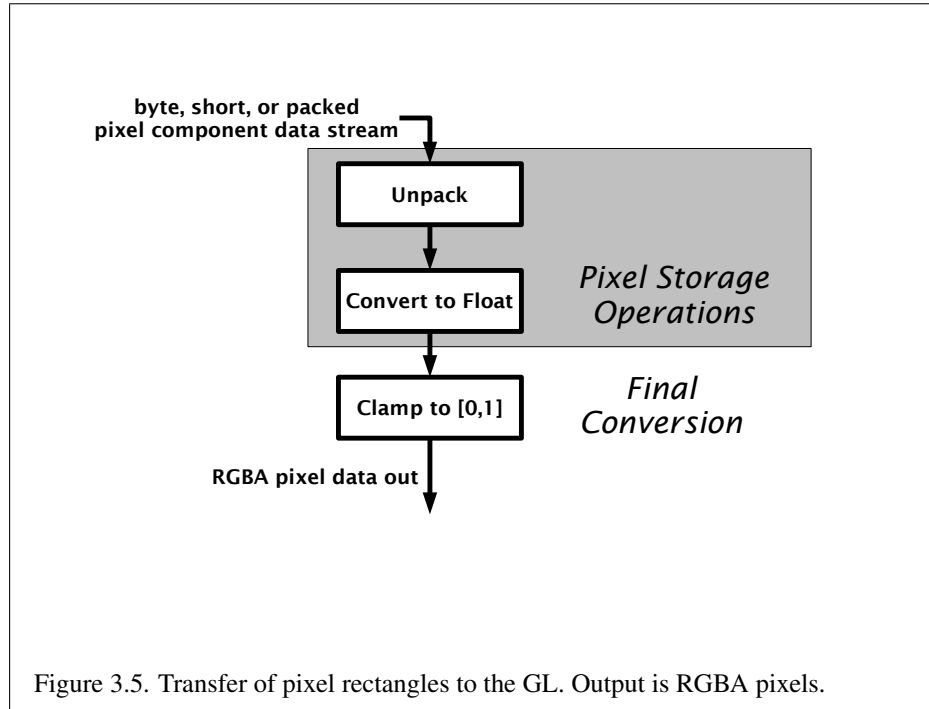
Table 3.1: **PixelStore** parameters pertaining to **TexSubImage2D**.

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 several GL data types, specified by *type*. The correspondence between the *type*



<i>type</i> 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: **TexSubImage2D** and **ReadnPixels** *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. **ReadnPixels** accepts only a subset of these types (see section 4.3.1).

Format Name	Element Meaning and Order	Target Buffer
RED	R	Color
RG	R, G	Color
RGB	R, G, B	Color
RGBA	R, G, B, A	Color

Table 3.3: **TexSubImage2D** and **ReadnPixels** formats. The second column gives a description of and the number and order of elements in a group. **ReadnPixels** 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 row pointed to by the *data* pointer passed to **TexSubImage2D**. The number of

Format	Type	Bytes per Pixel	Internal Format
RGBA	UNSIGNED_BYTE	4	RGBA8
RGB	UNSIGNED_BYTE	3	RGB8
RG	UNSIGNED_BYTE	2	RG8
RED	UNSIGNED_BYTE	1	R8
RGBA	UNSIGNED_SHORT_4_4_4_4	2	RGBA4
RGBA	UNSIGNED_SHORT_5_5_5_1	2	RGB5_A1
RGB	UNSIGNED_SHORT_5_6_5	2	RGB565

Table 3.4: Valid pixel format, type, and sized internal format.

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  $N$ th row is indicated by

$$p + Nk \quad (3.9)$$

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

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

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 `GLubyte`, of an element. If the number of bits per element is not 1, 2, 4, or 8 times the number of bits in a `GLubyte`, 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.

Components are packed with the first component in the most significant bits of the bitfield, and successive component occupying progressively less significant

<i>type</i> 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.

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:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1st Component					2nd					3rd					

UNSIGNED\_SHORT\_4\_4\_4\_4:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1st Component				2nd				3rd				4th			

UNSIGNED\_SHORT\_5\_5\_5\_1:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1st Component					2nd					3rd				4th	

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.

### 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 2.11.3) 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 3.7.6) 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 **TexSubImage2D** 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 TexStorage2D( enum target, int level,  
int internalformat, sizei width, sizei height );
```

is provided for specifying the properties of all levels of a two-dimensional texture at once. Once a texture is specified, the format and dimensions of all levels becomes immutable. The contents of the images and the parameters can still be modified. Such a texture is referred to as an *immutable-format* texture. The immutability status of a texture can be determined by calling **GetTexParameter** with *pname* `TEXTURE_IMMUTABLE_FORMAT`. Cube map textures are not supported.

If the command is successful, `TEXTURE_IMMUTABLE_FORMAT` becomes `TRUE`. If *internalformat* is a compressed texture format, then references to **TexSubImage2D** should be replaced by **CompressedTexSubImage2D**, with *format*, *type* and *data* replaced by any valid *imageSize* and *data*.

**TexStorage2D** will generate the following errors:

- An `INVALID_OPERATION` error is generated if zero is bound to *target*.
- An `INVALID_VALUE` error is generated if *width*, *height*, or *levels* are less than 1.
- An `INVALID_ENUM` error is generated if *internalformat* is not one of the sized internal formats listed in table 3.4.
- An `INVALID_OPERATION` error is generated if *levels* is greater than  $\text{floor}(\log_2(\max(\text{width}, \text{height}))) + 1$ .
- An `INVALID_OPERATION` error is generated if *levels* is not one and *width* or *height* is not a power of two.
- After a successful call to **TexStorage2D**, an `INVALID_OPERATION` error is generated if **TexStorage2D** is called with the same texture, even if it does not affect the dimensions or format.

If an error is generated then the command will have no effect. After a successful call to any `TexStorage` command, no further changes to the dimensions or format of the texture object may be made. Other commands may only alter the texel values and texture parameters.

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 **TexStorage2D** 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 *levels* argument to **TexStorage2D** 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).

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.

### 3.7.2 Texture Image Specification Commands

A two-dimensional texture consists of a single two-dimensional texture image. 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 **TexStorage2D**.

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 (or derived from) *internalformat*. Table 3.4 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. Specifying a combination of values for *format*, *type*, and *internalformat* that is not listed as a valid combination in table 3.4 generates the error `INVALID_OPERATION`. Rectangular subregions of existing texture images may be respecified.

The command

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

can specify an entire region and also respecify 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* argument of **TexSubImage2D** must be `TEXTURE_2D`.

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* and *height*, match the corresponding arguments to **TexStorage2D**, meaning that they accept the same values, and have the same meanings.

**TexSubImage2D** arguments *format* and *type* must be a valid combination from table 3.4 and correspond to the *internalformat* value used for **TexStorage2D**.

Arguments *xoffset* and *yoffset* of **TexSubImage2D** specify the lower left texel coordinates of a *width*-wide by *height*-high rectangular subregion of the texture array address. 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  $n$ th pixel group is assigned to the texel with internal integer coordinates  $[i, j]$ , where

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

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

### 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. Online texture compression is not supported. Instead applications must use an offline compressor and provide a pre-compressed texture in binary format. 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 **TexStorage2D** 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 CompressedTexSubImage2D( enum target, int level,
                             int xoffset, int yoffset, sizei width, sizei height,
                             enum format, sizei imageSize, void *data );
```

can specify an entire region and also respecify a rectangular subregion 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*.

Compressed texture images are treated as an array of *imageSize* ubytes 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.



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.

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. 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`. *pname* is a symbolic constant indicating the parameter to be set; the possible constants and corresponding parameters are summarized in table 3.8. 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.

#### 3.7.5 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.

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 3.8: Texture parameters and their values.

**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  $\lfloor \cdot \rfloor$  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`.

**3.7.6 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

$$\lambda(x, y) = \log_2[\rho(x, y)] \quad (3.11)$$

If  $\lambda(x, y)$  is less than or equal to the constant  $c$  (described below in section 3.7.7) 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,  $\rho$  is given at a fragment with window coordinates  $(x, y)$  by

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

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

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

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  $\rho$  with a function  $f(x, y)$  subject to these conditions:

1.  $f(x, y)$  is continuous and monotonically increasing in each of  $|\partial u/\partial x|$ ,  $|\partial u/\partial y|$ ,  $|\partial v/\partial x|$ ,  $|\partial v/\partial y|$ ,
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  $\lambda$  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

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

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

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

When `TEXTURE_MIN_FILTER` is `LINEAR`, a  $2 \times 2$  square of texels in the level zero array is selected. This square is obtained by first wrapping texture coordinates as described in section 3.7.5, then computing

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

and

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

Then

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

and

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

Let

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

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

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

The texture value  $\tau$  is found as

$$\tau = (1 - \alpha)(1 - \beta)\tau_{i_0j_0} + \alpha(1 - \beta)\tau_{i_1j_0} + (1 - \alpha)\beta\tau_{i_0j_1} + \alpha\beta\tau_{i_1j_1} \quad (3.16)$$

where  $\tau_{ij}$  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 implementation specific. 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

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

until the last array is reached with dimension  $1 \times 1$ .

Each array in a mipmap is defined using **TexSubImage2D** or **CompressedTexSubImage2D**; 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 3.7.9.

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  $\lambda$  at which the transition from minification to magnification occurs (since this discussion pertains to minification, we are concerned only with values of  $\lambda$  where  $\lambda > c$ ).

For mipmap filters NEAREST\_MIPMAP\_NEAREST and LINEAR\_MIPMAP\_NEAREST, the  $d$ th mipmap array is selected, where

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

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

$$d_1 = \begin{cases} q, & \lambda \geq q \\ \lfloor \lambda \rfloor, & \text{otherwise} \end{cases} \quad (3.18)$$

$$d_2 = \begin{cases} q, & \lambda \geq q \\ d_1 + 1, & \text{otherwise} \end{cases} \quad (3.19)$$

The rules for NEAREST or LINEAR filtering are then applied to each of the selected arrays, yielding two corresponding texture values  $\tau_1$  and  $\tau_2$ . The final texture value is then found as

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

### 3.7.7 Texture Magnification

When  $\lambda$  indicates magnification, the value assigned to TEXTURE\_MAG\_FILTER determines how the texture value is obtained. There are two possible values for TEXTURE\_MAG\_FILTER: NEAREST and LINEAR. NEAREST behaves exactly as NEAREST for TEXTURE\_MIN\_FILTER (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  $c = 0$ .

### 3.7.8 Texture Framebuffer Attachment

The texture values are considered implementation specific 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 inclusive range from zero to last mip-level.

### 3.7.9 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 are 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 3.7.6) 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 3.7.6.
- Each dimension of the level zero array is positive.

#### Effects of Completeness on Texture Application

Texture lookups performed in vertex and fragment shaders are affected by completeness of the texture being sampled as described in sections 2.11.3 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.

### 3.7.10 Mipmap Generation

Mipmaps can be generated with the command

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

*target* is the target, which must be TEXTURE\_2D.

**GenerateMipmap** computes a complete set of mipmap arrays (as defined in section 3.7.9) 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 3.7.9.

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.

If either the width or height of the level zero array are not a power of 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.

### 3.7.11 Texture State

The state necessary for texture can be divided into two categories. First, there is the mipmap array. Each array has associated with it a width and height, an integer describing the internal format of the image, an integer value describing the type of each of the components, a boolean describing whether the image is compressed or not, and an integer size of a compressed image. Each initial texture array is null (zero width and height). Next, there is a set of texture properties; it 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`.

### 3.7.12 Texture Objects

The name space for texture objects is the unsigned integers, with zero reserved by the GL.

A texture object is created by *binding* a texture object name obtained by calling **GenTextures** to `TEXTURE_2D`. Texture objects cannot be deleted.

The binding is effected by calling

```
void BindTexture( enum target, uint texture );
```

with *target* set to `TEXTURE_2D` and *texture* set to the texture object name obtained from **GenTextures**. The resulting texture object is a new state vector, comprising all the state values listed in section 3.7.11, set to the same initial values. Calling **BindTexture** with a texture object name not obtained by **GenTextures** will result in a `INVALID_OPERATION` error.

**BindTexture** may also be used to bind an existing texture object. 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` has a two-dimensional texture state vector associated with it. In order that access to the initial texture not be lost, it is treated as texture object whose name is 0. The initial two-dimensional texture is therefore operated upon, queried, and applied as `TEXTURE_2D 0` is bound to the corresponding targets.

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`.

## 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.

Fragment shaders are defined by pre-compiled shader binary code, in the same way as described for vertex shaders in section 2.11.

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, fragment shader execution does not happen.

### 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 accessed by a fragment shader. These built-in varying variables include the fragment's position, eye  $z$  coordinate, and front-facing flag.

A vertex shader may define one or more *varying* variables (see section 2.11.2 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  $\tau$  in the manner described in sections 3.7.6 and 3.7.7, and converts it to a texture source color  $C_s$  according to table 3.9. The GL returns a four-component vector  $(R_s, G_s, B_s, A_s)$  to the fragment shader. For the purposes of level-of-detail calculations, the derivatives  $\frac{du}{dx}$ ,  $\frac{du}{dy}$ ,  $\frac{dv}{dx}$ ,  $\frac{dv}{dy}$ ,  $\frac{dw}{dx}$  and  $\frac{dw}{dy}$  may be approximated by a differencing algorithm as detailed in section 8.8 of the OpenGL 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 associated texture object is not complete, as defined in sections 3.7.1 and 3.7.9,

Texture Base Internal Format	Texture source color ( $R_s, G_s, B_s$ )	Texture source alpha $A_s$
RED	( $R_t, 0, 0$ )	1
RG	( $R_t, G_t, 0$ )	1
RGB	( $R_t, G_t, B_t$ )	1
RGBA	( $R_t, G_t, B_t$ )	$A_t$

Table 3.9: Correspondence of filtered texture components to texture source color components. The values  $R_t$ ,  $G_t$ ,  $B_t$ , and  $A_t$  are respectively the red, green, blue, and alpha components of the filtered texture value  $\tau$ .

- A two-dimensional sampler is called, the minification filter is not one that requires a mipmap (neither `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 3.7.6), and either the texture wrap mode is not `CLAMP_TO_EDGE`, or the minification filter is neither `NEAREST` nor `LINEAR`.

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 2.13).

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]`<sup>2</sup>. 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 pre-processing, it contains a statement that would write to the variable, whether or not run-time flow of control will cause that statement to be executed.

---

<sup>2</sup>`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 and application-created framebuffer objects. Every OpenGL SC context has at most one 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 implementa-

tion or context may not provide depth or stencil buffers<sup>1</sup>. If no default framebuffer is associated with the GL context, the framebuffer is incomplete except when a framebuffer object is bound. (see sections 4.4.1 and 4.4.5)

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 uninitialized.

## 4.1 Per-Fragment Operations

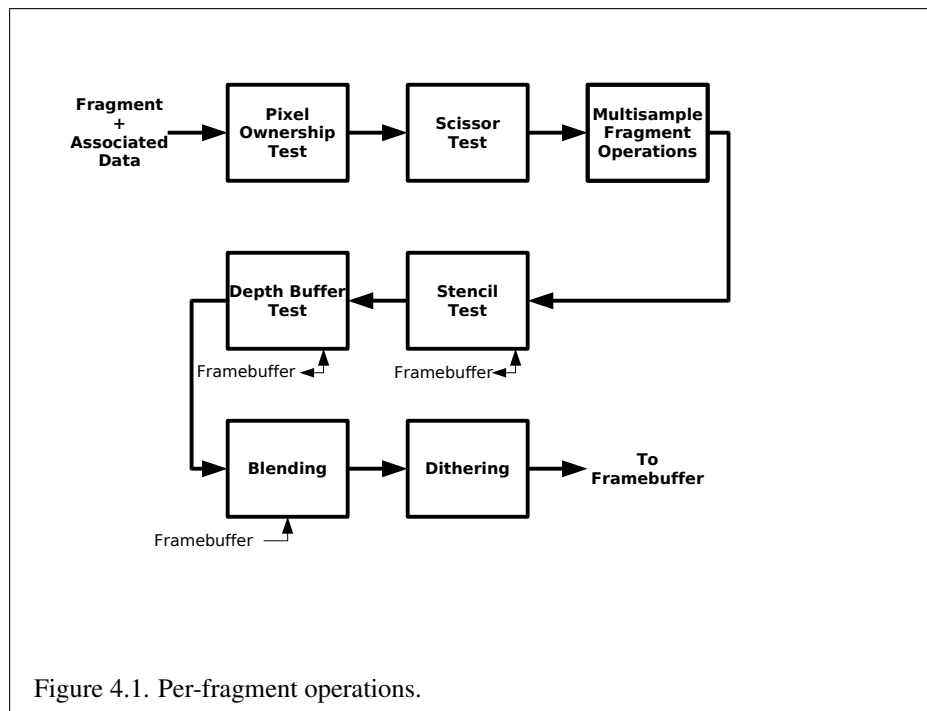
A fragment produced by rasterization with window coordinates of  $(x_w, y_w)$  modifies the pixel in the framebuffer at that location based on a number of parameters and conditions. We describe these modifications and tests, diagrammed in Figure 4.1, in the order in which they are performed.

### 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.

---

<sup>1</sup>However, an OpenGL SC implementation must support at least one config with a depth bit depth of 16 or higher and a stencil bit depth of 8 or higher





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

#### 4.1.2 Scissor Test

The scissor test determines if  $(x_w, y_w)$  lies within the scissor rectangle defined by four values. These values are set with

```
void Scissor( int left, int bottom, sizei width,
              sizei height );
```

If  $left \leq x_w < left + width$  and  $bottom \leq y_w < bottom + height$ , then the scissor test passes. Otherwise, the test fails and the fragment is discarded. The test is enabled or disabled using **Enable** or **Disable** using the constant `SCISSOR_TEST`. When disabled, it is as if the scissor test always passes. If either *width* or *height* is less than zero, then the error `INVALID_VALUE` is generated. The state required consists of four integer values and a bit indicating whether the test is enabled or disabled. In the initial state  $left = bottom = 0$ ; *width* and *height* are determined by the size of the GL window. Initially, the scissor test is disabled. If the default framebuffer is bound but no default framebuffer is associated with the GL context (see chapter 4), then width and height are initially set to zero.

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

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

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

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

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

Table 4.1: RGB and alpha blend equations.

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

### Blend Functions

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

```
void BlendFuncSeparate( enum srcRGB, enum dstRGB,
                        enum srcAlpha, enum dstAlpha );
void BlendFunc( enum src, enum dst );
```

**BlendFuncSeparate** arguments *srcRGB* and *dstRGB* determine the source and destination RGB blend functions, respectively, while *srcAlpha* and *dstAlpha* determine the source and destination alpha blend functions. **BlendFunc** argument *src* determines both RGB and alpha source functions, while *dst* determines both RGB and alpha destination functions.

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 <sup>1</sup>	$(f, f, f)$ <sup>2</sup>	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.

<sup>1</sup> SRC\_ALPHA\_SATURATE is valid only for source RGB and alpha blending functions.

<sup>2</sup>  $f = \min(A_s, 1 - A_d)$ .

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( float red, float green, float blue,
                  float alpha );
```

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  $(R, G, B, A) = (0, 0, 0, 0)$ . 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*, where *s* is the number of bits in the stencil buffer, specify a mask. Where a 1 appears in this mask, the corresponding bit in the stencil buffer is written; where a 0 appears, the bit is not written.

The *face* parameter of **StencilMaskSeparate** can be FRONT, BACK, or FRONT\_AND\_BACK and indicates whether the front or back stencil mask state is affected. **StencilMask** sets both front and back stencil mask state to identical values.

Fragments generated by front facing primitives use the front mask and fragments generated by back facing primitives use the back mask (see section 4.1.4). The clear operation always uses the front stencil write mask when clearing the stencil buffer.

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

### Fine Control of Multisample Buffer Updates

When the value of SAMPLE\_BUFFERS is one, **ColorMask**, **DepthMask**, and **StencilMask** 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(float r, float g, float b, float 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(float d);
```

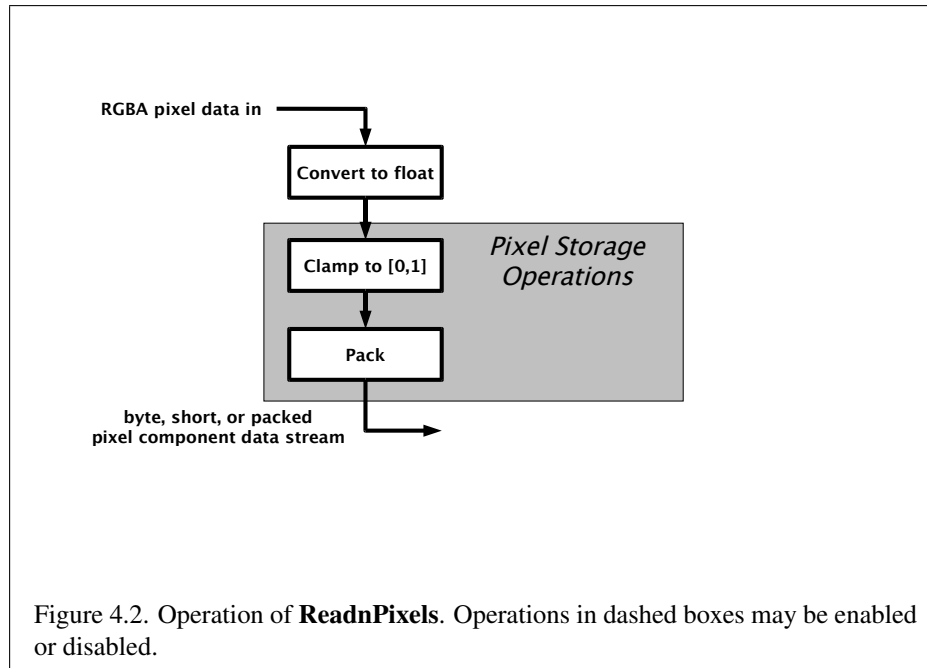
takes a value that is clamped to the range  $[0, 1]$  and converted to fixed-point according to the rules for a window  $z$  value given in section 2.13.1. Similarly,

```
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.



### 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 **ReadnPixels** commands, as described below. Pixels may also be copied from client memory to texture images in the GL using the **TexSubImage2D** command, 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 **ReadnPixels**.

Pixels are read using

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

The arguments after *x* and *y* to **ReadnPixels** 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. 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. The maximum number of bytes that may be written into *data* is specified by *bufSize*. Unsupported combinations of *format* and *type* will generate an `INVALID_OPERATION` error. The pixel storage modes that apply to **ReadnPixels** 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.

**ReadnPixels** 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 *i*th pixel in the *j*th 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, **ReadnPixels** 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.

<i>type</i> 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 **TexSubImage2D**. That is, the  $i$ th group of the  $j$ th row (corresponding to the  $i$ th pixel in the  $j$ th row) is placed in memory just where the  $i$ th group of the  $j$ th row would be taken from for **TexSubImage2D**. 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 **RED**, 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 SC renders into (and reads values from) a framebuffer. OpenGL SC defines two classes of framebuffers: window-system-provided framebuffers and application-created framebuffers.

By default, OpenGL SC 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, cannot be changed by OpenGL SC, nor can the window-system-provided framebuffer itself, or its images, be deleted by OpenGL SC.

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 SC provides a mechanism to support *off-screen* rendering. Further, by allowing the images of a texture to be attached to a framebuffer, OpenGL SC 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 SC to refer to the default framebuffer. A framebuffer object is created by binding a framebuffer object name obtained by calling **GenFramebuffers** 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 framebuffer object name obtained from **GenFramebuffers**. 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. Calling **BindFramebuffer** with a framebuffer object name not obtained by **GenFramebuffers** will result in a `INVALID_OPERATION` error. Framebuffer objects cannot be deleted.

**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 SC 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 **ReadnPixels**.

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, if there is one. 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 SC 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.

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 SC.

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 SC provides the methods described below to allocate 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 SC. A renderbuffer object is created by binding a renderbuffer object name obtained by calling **GenRenderbuffers** to `RENDERBUFFER`. The binding is effected by calling

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

with *target* set to `RENDERBUFFER` and *renderbuffer* set to the renderbuffer object name obtained from **GenRenderBuffers**. 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. Calling **BindRenderbuffer** with a renderbuffer object name not obtained by **GenRenderbuffers** will result in a `INVALID_OPERATION` error. Renderbuffer objects cannot be deleted.

**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 SC 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`.

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 SC is unable to create a data store of the requested size, the error `OUT_OF_MEMORY` is generated and the renderbuffer storage is not allocated.

An OpenGL SC 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. The width, height, and internal format cannot be changed once they are established. Attempting to change the width, height, or internal format once established will result in an `INVALID_OPERATION` error. 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.

### Attaching Texture Images to a Framebuffer

OpenGL SC supports using the rendered contents of the framebuffer as the images of a texture object through directly rendering 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 name an existing texture object with a target of *textarget*. 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 `TEXTURE_2D`.

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*. 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.

#### 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, resulting behavior is implementation specific. This section describes *rendering feedback loops* (see section 3.7.6) 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 implementation specific results, as described at the end of section 3.7.6. Conditions resulting in such implementation specific behavior are defined in more detail below.

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 en-

abled for texturing. Doing so could lead to the creation of a *rendering feedback loop* between the writing of pixels by OpenGL SC 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 implementation specific. The values of texture samples may be implementation specific as well, as described under “Rendering Feedback Loops” in section 3.7.6.

Specifically, the values of rendered fragments are implementation specific 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.

#### 4.4.5 Framebuffer Completeness

A framebuffer must be framebuffer complete to effectively be used as the draw or read framebuffer of the GL. The default framebuffer is always complete if it exists; however, if no default framebuffer exists (no window system-provided drawable is associated with the GL context), it is deemed to be incomplete.

Sized Internal Format	Renderable Type	<i>R</i> bits	<i>G</i> bits	<i>B</i> bits	<i>A</i> bits	<i>D</i> bits	<i>S</i> 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.

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 SC framebuffer rendering operations and the source for OpenGL SC 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.

### 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 all the following conditions are true:

- If *target* is the default framebuffer, the default frame buffer exists.  
`FRAMEBUFFER_UNDEFINED`
- All framebuffer attachment points are *framebuffer attachment complete*.  
`FRAMEBUFFER_INCOMPLETE_ATTACHMENT`
- There is at least one image attached to the framebuffer.  
`FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT`
- 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**.
- Associating a different window system-provided drawable, or no drawable, with the default framebuffer using a window system binding API such as EGL.

Although OpenGL SC 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 **DrawRangeElements**, as well as commands that read the framebuffer such as **ReadnPixels**, 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 implementation specific.

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 SC is unable to create a data store of the required size when calling **RenderbufferStorage**.

## 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

State variables for which any command other than **IsEnabled** is listed as the query command can be obtained only by using that command. This ensures that no data

conversions occur for queries. State variables for which `IsEnabled` is listed may also be queried by `GetBooleanv`.

Unless otherwise indicated, multi-valued state variables return their multiple values in the same order as they are given as arguments to the commands that set them. For instance, the two **DepthRange** parameters are returned in the order *n* followed by *f*.

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 must be `TEXTURE_2D`, indicating the currently bound two-dimensional 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 3.8.

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`, or `FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL`.

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.
- 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 String Queries

The command

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

returns a pointer to a static string describing some aspect of the current GL connection <sup>1</sup>. 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 SC N.M vendor-specific information"
```

The `SHADING_LANGUAGE_VERSION` string is laid out as follows:

```
"OpenGL SC 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

---

<sup>1</sup>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.

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.

### 6.1.5 Program Queries

State stored program objects can be queried by commands that accept program object names. These commands will generate the error `INVALID_VALUE` if the provided name is not the name of a program object. If an error is generated, variables used to hold return values are not modified.

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 `LINK_STATUS`, `TRUE` is returned if the shader was last loaded successfully, and `FALSE` is returned otherwise.

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 GetnUniformfv(uint program, int location, sizei
                   bufSize, float *params);
void GetnUniformiv(uint program, int location, sizei
                   bufSize, int *params);
```

return the value or values of the uniform at location *location* for program object *program* in the array *params*. The maximum number of bytes that may be written into *params* is specified by *bufSize*. The type of the uniform at *location* determines the number of values returned. The error `INVALID_OPERATION` is generated if *program* has not been loaded successfully, or if *location* is not a valid location for *program*. In order to query the values of an array of uniforms, a **GetnUniform\*** 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 **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**. State variables for which any command other than **IsEnabled** is listed as the query command can be obtained only by using that command.

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

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 textures, where only the selected texture or texture parameter is returned.

Get value	Type	Get Cmd	Initial Value	Description	Sec.
VERTEX_ATTRIB_ARRAY_ENABLED	$8 * \times B$	<b>GetVertexAttrib</b>	<i>False</i>	Vertex attrib array enable	<b>2.9</b>
VERTEX_ATTRIB_ARRAY_SIZE	$8 * \times Z$	<b>GetVertexAttrib</b>	4	Vertex attrib array size	<b>2.9</b>
VERTEX_ATTRIB_ARRAY_STRIDE	$8 * \times Z^+$	<b>GetVertexAttrib</b>	0	Vertex attrib array stride	<b>2.9</b>
VERTEX_ATTRIB_ARRAY_TYPE	$8 * \times Z_4$	<b>GetVertexAttrib</b>	FLOAT	Vertex attrib array type	<b>2.9</b>
VERTEX_ATTRIB_ARRAY_NORMALIZED	$8 * \times B$	<b>GetVertexAttrib</b>	<i>False</i>	Vertex attrib array normalized	<b>2.9</b>
VERTEX_ATTRIB_ARRAY_POINTER	$8 * \times Y$	<b>GetVertexAttribPointer</b>	NULL	Vertex attrib array pointer	<b>2.9</b>
ARRAY_BUFFER_BINDING	$Z^+$	<b>GetIntegerv</b>	0	current buffer binding	<b>2.10</b>
ELEMENT_ARRAY_BUFFER_BINDING	$Z^+$	<b>GetIntegerv</b>	0	element array buffer binding	<b>2.10.2</b>
VERTEX_ATTRIB_ARRAY_BUFFER_BINDING	$8 * \times Z^+$	<b>GetVertexAttribiv</b>	0	Attribute array buffer binding	<b>2.10</b>

Table 6.2. Vertex Array Data

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
BUFFER_SIZE	$n \times Z^+$	<b>GetBufferParameteriv</b>	0	buffer data size	<b>2.10</b>
BUFFER_USAGE	$n \times Z_3$	<b>GetBufferParameteriv</b>	STATIC_DRAW	buffer usage pattern	<b>2.10</b>

Table 6.3. Buffer Object State

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
VIEWPORT	$4 \times Z$	<b>GetIntegerv</b>	see <b>2.13.1</b>	Viewport origin & extent	<b>2.13.1</b>
DEPTH_RANGE	$2 \times R^+$	<b>GetFloatv</b>	0,1	Depth range near & far	<b>2.13.1</b>

Table 6.4. Transformation state

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

Table 6.5. Rasterization



Get value	Type	Get Cmd	Initial Value	Description	Sec.
SAMPLE.ALPHA.TO.COVERAGE	$B$	<b>IsEnabled</b>	<i>False</i>	Modify coverage from alpha	4.1.3
SAMPLE.COVERAGE	$B$	<b>IsEnabled</b>	<i>False</i>	Mask to modify coverage	4.1.3
SAMPLE.COVERAGE.VALUE	$R^+$	<b>GetFloatv</b>	1	Coverage mask value	4.1.3
SAMPLE.COVERAGE.INVERT	$B$	<b>GetBooleanv</b>	<i>False</i>	Invert coverage mask value	4.1.3

Table 6.6. Multisampling

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
TEXTURE.BINDING_2D	$8 * Z^+$	<b>GetIntegerv</b>	0	Texture object bound to TEXTURE_2D	<b>3.7.12</b>

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

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
TEXTURE_MIN_FILTER	$n \times Z_6$	<b>GetTexParameter</b>	see 3.7	Texture minification function	3.7.6
TEXTURE_MAG_FILTER	$n \times Z_2$	<b>GetTexParameter</b>	see 3.7	Texture magnification function	3.7.7
TEXTURE_WRAP_S	$n \times Z_2$	<b>GetTexParameter</b>	REPEAT	Texcoord $s$ wrap mode	3.7.5
TEXTURE_WRAP_T	$n \times Z_2$	<b>GetTexParameter</b>	REPEAT	Texcoord $t$ wrap mode	3.7.5
TEXTURE_IMMUTABLE_FORMAT	$n \times B$	<b>GetTexParameter</b>	FALSE	Size and format im- mutable	3.7.1

Table 6.8. Textures (state per texture object)

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
ACTIVE_TEXTURE	$Z_{8*}$	<b>GetIntegerv</b>	TEXTURE0	Active texture unit selector	2.8

Table 6.9. Texture Environment and Generation

Get value	Type	Get Cmd	Initial Value	Description	Sec.
SCISSOR.TEST	$B$	<b>IsEnabled</b>	<i>False</i>	Scissoring enabled	4.1.2
SCISSOR_BOX	$4 \times Z$	<b>GetIntegerv</b>	see 4.1.2	Scissor box	4.1.2
STENCIL.TEST	$B$	<b>IsEnabled</b>	<i>False</i>	Stenciling enabled	4.1.4
STENCIL.FUNC	$Z_8$	<b>GetIntegerv</b>	ALWAYS	Front stencil function	4.1.4
STENCIL.VALUE.MASK	$Z^+$	<b>GetIntegerv</b>	1's	Front stencil mask	4.1.4
STENCIL.REF	$Z^+$	<b>GetIntegerv</b>	0	Front stencil reference value	4.1.4
STENCIL.FAIL	$Z_8$	<b>GetIntegerv</b>	KEEP	Front stencil fail action	4.1.4
STENCIL.PASS.DEPTH.FAIL	$Z_8$	<b>GetIntegerv</b>	KEEP	Front stencil depth buffer fail action	4.1.4
STENCIL.PASS.DEPTH.PASS	$Z_8$	<b>GetIntegerv</b>	KEEP	Front stencil depth buffer pass action	4.1.4
STENCIL.BACK.FUNC	$Z_8$	<b>GetIntegerv</b>	ALWAYS	Back stencil function	4.1.4
STENCIL.BACK.VALUE.MASK	$Z^+$	<b>GetIntegerv</b>	1's	Back stencil mask	4.1.4
STENCIL.BACK.REF	$Z^+$	<b>GetIntegerv</b>	0	Back stencil reference value	4.1.4
STENCIL.BACK.FAIL	$Z_8$	<b>GetIntegerv</b>	KEEP	Back stencil fail action	4.1.4
STENCIL.BACK.PASS.DEPTH.FAIL	$Z_8$	<b>GetIntegerv</b>	KEEP	Back stencil depth buffer fail action	4.1.4
STENCIL.BACK.PASS.DEPTH.PASS	$Z_8$	<b>GetIntegerv</b>	KEEP	Back stencil depth buffer pass action	4.1.4
DEPTH.TEST	$B$	<b>IsEnabled</b>	<i>False</i>	Depth buffer enabled	4.1.5
DEPTH.FUNC	$Z_8$	<b>GetIntegerv</b>	LESS	Depth buffer test function	4.1.5

Table 6.10. Pixel Operations

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
BLEND	<i>B</i>	<b>IsEnabled</b>	<i>False</i>	Blending enabled	4.1.6
BLEND.SRC.RGB (v1.1:BLEND.SRC)	<i>Z</i> <sub>15</sub>	<b>GetIntegerv</b>	ONE	Blending source RGB function	4.1.6
BLEND.SRC.ALPHA	<i>Z</i> <sub>15</sub>	<b>GetIntegerv</b>	ONE	Blending source A function	4.1.6
BLEND.DST.RGB (v1.1:BLEND.DST)	<i>Z</i> <sub>14</sub>	<b>GetIntegerv</b>	ZERO	Blending dest. RGB function	4.1.6
BLEND.DST.ALPHA	<i>Z</i> <sub>14</sub>	<b>GetIntegerv</b>	ZERO	Blending dest. A function	4.1.6
BLEND.EQUATION.RGB (v1.1: BLEND.EQUATION)	<i>Z</i> <sub>5</sub>	<b>GetIntegerv</b>	FUNC_ADD	RGB blending equation	4.1.6
BLEND.EQUATION.ALPHA	<i>Z</i> <sub>5</sub>	<b>GetIntegerv</b>	FUNC_ADD	Alpha blending equation	4.1.6
BLEND.COLOR	<i>C</i>	<b>GetFloatv</b>	ZERO	Constant blend color	4.1.6
DITHER	<i>B</i>	<b>IsEnabled</b>	<i>True</i>	Dithering enabled	4.1.7

Table 6.11. Pixel Operations (cont.)

Get value	Type	Get Cmd	Initial Value	Description	Sec.
COLOR.WRITEMASK	$4 \times B$	<b>GetBooleanv</b>	<i>True</i>	Color write enables; R, G, B, or A	4.2.2
DEPTH.WRITEMASK	$B$	<b>GetBooleanv</b>	<i>True</i>	Depth buffer enabled for writing	4.2.2
STENCIL.WRITEMASK	$Z^+$	<b>GetIntegerv</b>	1's	Front stencil buffer writemask	4.2.2
STENCIL.BACK.WRITEMASK	$Z^+$	<b>GetIntegerv</b>	1's	Back stencil buffer writemask	4.2.2
COLOR.CLEAR.VALUE	$C$	<b>GetFloatv</b>	0,0,0,0	Color buffer clear value (RGBA mode)	4.2.3
DEPTH.CLEAR.VALUE	$R^+$	<b>GetFloatv</b>	1	Depth buffer clear value	4.2.3
STENCIL.CLEAR.VALUE	$Z^+$	<b>GetIntegerv</b>	0	Stencil clear value	4.2.3

Table 6.12. Framebuffer Control

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
UNPACK_ALIGNMENT	$Z^+$	<b>GetIntegerv</b>	4	Value of UNPACK_ALIGNMENT	<b>3.6.1</b>
PACK_ALIGNMENT	$Z^+$	<b>GetIntegerv</b>	4	Value of PACK_ALIGNMENT	<b>4.3.1</b>

Table 6.13. Pixels



Get value	Type	Get Cmnnd	Initial Value	Description	Sec.
CURRENT_PROGRAM	$Z^+$	<b>GetIntegerv</b>	0	Name of current program object	<b>2.11.1</b>
LINK_STATUS	$B$	<b>GetProgramiv</b>	<i>False</i>	Last load attempt succeeded	<b>6.1.5</b>
-	$0 * \times Z$	<b>GetUniformLocation</b>	-	Location of active uniforms	<b>6.1.5</b>
	$512 * \times R$	<b>GetnUniform</b>	0	Uniform value	<b>2.11.2</b>
-	$0 * \times Z$	<b>GetAttribLocation</b>	-	Location of active generic attribute	<b>2.11.2</b>

Table 6.14. Program Object State

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
CURRENT_VERTEX_ATTRIB	$16 * R^4$	<b>GetVertexAttrib</b>	0,0,0,1	Generic vertex attribute	<b>2.8</b>

Table 6.15. Vertex Shader State

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
GENERATE_MIPMAP_HINT	$Z_3$	<b>GetIntegerv</b>	DONT_CARE	Mipmap generation hint	<b>5.2</b>

Table 6.16. Hints

Get value	Type	Get Cmnd	Minimum Value	Description	Sec.
SUBPIXEL_BITS	$Z^+$	<b>GetIntegerv</b>	4	Number of bits of sub-pixel precision in screen $x_w$ and $y_w$	<b>3</b>
MAX_TEXTURE_SIZE	$Z^+$	<b>GetIntegerv</b>	64	Maximum texture image dimension	<b>3.7.1</b>
MAX_VIEWPORT_DIMS	$2 \times Z^+$	<b>GetIntegerv</b>	see <b>2.13.1</b>	Maximum viewport dimensions	<b>2.13.1</b>
ALIASED_POINT_SIZE_RANGE	$2 \times R^+$	<b>GetFloatv</b>	1,1	Range (lo to hi) of aliased point sizes	<b>3.3</b>
ALIASED_LINE_WIDTH_RANGE	$2 \times R^+$	<b>GetFloatv</b>	1,1	Range (lo to hi) of aliased line widths	<b>3.4</b>
SAMPLE_BUFFERS	$Z^+$	<b>GetIntegerv</b>	0	Number of multisample buffers	<b>3.2</b>
SAMPLES	$Z^+$	<b>GetIntegerv</b>	0	Coverage mask size	<b>3.2</b>
COMPRESSED_TEXTURE_FORMATS	$0 * \times Z_{0*}$	<b>GetIntegerv</b>	-	Enumerated compressed texture formats	<b>3.7.3</b>
NUM_COMPRESSED_TEXTURE_FORMATS	$Z$	<b>GetIntegerv</b>	0	Number of enumerated compressed texture formats	<b>3.7.3</b>

Table 6.17. Implementation Dependent Values

Get value	Type	Get Cmnnd	Minimum Value	Description	Sec.
PROGRAM.BINARY_FORMATS	$nf \times Z$	<b>GetIntegerv</b>	-	Enumerated program binary formats	2.11.1
NUM.PROGRAM.BINARY_FORMATS	$Z$	<b>GetIntegerv</b>	0	Number of program binary formats	2.11.1
EXTENSIONS	$S$	<b>GetString</b>	-	Supported extensions	6.1.4
RENDERER	$S$	<b>GetString</b>	-	Renderer string	6.1.4
SHADING.LANGUAGE.VERSION	$S$	<b>GetString</b>	-	Shading Language version supported	6.1.4
VENDOR	$S$	<b>GetString</b>	-	Vendor string	6.1.4
VERSION	$S$	<b>GetString</b>	-	OpenGL version supported	6.1.4

Table 6.18. Implementation Dependent Values (cont.)

Get value	Type	Get Cmd	Minimum Value	Description	Sec.
MAX_VERTEX_ATTRIBS	$Z^+$	<b>GetIntegerv</b>	8	Number of active vertex attributes	<b>2.8</b>
MAX_VERTEX_UNIFORM_VECTORS	$Z^+$	<b>GetIntegerv</b>	128	Number of vectors for vertex shader uniform variables	<b>2.11.2</b>
MAX_VARYING_VECTORS	$Z^+$	<b>GetIntegerv</b>	8	Number of vectors for varying variables	<b>2.11.2</b>
MAX_COMBINED_TEXTURE_IMAGE_UNITS	$Z^+$	<b>GetIntegerv</b>	8	Total number of texture units accessible by the GL	<b>2.11.3</b>
MAX_VERTEX_TEXTURE_IMAGE_UNITS	$Z^+$	<b>GetIntegerv</b>	0	Number of texture image units accessible by a vertex shader	<b>2.11.3</b>
MAX_TEXTURE_IMAGE_UNITS	$Z^+$	<b>GetIntegerv</b>	8	Number of texture image units accessible by fragment processing	<b>2.11.3</b>
MAX_FRAGMENT_UNIFORM_VECTORS	$Z^+$	<b>GetIntegerv</b>	16	Number of vectors for fragment shader uniform variables	<b>3.8.1</b>
MAX_RENDERBUFFER_SIZE	$Z^+$	<b>GetIntegerv</b>	1	Maximum renderbuffer size	<b>4.4.3</b>

Table 6.19. Implementation Dependent Values (cont.)

Get value	Type	Get Cmnnd	Initial Value	Description	Sec.
$x\_BITS$	$Z^+$	<b>GetIntegerv</b>	-	Number of bits in $x$ color buffer component; $x$ is one of RED, GREEN, BLUE, or ALPHA	<b>4</b>
DEPTH_BITS	$Z^+$	<b>GetIntegerv</b>	-	Number of depth buffer planes	<b>4</b>
STENCIL_BITS	$Z^+$	<b>GetIntegerv</b>	-	Number of stencil planes	<b>4</b>
IMPLEMENTATION_COLOR_READ_TYPE	$Z^+$	<b>GetIntegerv</b>	-	Implementation preferred pixel <i>type</i>	<b>4.3.1</b>
IMPLEMENTATION_COLOR_READ_FORMAT	$Z^+$	<b>GetIntegerv</b>	-	Implementation preferred pixel <i>format</i>	<b>4.3.1</b>

Table 6.20. Implementation Dependent Pixel Depths

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
-	$n \times Z_6$	<b>GetError</b>	NO_ERROR	Current error code(s)	2.5
-	$n \times B$	-	<i>False</i>	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 <sup>+</sup>	<b>GetIntegerv</b>	0	Renderbuffer binding	4.4.3
RENDERBUFFER_WIDTH	Z <sup>+</sup>	<b>GetRenderbufferParameteriv</b>	0	Renderbuffer width	4.4.3
RENDERBUFFER_HEIGHT	Z <sup>+</sup>	<b>GetRenderbufferParameteriv</b>	0	Renderbuffer height	4.4.3
RENDERBUFFER_INTERNAL_FORMAT	Z <sup>+</sup>	<b>GetRenderbufferParameteriv</b>	RGBA4	Renderbuffer internal format	4.4.3
RENDERBUFFER_RED_SIZE	Z <sup>+</sup>	<b>GetRenderbufferParameteriv</b>	0	Renderbuffer red size	4.4.3
RENDERBUFFER_GREEN_SIZE	Z <sup>+</sup>	<b>GetRenderbufferParameteriv</b>	0	Renderbuffer green size	4.4.3
RENDERBUFFER_BLUE_SIZE	Z <sup>+</sup>	<b>GetRenderbufferParameteriv</b>	0	Renderbuffer blue size	4.4.3
RENDERBUFFER_ALPHA_SIZE	Z <sup>+</sup>	<b>GetRenderbufferParameteriv</b>	0	Renderbuffer alpha size	4.4.3
RENDERBUFFER_DEPTH_SIZE	Z <sup>+</sup>	<b>GetRenderbufferParameteriv</b>	0	Renderbuffer depth size	4.4.3
RENDERBUFFER_STENCIL_SIZE	Z <sup>+</sup>	<b>GetRenderbufferParameteriv</b>	0	Renderbuffer stencil size	4.4.3

Table 6.22. Renderbuffer State

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
FRAMEBUFFER_BINDING	$\mathbb{Z}^+$	<b>GetIntegerv</b>	0	Framebuffer binding	4.4.2
FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE	$n \times \mathbb{Z}_3$	<b>GetFramebufferAttachmentParameteriv</b>	NONE	Framebuffer object type	4.4.2
FRAMEBUFFER_ATTACHMENT_OBJECT_NAME	$n \times \mathbb{Z}^+$	<b>GetFramebufferAttachmentParameteriv</b>	0	Framebuffer object name	4.4.2
FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL	$n \times \mathbb{Z}^+$	<b>GetFramebufferAttachmentParameteriv</b>	0	Framebuffer texture level	4.4.2

Table 6.23. Framebuffer State

Get value	Type	Get Cmnd	Initial Value	Description	Sec.
CONTEXT_ROBUST_ACCESS	$B$	<b>GetBooleanv</b>	TRUE	Robust access enabled	<b>2.6</b>
GL_RESET_NOTIFICATION_STRATEGY	$Z_2$	<b>GetIntegerv</b>	LOSE_CONTEXT_ON_RESET	Reset notification behavior	<b>2.6</b>

Table 6.24. Robustness State

## Appendix A

# Invariance

The OpenGL SC 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 in a different color.
- 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 SC .

## 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 SC 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 SC 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 SC 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 SC 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. As a consequence of the chosen model, left- or right-handedness of any of the coordinate systems is not an implementation consideration, and applications using OpenGL-SC may choose to operate in either way,
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 coordinate transformations is identical when A, B, C, and D are specified). Then, when the fragments produced by rasterization of both polygons are taken together, each fragment intersecting the interior of the shared edge is produced exactly once.
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

# Shared Objects and Multiple Contexts

This appendix describes special considerations for objects shared between multiple OpenGL SC contexts, including how changes to shared objects are propagated between contexts. <sup>1</sup>

The *share list* of a context is the group of all contexts which share objects with that context.

### C.1 Sharing Contexts Between Different Versions of OpenGL SC

Implementations may or may not allow sharing between contexts implementing different OpenGL SC versions. However, implementation-dependent behavior may result when aspects and/or behaviors of such shared objects do not apply to, and/or are not described by more than one version.

### C.2 Sharing Objects Between Different Contexts in OpenGL SC

Implementations may choose to support sharing of any object such as vertex buffer objects, program objects, renderbuffer objects, and texture objects (except for the texture objects named zero).

---

<sup>1</sup>This appendix was entirely rewritten in version 2.0.25 of the OpenGL ES Specification, to match the same appendix in the OpenGL 4.1 Specification and add caveats regarding different treatment of framebuffer objects in OpenGL SC .

## C.3 Propagating Changes to Objects

GL objects contain two types of information, *data* and *state*. Collectively these are referred to below as the *contents* of an object. For the purposes of propagating changes to object contents as described below, data and state are treated consistently.

*Data* is information the GL implementation does not have to inspect, and does not have an operational effect. Currently, data consists of:

- Pixels in the framebuffer.
- The contents of textures and renderbuffers.
- The contents of buffer objects.

*State* determines the configuration of the rendering pipeline and the driver does have to inspect it.

In hardware-accelerated GL implementations, state typically lives in GPU registers, while data typically lives in GPU memory.

When the contents of an object *T* are changed, such changes are not always immediately visible, and do not always immediately affect GL operations involving that object. Changes may occur via any of the following means:

- State-setting commands, such as **TexParameter**.
- Data-setting commands, such as **TexSubImage\*** or **BufferSubData**.
- Data-setting through rendering to attached renderbuffers.

### C.3.1 Determining Completion of Changes to an object

The contents of an object *T* are considered to have been changed once a command such as described in section C.3 has completed. Completion of a command may be determined by calling **Finish**.

### C.3.2 Definitions

In the remainder of this section, the following terminology is used:

- An object *T* is *directly attached* to the current context if it has been bound to one of the context binding points. Examples include but are not limited to bound textures and current programs.

- *T* is *indirectly attached* to the current context if it is attached to another object *C*, referred to as a *container object*, and *C* is itself directly or indirectly attached. Examples include but are not limited to renderbuffers or textures attached to framebuffers and shaders attached to programs.
- An object *T* which is directly attached to the current context may be *re-attached* by re-binding *T* at the same bind point. An object *T* which is indirectly attached to the current context may be re-attached by re-attaching the container object *C* to which *T* is attached.

*Corollary:* re-binding *C* to the current context re-attaches *C* and its hierarchy of contained objects.

### C.3.3 Rules

The following rules must be obeyed by all GL implementations:

**Rule 1** *If the contents of an object T are changed in the current context while T is directly or indirectly attached, then all operations on T will use the new contents in the current context.*

Note: *The intent of this rule is to address changes in a single context only. The multi-context case is handled by the other rules.*

Note: *“Updates” via rendering are treated consistently with updates via GL commands.*

**Rule 2** *While a container object C is bound, any changes made to the contents of C’s attachments in the current context are guaranteed to be seen. To guarantee seeing changes made in another context to objects attached to C, such changes must be completed in that other context (see section C.3.1) prior to C being bound. Changes made in another context but not determined to have completed as described in section C.3.1, or after C is bound in the current context, are not guaranteed to be seen.*

**Rule 3** *Changes to the contents of shared objects are not automatically propagated between contexts. If the contents of a shared object T are changed in a context other than the current context, and T is already directly or indirectly attached to the current context, any operations on the current context involving T via those attachments are not guaranteed to use its new contents.*

**Rule 4** *If the contents of an object T are changed in a context other than the current context, T must be attached or re-attached to at least one binding point in the*

*current context in order to guarantee that the new contents of T are visible in the current context.*

Note: “Attached or re-attached” means either attaching an object to a binding point it wasn’t already attached to, or attaching an object again to a binding point it was already attached.

Example: If a texture image is bound to multiple texture bind points and the texture is changed in another context, re-binding the texture at any one of the texture bind points is sufficient to cause the changes to be visible at all texture bind points.

## Appendix D

### Version 2.0

OpenGL SC 2.0 is **not** compatible with the prior version (OpenGL SC 1.0). It introduces programmable vertex and fragment shaders, but removes the fixed-function pipeline.

## Appendix E

# Extension Registry, Header Files, and Extension Naming Conventions

### E.1 Extension Registry

Many extensions to the OpenGL SC API have been defined by vendors, groups of vendors, and the Khronos OpenGL SC Working Group. In order not to compromise the readability of the OpenGL SC Specification, such extensions are not integrated into the core language; instead, they are made available online in the *OpenGL SC Extension Registry*.

Extensions are documented as changes to a particular version of the Specification. The Registry is available on the World Wide Web at URL

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

### E.2 Header Files

OpenGL SC 2.0 provides two header files.

`<GLSC2/glsc2.h>` defines APIs for core OpenGL SC 2.0.

`<GLSC2/glsc2ext.h>` defines APIs for all registered OGLSC, EXT, and vendor extensions compatible with OpenGL SC 2.0 (some extensions are only compatible with OpenGL SC 1.x).

## E.3 OGLSC Extensions

OpenGL SC extensions that have been approved by the Khronos OpenGL SC Working Group are summarized in this section. These extensions are not required to be supported by a conformant OpenGL SC implementation, but are expected to be widely available; they define functionality that is likely to move into the required feature set in a future version of the Specification.

### E.3.1 Naming Conventions

To distinguish OGLSC extensions from core OpenGL SC features and from vendor-specific extensions, the following naming conventions are used:

- A unique *name string* of the form "GL\_OGLSC\_*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*OGLSC**
- All enumerants defined by the extension will have names of the form *NAME\_OGLSC*.

## E.4 Vendor and EXT Extensions

Vendor extensions (not approved by Khronos) use the same naming conventions as OGLSC extensions, but with a different tag replacing **OGLSC**. The following policies should always be followed when defining and shipping vendor extensions:

- A vendor tag will be assigned to a vendor on request to the Khronos Registrar, if one is not already defined.
- This vendor tag must be used consistently in the extension name strings and the corresponding function and enumerant names for extensions defined solely by that vendor.
- Numeric values assigned to enumerants must follow the guidelines described in the OpenGL SC Extension Registry. Reserved blocks of enumerant values will be assigned to vendors on request, following the process defined in the Registry.

- The reserved tag **EXT** may be used instead of a company-specific tag if multiple vendors agree to ship the same vendor extension.
- If a vendor decides to ship another vendor's extension at a later date, the original extension name and vendor tag should still be used, unless both vendors agree to promote that extension to an **EXT**.

An implementation exporting extension strings, or supporting function or enumerant names not following these naming guidelines, is not conformant.

Khronos strongly encourages vendors to submit full extension specifications to the OpenGL SC Extension Registry for publication, once they have finished defining the functionality in an extension. Extension writing guidelines, templates, and other process documents are also found in the Registry.



## Appendix F

# GLSL Limitations

### F.1 Overview

OpenGL SC 2.0 implementations are not required to support the full GLSL ES 1.00 specification. This section lists the features which are not fully supported in OpenGL SC 2.0. Features not listed in this section must be supported in their entirety. Within the GLSL ES specification, implementations are permitted to implement features beyond the minimal described in this section, without the use of an extension.

### F.2 Length of Shader Executable

The maximum length of the shader executable is implementation specific.

### F.3 Usage of Temporary Variables

The maximum number of variables is implementation specific.

### F.4 Control Flow

In general, control flow is limited to forward branching and to loops where the maximum number of iterations can easily be determined at compile time. Forward branching is allowed, both for constant and non-constant conditions. Therefore `if-then`, `if-then-else`, `break` and `continue` statements are permitted. Backward branching is only permitted for constant iteration loops as defined below. In the following section, loop indices are defined as all the non-constant variables

appearing in the branch condition expression in a loop. `for` loops are supported but with the following restrictions:

- There is one loop index.
- The loop index has type `int` or `float`.
- The `for` statement has the form:

```
for ( init-declaration ; condition ; expression ) statement
```

- `init-declaration` has the form:

```
type-specifier identifier = constant-expression
```

Consequently the loop variable cannot be a global variable.

- `condition` has the form

```
loop_index relational_operator constant_expression
```

where `relational_operator` is one of: `>`, `>=`, `<`, `<=`, `==`, or `!=`

- `for_header` has one of the following forms:

```
loop_index++
loop_index--
loop_index += constant_expression
loop_index -= constant_expression
```

- Within the body of the loop, the loop index is not statically assigned to nor is it used as the argument to a function `out` or `inout` parameter.

Support for `while` and `do-while` is not mandated.

## F.5 Indexing of Arrays, Vectors and Matrices

Definition:

`constant-index-expressions` are a superset of `constant-expressions`. `Constant-index-expressions` can include loop indices as defined in section F.4.

The following are `constant-index-expressions`:

- Constant expressions
- Loop indices as defined in section F.4
- Expressions composed of both of the above

When used as an index, a `constant-index-expression` must have integral type.

**Uniforms (excluding samplers)**

In the vertex shader, support for all forms of array indexing is mandated. In the fragment shader, support for indexing is only mandated for `constant-index-expressions`.

**Samplers**

GLSL ES 1.00 supports both arrays of samplers and arrays of structures which contain samplers. In both these cases, for OpenGL SC 2.0, support for indexing with a `constant-index-expression` is mandated but support for indexing with other values is not mandated.

**Attributes**

Support for indexing of matrices and vectors with `constant-index-expressions` is mandated. Support for indexing of matrices and vectors with other values is not mandated. Attribute arrays are disallowed by the specification.

**Varyings**

Support for indexing with a `constant-index-expression` is mandated. Support for indexing with other values is not mandated.

**Variables**

Support for indexing with a `constant-index-expression` is mandated. Support for indexing with other values is not mandated.

**Constants**

Support for indexing of matrices and vectors with `constant-index-expression` is mandated. Support for indexing of matrices and vectors with other values is not mandated. Constant arrays

are disallowed by the specification.

### Summary

The following array indexing functionality must be supported:

	Vertex Shaders	Fragment Shaders
Uniforms	Any integer	constant-index-expression
Attribute (vectors and matrices)	constant-index-expression	Not applicable
Varyings	constant-index-expression	constant-index-expression
Samplers	constant-index-expression	constant-index-expression
Variables	constant-index-expression	constant-index-expression
Constants (vectors and matrices)	constant-index-expression	constant-index-expression

## F.6 Counting of Varyings and Uniforms

GLSL ES 1.0 specifies the storage available for varying variables in terms of an array of 4-vectors. Similarly for uniform variables. The assumption is that variables will be packed into these arrays without wasting space. This places significant burden on implementations since optimal packing is computationally intensive. Implementations may have more internal resources than exposed to the application and so avoid the need to perform packing but this is also considered an expensive solution.

OpenGL SC 2.0 therefore relaxes the requirements for packing by specifying a simpler algorithm that may be used. This algorithm specifies a minimum requirement for when a set of variables must be supported by an implementation. The implementation is allowed to support more than the minimum and so may use a more efficient algorithm and/or may support more registers than the virtual target machine.

In all cases, failing resource allocation for variables must result in an error.

The resource allocation of variables must succeed for all cases where the following packing algorithm succeeds:

- The target architecture consists of a grid of registers, 8 rows by 4 columns for varying variables and 128 rows by 4 columns for uniform variables. Each register can contain a float value.
- Variables are packed into the registers one at a time so that they each occupy a contiguous subrectangle. No splitting of variables is permitted.

- The orientation of variables is fixed. Vectors always occupy registers in a single row. Elements of an array must be in different rows. E.g. `vec4` will always occupy one row; `float[8]` will occupy one column. Since it is not permitted to split a variable, large arrays e.g. for varyings, `float[16]` will always fail with this algorithm.
- Variables consume only the minimum space required with the exception that `mat2` occupies 2 complete rows. This is to allow implementations more flexibility in how variables are stored.
- Arrays of size `N` are assumed to take `N` times the size of the base type.
- Variables are packed in the following order:
  1. Arrays of `mat4` and `mat4`
  2. Arrays of `mat2` and `mat2` (since they occupy full rows)
  3. Arrays of `vec4` and `vec4`
  4. Arrays of `mat3` and `mat3`
  5. Arrays of `vec3` and `vec3`
  6. Arrays of `vec2` and `vec2`
  7. Arrays of `float` and `float`
- For each of the above types, the arrays are processed in order of size, largest first. Arrays of size 1 and the base type are considered equivalent. In the case of varyings, the first type to be packed (successfully) is `mat4[2]` followed by `mat4`, `mat2[2]`, `mat2`, `vec4[8]`, `vec4[7]`,...`vec4[1]`, `vec4`, `mat3[2]`, `mat3` and so on. The last variables to be packed will be `float` (and `float[1]`).
- For 2,3 and 4 component variables packing is started using the 1st column of the 1st row. Variables are then allocated to successive rows, aligning them to the 1st column.
- For 2 component variables, when there are no spare rows, the strategy is switched to using the highest numbered row and the lowest numbered column where the variable will fit. (In practice, this means they will be aligned to the x or z component.) Packing of any further 3 or 4 component variables will fail at this point.
- 1 component variables (i.e. floats and arrays of floats) have their own packing rule. They are packed in order of size, largest first. Each variable is placed in the column that leaves the least amount of space in the column and

aligned to the lowest available rows within that column. During this phase of packing, space will be available in up to 4 columns. The space within each column is always contiguous.

- If at any time the packing of a variable fails, the compiler or linker must report an error.

Example: pack the following types:

```

varying vec4 a;      // top left
varying mat3 b;      // align to left, lowest numbered rows
varying vec2 c[3];   // align to left, lowest numbered rows
varying vec2 d[2];   // Cannot align to left so align to z column,
                    // highest numbered rows
varying vec2 e;      // Align to left, lowest numbered rows.
varying float f[3];  // Column with minimum space
varying float g[2];  // Column with minimum space (choice of 2,
                    // either one can be used)
varying float h;     // Column with minimum space

```

In this example, the varyings happen to be listed in the order in which they are packed. Packing is independent of the order of declaration.

	x	y	z	w
0	a	a	a	a
1	b	b	b	f
2	b	b	b	f
3	b	b	b	f
4	c	c	g	h
5	c	c	g	
6	c	c	d	d
7	e	e	d	d

Some varyings e.g. `mat4[8]` will be too large to fit. These always fail with this algorithm.

If referenced in the fragment shader (after preprocessing), the built-in special variables (`gl_FragCoord`, `gl_FrontFacing` and `gl_PointCoord`) are included when calculating the storage requirements of varyings.

Only varyings statically used in both shaders are counted.

When calculating the number of uniform variables used, any literal constants present in the shader source after preprocessing are included when calculating the

storage requirements. Multiple instances of identical constants should count multiple times.

Part of the storage may be reserved by an implementation for its own use e.g. for computation of transcendental functions. This reduces the number of uniforms available to the shader. The size of this reduction is hardware specific.

## F.7 Shader Parameters

The following are the minimum values that must be supported by an OpenGL SC 2.0 implementation:

```
const mediump int gl_MaxVertexAttribs = 8;
const mediump int gl_MaxVertexUniformVectors = 128;
const mediump int gl_MaxVaryingVectors = 8;
const mediump int gl_MaxVertexTextureImageUnits = 0;
const mediump int gl_MaxCombinedTextureImageUnits = 8;
const mediump int gl_MaxTextureImageUnits = 8;
const mediump int gl_MaxFragmentUniformVectors = 16;
const mediump int gl_MaxDrawBuffers = 1;
```

## Appendix G

# Packaging and Acknowledgements

### G.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 SC , including links to the header file defining interfaces for the OpenGL SC API (`glsc2.h`) as well as a separate headers (`glsc2platform.h` and `KHRplatform.h`) referenced by OpenGL SC .

Preprocessor tokens `GL_SC_VERSION_n_m`, where `n` and `m` are the major and minor version numbers as described in section 6.1.4, are included in `glsc2.h`. These tokens indicate the OpenGL SC versions supported at compile-time.

### G.2 Acknowledgements

The OpenGL SC 2.0 specification is the result of the contributions of many people, representing a cross section of the embedded computer industry. It builds upon the existing OpenGL ES 2.0 specification which contains a list of OpenGL ES 2.0 acknowledgements. The acknowledgements here represent the contributions made to transform OpenGL ES 2.0 into OpenGL SC 2.0. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:



Bob Schulman, AMD  
Alastair Murray, Codeplay  
Illya Rudkin, Codeplay  
Aidan Fabius, Core Avionic & Industrial Inc.  
John Lawless, Core Avionics & Industrial Inc.  
Tom Malnar, Core Avionics & Industrial Inc.  
John McCormick, Core Avionics & Industrial Inc.  
Greg Szober, Core Avionics & Industrial Inc.  
Steve Viggers, Core Avionics & Industrial Inc.  
Ken Wenger, Core Avionics & Industrial Inc.  
Steve Ramm, Imagination Technologies  
Andy Southwell, Imagination Technologies  
Erik Noreke, Independent (working group chair)  
Nakhon Baek, Kyungpook National University  
Hwanyong Lee, Kyungpook National University  
Mark Rivers, Mentor Graphics  
Paul Jennings, Presagis  
Cary Ashby, Rockwell Collins  
Doug Singkoffer, Rockwell Collins  
Levi Van Ort, Rockwell Collins  
Anand Balagopalakrishnan, Texas Instruments  
Ajay Jayaraj, Texas Instruments

## **G.3 Document History**

### **G.3.1 Version 2.0.0, April 19, 2016**

Initial revision of the full specification, based on OpenGL ES 2.0.25.

# Index

ACTIVE\_TEXTURE, 59, 72, 109  
ActiveTexture, 33, 59, 59  
ALIASED\_POINT\_SIZE\_RANGE, 44  
ALPHA, 85, 133  
ALWAYS, 81, 82, 123  
ARRAY\_BUFFER, 24–27, 109  
ARRAY\_BUFFER\_BINDING, 26  
  
BACK, 50, 81, 88, 118  
BindBuffer, 24, 24, 27  
BindFramebuffer, 94, 94, 103  
BindRenderbuffer, 96, 96  
BindTexture, 33, 59, 71, 71, 72  
BLEND, 83  
BlendColor, 85  
BlendEquation, 83, 83  
BlendEquationSeparate, 83, 83  
BlendFunc, 84, 84  
BlendFuncSeparate, 84, 84  
BLUE, 133  
break, 151  
BUFFER\_SIZE, 24, 26  
BUFFER\_USAGE, 24, 26  
BufferData, 25, 25, 27  
BufferSubData, 26, 27, 144  
bvec2, 32  
BYTE, 22  
  
CCW, 50, 118  
CheckFramebufferStatus, 102, 103, 103, 105  
CLAMP\_TO\_EDGE, 64, 65, 74  
  
Clear, 89, 89, 90  
ClearColor, 89  
ClearDepthf, 89  
ClearStencil, 89  
COLOR\_ATTACHMENT0, 77, 91, 95, 98, 102, 110  
COLOR\_BUFFER\_BIT, 89, 90  
ColorMask, 88, 88, 105  
COMPRESSED\_TEXTURE\_FORMATS, 62  
CompressedTexSubImage2D, 59, 62, 63, 68  
condition, 152  
constant-expressions, 152  
constant-index-expression, 152–154  
constant-index-expressions, 152, 153  
CONSTANT\_ALPHA, 85  
CONSTANT\_COLOR, 85  
CONTEXT\_LOST, 15–17  
continue, 151  
CreateProgram, 28  
CULL\_FACE, 50  
CullFace, 50, 50, 52  
CURRENT\_VERTEX\_ATTRIB, 112, 113  
  
DECR, 81  
DECR\_WRAP, 81  
DEPTH\_ATTACHMENT, 77, 95, 98, 102, 110  
DEPTH\_BUFFER\_BIT, 89, 90  
DEPTH\_COMPONENT16, 101

- DEPTH\_TEST, 82
- DepthFunc, 82
- DepthMask, 88, 88, 105
- DepthRange, 38, 109
- Disable, 50, 52, 79, 80, 82, 83, 86
- DisableVertexAttribArray, 23, 112
- DITHER, 86
- do-while, 152
- DONT\_CARE, 107, 129
- DrawArrays, 19, 23, 23, 26, 27, 29, 104
- DrawRangeElements, 19, 23, 23, 26, 27, 29, 104
- DST\_ALPHA, 85
- DST\_COLOR, 85
- DYNAMIC\_DRAW, 24, 25
- ELEMENT\_ARRAY\_BUFFER, 27, 109
- Enable, 50, 52, 79, 80, 82, 83, 86, 108
- EnableVertexAttribArray, 22, 112
- EQUAL, 81, 82
- EXTENSIONS, 111, 112, 149
- FALSE, 32, 36, 80, 121
- false, 74
- FASTEST, 107
- Finish, 106, 106, 141, 144
- FLOAT, 22, 24, 115
- float, 29, 152
- Flush, 106, 141
- for, 152
- for\_header, 152
- FRAMEBUFFER, 79, 93, 94, 98, 103, 105, 110
- FRAMEBUFFER\_ATTACHMENT\_-  
OBJECT\_NAME, 98, 99, 102, 110
- FRAMEBUFFER\_ATTACHMENT\_-  
OBJECT\_TYPE, 98, 99, 101, 102, 110
- FRAMEBUFFER\_ATTACHMENT\_-  
TEXTURE\_LEVEL, 69, 70, 99, 100, 110
- FRAMEBUFFER\_BINDING, 69, 91, 94, 98, 104, 105
- FRAMEBUFFER\_COMPLETE, 103, 105
- FRAMEBUFFER\_INCOMPLETE\_ATTACHMENT, 102
- FRAMEBUFFER\_INCOMPLETE\_DIMENSIONS, 102
- FRAMEBUFFER\_INCOMPLETE\_MISSING\_ATTACHMENT, 102
- FRAMEBUFFER\_UNDEFINED, 102
- FRAMEBUFFER\_UNSUPPORTED, 102, 103
- FramebufferRenderbuffer, 97, 98, 103, 105
- FramebufferTexture2D, 98, 99, 103, 105
- FRONT, 50, 81, 88
- FRONT\_AND\_BACK, 50, 81, 88
- FrontFace, 50, 50, 74
- FUNC\_ADD, 83, 84, 86, 124
- FUNC\_REVERSE\_SUBTRACT, 83, 84
- FUNC\_SUBTRACT, 83, 84
- GenBuffers, 24, 25, 25
- GENERATE\_MIPMAP\_HINT, 107
- GenerateMipmap, 70, 70, 107
- GenFramebuffers, 94, 95, 95
- GenRenderBuffers, 96
- GenRenderbuffers, 96, 97, 97
- GenTextures, 71, 72, 72
- GEQUAL, 81, 82
- Get, 39, 108
- GetAttribLocation, 30, 30
- GetBooleanv, 80, 108, 113
- GetBufferParameteriv, 109

- GetError, **14**, 14, 16
- GetFloatv, **10**, 80, **108**, 113
- GetFramebufferAttachmentParameteriv, **105**, **109**, 110
- GetGraphicsResetStatus, **15**, 16
- GetIntegerv, **43**, 91, 97, **108**, 113
- GetnUniform\*, 113
- GetnUniformfv, **113**
- GetnUniformiv, **113**
- GetProgramiv, **112**
- GetRenderbufferParameteriv, 97, **110**, 111
- GetString, **111**, 112
- GetTexParameter, 59, **109**
- GetUniformLocation, **31**, 33
- GetVertexAttribfv, **112**
- GetVertexAttribiv, **112**
- GetVertexAttribPointerv, **113**
- gl\_FragColor, 75
- gl\_FragCoord, 74
- gl\_FragData, 75
- gl\_FragData[0], 75
- gl\_FrontFacing, 74
- gl\_PointCoord, 44
- gl\_PointSize, 44
- gl\_Position, 34, 37
- GL\_SC\_VERSION\_n.m, 158
- GREATER, 81, 82
- GREEN, 133
- GUILTY\_CONTEXT\_RESET, 16
- Hint, **106**
- if-then, 151
- if-then-else, 151
- IMPLEMENTATION\_COLOR\_READ\_FORMAT, 91
- IMPLEMENTATION\_COLOR\_READ\_TYPE, 91
- INCR, 81
- INCR\_WRAP, 81
- INNOCENT\_CONTEXT\_RESET, 16
- inout, 152
- INT, 92
- int, 152
- INVALID\_ENUM, 15, 17, 59, 60, 103, 105, 110, 111
- INVALID\_FRAMEBUFFER\_OPERATION, 17, 104, 105
- INVALID\_OPERATION, 17, 24, 25, 29–33, 56, 59–61, 63, 71, 91, 94, 96–99, 105, 109–111, 113
- INVALID\_VALUE, 15, 17, 21–23, 26, 32, 39, 45, 53, 60–63, 79, 89, 97, 99, 105, 112, 113
- INVERT, 81
- IsEnabled, 79, **108**, 113
- KEEP, 81, 82, 123
- LEQUAL, 81, 82
- LESS, 81, 82, 123
- level zero array, 60
- LINE\_LOOP, 19
- LINE\_STRIP, 19
- LINEAR, 64, 67–69, 71, 73, 74, 100
- LINEAR\_MIPMAP\_LINEAR, 64, 68–70, 100
- LINEAR\_MIPMAP\_NEAREST, 64, 68, 70, 100
- LINES, 19
- LineWidth, **45**
- LOSE\_CONTEXT\_ON\_RESET, 16, 137
- m, 158
- mat2, 30
- mat3, 30
- mat4, 30
- MAX\_COMBINED\_TEXTURE\_IMAGE\_UNITS, 34, 59, 109

- MAX\_FRAGMENT\_UNIFORM\_VECTORS, 73
- MAX\_RENDERBUFFER\_SIZE, 97, 105
- MAX\_TEXTURE\_IMAGE\_UNITS, 34, 58, 74
- MAX\_TEXTURE\_SIZE, 60
- MAX\_VARYING\_VECTORS, 34
- MAX\_VERTEX\_ATTRIBS, 21–24, 30, 112, 113
- MAX\_VERTEX\_TEXTURE\_IMAGE\_UNITS, 34, 58
- MAX\_VERTEX\_UNIFORM\_VECTORS, 30
- MIRRORED\_REPEAT, 64, 65
- n, 158
- NEAREST, 64, 66, 68, 69, 73, 74, 100
- NEAREST\_MIPMAP\_LINEAR, 64, 68–71, 100
- NEAREST\_MIPMAP\_NEAREST, 64, 68, 69, 100
- NEVER, 81, 82
- NICEST, 107
- NO\_ERROR, 14–16, 134
- NO\_RESET\_NOTIFICATION, 16
- NONE, 101, 110, 136
- NOTEQUAL, 81, 82
- NULL, 24, 114, 115
- NUM\_COMPRESSED\_TEXTURE\_FORMATS, 62
- NUM\_PROGRAM\_BINARY\_FORMATS, 29
- ONE, 85, 86, 124
- ONE\_MINUS\_CONSTANT\_ALPHA, 85
- ONE\_MINUS\_CONSTANT\_COLOR, 85
- ONE\_MINUS\_DST\_ALPHA, 85
- ONE\_MINUS\_DST\_COLOR, 85
- ONE\_MINUS\_SRC\_ALPHA, 85
- ONE\_MINUS\_SRC\_COLOR, 85
- out, 152
- OUT\_OF\_MEMORY, 15, 17, 26, 97, 105
- PACK\_ALIGNMENT, 91, 126
- PixelStore, 54, 91, 93
- PixelStorei, 53, 53
- POINTS, 19
- POLYGON\_OFFSET\_FILL, 52
- PolygonOffset, 51
- PROGRAM\_BINARY\_FORMATS, 29
- ProgramBinary, 28, 28
- ReadnPixels, 53, 55, 90, 91, 91, 94, 104
- RED, 55, 56, 74, 92, 133
- RENDERBUFFER, 96–98, 110
- RENDERBUFFER\_ALPHA\_SIZE, 111
- RENDERBUFFER\_BINDING, 97, 105, 110
- RENDERBUFFER\_BLUE\_SIZE, 111
- RENDERBUFFER\_DEPTH\_SIZE, 111
- RENDERBUFFER\_GREEN\_SIZE, 111
- RENDERBUFFER\_HEIGHT, 111
- RENDERBUFFER\_INTERNAL\_FORMAT, 111
- RENDERBUFFER\_RED\_SIZE, 111
- RENDERBUFFER\_STENCIL\_SIZE, 111
- RENDERBUFFER\_WIDTH, 111
- RenderbufferStorage, 97, 97, 105
- RENDERER, 111
- REPEAT, 64, 66, 67, 71, 121
- REPLACE, 81
- RESET\_NOTIFICATION\_STRATEGY, 16
- RG, 55, 56, 74
- RGB, 55–58, 74, 85

- RGB565, 101
- RGB5\_A1, 101
- RGBA, 55–58, 74, 91
- RGBA4, 101, 135
- SAMPLE\_ALPHA\_TO\_COVERAGE, 79
- SAMPLE\_BUFFERS, 43, 44, 48, 52, 79, 86–88, 95
- SAMPLE\_COVERAGE, 79, 80
- SAMPLE\_COVERAGE\_INVERT, 79, 80
- SAMPLE\_COVERAGE\_VALUE, 79, 80
- SampleCoverage, 80
- sampler2D, 33
- SAMPLES, 43, 95
- Scissor, 79
- SCISSOR\_TEST, 79
- SHADING\_LANGUAGE\_VERSION, 111
- SHORT, 22
- SRC\_ALPHA, 85
- SRC\_ALPHA\_SATURATE, 85
- SRC\_COLOR, 85
- STATIC\_DRAW, 24, 25, 116
- STENCIL\_ATTACHMENT, 77, 95, 98, 102, 110
- STENCIL\_BUFFER\_BIT, 89, 90
- STENCIL\_INDEX8, 101
- STENCIL\_TEST, 80
- StencilFunc, 80, 81, 82, 141
- StencilFuncSeparate, 80, 81, 82
- StencilMask, 88, 88, 105, 141
- StencilMaskSeparate, 88, 88, 105
- StencilOp, 80, 81, 82
- StencilOpSeparate, 80, 81, 82
- STREAM\_DRAW, 24, 25
- TexParameter, 63, 144
- TexStorage2D, 59, 59–62
- TexSubImage\*, 144
- TexSubImage2D, 53–55, 59, 61, 61, 62, 68, 90, 92
- TEXTURE, 99, 110
- TEXTURE*i*, 59
- TEXTURE0, 59, 122
- TEXTURE\_2D, 33, 61, 63, 70–72, 99, 109, 120
- TEXTURE\_IMMUTABLE\_FORMAT, 59
- TEXTURE\_MAG\_FILTER, 64, 69, 71
- TEXTURE\_MIN\_FILTER, 64, 66–69, 71, 100
- TEXTURE\_WRAP\_S, 63, 64, 66, 67
- TEXTURE\_WRAP\_T, 63, 64, 67
- TRIANGLE\_FAN, 20
- TRIANGLE\_STRIP, 19
- TRIANGLES, 20
- TRUE, 22, 32, 59, 80, 88, 137
- true, 74
- Uniform, 11, 31
- Uniform\*, 31–33
- Uniform\*f{v}, 31, 32
- Uniform\*i{v}, 31, 32
- Uniform1f, 11
- Uniform1i, 11
- Uniform1i{v}, 32, 33
- Uniform1iv, 32
- Uniform2f, 11
- Uniform2f{v}, 32
- Uniform2i, 11
- Uniform2i{v}, 32
- Uniform3f, 11
- Uniform3i, 11
- Uniform4f, 10, 11
- Uniform4f{v}, 32
- Uniform4i, 11
- Uniform4i{v}, 32

UniformMatrix\*, 32  
 UniformMatrix3fv, 32  
 UniformMatrix{234}fv, 31, 32  
 UNKNOWN\_CONTEXT\_RESET, 16  
 UNPACK\_ALIGNMENT, 54, 56, 126  
 UNSIGNED\_BYTE, 22, 55, 56, 91, 92  
 UNSIGNED\_SHORT, 22, 23, 57  
 UNSIGNED\_SHORT\_4\_4\_4\_4, 55–57, 92  
 UNSIGNED\_SHORT\_5\_5\_5\_1, 55–57, 92  
 UNSIGNED\_SHORT\_5\_6\_5, 55–57, 92  
 UseProgram, 29, 29, 34, 36  
  
 vec2, 29  
 vec3, 29  
 vec4, 29, 32  
 VENDOR, 111  
 VERSION, 111, 112  
 VERTEX\_ATTRIB\_ARRAY\_–  
     BUFFER\_BINDING, 26, 112  
 VERTEX\_ATTRIB\_ARRAY\_EN-  
     ABLED, 112  
 VERTEX\_ATTRIB\_ARRAY\_NOR-  
     MALIZED, 112  
 VERTEX\_ATTRIB\_ARRAY\_–  
     POINTER, 113  
 VERTEX\_ATTRIB\_ARRAY\_SIZE, 112  
 VERTEX\_ATTRIB\_ARRAY\_STRIDE,  
     112  
 VERTEX\_ATTRIB\_ARRAY\_TYPE,  
     112  
 VertexAttrib, 21  
 VertexAttrib\*, 21, 29  
 VertexAttrib1\*, 21  
 VertexAttrib2\*, 21  
 VertexAttrib3\*, 21  
 VertexAttrib4\*, 21  
 VertexAttribPointer, 22, 22, 26, 113  
 Viewport, 38  
  
 while, 152  
 ZERO, 81, 85, 86, 124